2. Componentes

En este segundo tema dedicado a los **componentes** vamos a empezar por generarlos. Veremos qué herramientas tenemos y cómo preconfigurarlas para que tengamos siempre **componentes** homogéneos y modernos en **Angular**. Es hora de entrar dentro de un **componente** y ver a qué se dedica y qué hay dentro del **componente**.

Veremos que hay dos partes bien diferenciadas: una dedicada a la presentación, que será una vista, un **template** en **HTML**, y otra dedicada a la lógica, que será una clase en **TypeScript**. Por último, nos dedicaremos a la presentación para ver qué tenemos. Tendremos **tuberías** para transformar los datos y que aparezcan como nosotros queremos, y **estilos** también para darle su magia.

2.1 Generación de componentes.

Bueno, pues vamos a empezar con la parte fundamental de **Angular**.

Tiramos la piedra angular, si me permite la broma, que son los **componentes**.

Los **componentes** son piezas visuales, que podremos definir con **HTML**, un **HTML** quizá no muy estándar, ¿vale?

Y en este **HTML** vamos a incrustar una serie de elementos que lo van a comunicar con una **clase**, una **clase** que tendrá pues un nombre, llaves, llaves, y tendrá propiedades y métodos, ¿vale?

Pues lo que ocurrirá es que esta plantilla o vista se va a comunicar con esta **clase** que vamos a llamar **controlador**. Pero todo ello será un **componente**, de forma que los datos de las propiedades o los métodos sean accesibles para esta vista.

La vista tendrá también asociado una serie de **estilos**, ¿vale? Podríamos decir que esto es la parte **CSS** del **componente**. Estos **estilos** se aplicarán aquí durante la vista y en muchos casos en función de los datos que tenga en la **plantilla**, en el **controlador**.

También tenemos que decir que es testeable este **controlador**. También la **plantilla** es testeable. Y estos tests opcionales se pueden crear de manera unitaria para cada **componente** y comprobarán la funcionalidad que aquí se haya hecho.

Por tanto, tenemos como cuatro cosas que definen un **componente** y podrían estar cada uno en un fichero distinto o se podrían agrupar. En **Angular Moderno** se sugiere que todo esto vaya agrupado y que esta parte se haga de manera voluntaria y específica en distintas tecnologías.

Puede ser la clásica **Karma Jasmine** o la moderna **Jest**. Bien, y de todo esto, una de las cosas con las que trabajaremos será que aparezca en las **clases** una decoración específica que diga que esto es un **componente** y dentro de ese **componente**, ya ves, ya ves, meterá una serie de propiedades de las cuales la fundamental será la que llamemos **selector**.

El **selector** será el nombre por el que se conozca a este **componente** fuera de sí mismo. Y llevará siempre un prefijo y un nombre. Por ejemplo, en mi aplicación el prefijo será lab y el nombre será lo que toque. Por ejemplo, cuando haga un **Header**, pues será así, **header**. Y esto será lo que se incluya en otro **componente**.

Para que todo esto sea fácil y no tenga que generar estos ficheros y ponerle estas informaciones así de manera manual, toda esta parte la puedo generar. Usando el **CLI** y desde la línea de comandos le diré ng, generame, todo esto se puede poner explícito, generate o abreviado, g, un **componente** con un determinado nombre.

No hace falta poner el prefijo, el prefijo ya se lo agrega él. Y a partir de ahí, diversas opciones. Opción 1, opción 2, etc. Todas estas opciones pueden ser controladas desde los **Schematics** en el fichero **angular.json**.

Bueno, vamos a ver esto en la práctica, que es más fácil de lo que parece.

2.1.1 EL CLI y el Angular.json

Bueno, pues vamos a empezar a estudiar la parte fundamental de **Angular**, que son los **componentes**.

Habíamos visto que en el **index.html**, dentro del **body**, lo que me encontraba no era quizá un **HTML** estándar esperado como un **div**, una **p**, etc., sino un elemento con un prefijo y un nombre, un nombre compuesto, que para nada es estándar, pero que lo habíamos vuelto a ver en otro fichero llamado **app.component.ts**, donde aparecía dentro de lo que llamaremos un **selector**.

Un **selector** no será más que el nombre por el cual será conocido un **componente** dentro de otro **componente**. Porque de eso se trata, de componer aplicaciones en base a piezas reutilizables.

La primera pieza, este **lab-root**, tiene el prefijo que hayamos escogido para la aplicación y como nombre genérico y por convenio **root**, de raíz, y es lo que nos ha regalado el generador cuando nos ha creado la aplicación.

Vamos a estudiar un poquito la anatomía de este **componente**, viendo que es una clase decorada. Una clase **TypeScript** normal y corriente, pero que lleva fuera una decoración, que es una función prefijada con la **@**, esto es algo propio de **TypeScript**, y específicamente de **Angular**, lo que viene a ser es esta configuración.

Dentro de esta configuración, por ahora, nos interesa el **selector**, que es como va a ser conocido fuera, y la **template**, que será el contenido en un **HTML** más o menos estándar, que sustituya, de alguna forma, a las apariciones de este **selector** dentro del **HTML** en producción.

Por supuesto, los **componentes**, al ser clases, tendrán propiedades y métodos, y veremos, estudiaremos, cómo invocar a esas propiedades desde las **plantillas**, pero no es este ahora el momento.

Ahora vamos a ver cómo el generador me va a ayudar a crear, a generar, nuevos **componentes**. Para ello, desde la línea de comandos, lanzaré el programa **ng**, pidiéndole que genere, abreviadamente con **g**, un **componente**, abreviadamente con **c**, si no lo podéis escribir todo junto, todo largo, **ng generate component**, y después, poniéndole un nombre.

Yo voy a empezar por generar el **componente** número 1, va a ser una prueba que después borraré, y veo lo que me crea, lo que me genera. Una carpeta, con el nombre del **componente**, y dos ficheros. Aparecen dos ficheros y podrían aparecer hasta cuatro. ¿De qué depende eso? Pues depende de las opciones que haya seleccionado durante la generación, o de los valores que tenga por defecto.

En cuanto a opciones que se pueden poner durante la generación, sería algo así como, **Angular**, génerame un **componente**, voy a poner en este caso el **componente 2**, y me lo vas a hacer en modo plano (**flat**). Al pedirle que lo haga plano (**flat**), lo que hará será crearme el **componente 2**, pero sin crearle una carpeta.

Esto sólo es a modo de ejemplo, para que veáis cómo estas opciones manipulan el cómo se van a generar los **componentes**.

Una tercera opción sería incluso decirle que no me cree pruebas. Voy a poner, por ejemplo, **ng generate component**, voy a poner el 3, le digo que sea plano (**flat**) y sin test, por ejemplo, con **skip-tests**, porque quiero hacer las pruebas de tipo **end-to-end** o hacer pruebas unitarias sólo para otras cosas que no sean **componentes**.

Bueno, aquí ha aparecido el **componente 3**, sin el hermano de pruebas y sin estar en una carpeta. Si a mí me gustase esta manera de generar **componentes**, tendría que ponerlo de cada vez.

O bien, ir al **angular.json** y encontrar aquí una sección llamada **schematics**, donde para los **componentes**, y para otras cosas, en este caso para los **componentes**, puedo dar los valores que yo entiendo por defecto. Por ejemplo, al haberle puesto **inlineTemplate: true** e **inlineStyle: true**, pues tanto las **plantillas** como los **estilos** aparecen dentro del **componente** y no como punteros externos.

¿Qué otras modificaciones podría poner? Pues si me ha gustado lo de **Flat**, podría ponerle perfectamente aquí **flat: true**, que es el valor que quiero cambiar. Si me ha gustado lo de no tener pruebas unitarias para los **componentes**, pues yo pondré aquí que me quite las pruebas unitarias.

E incluso otras modificaciones que ahora mismo no tienen sentido para vosotros, pero que forman parte de configuraciones modernas de **Angular** y que vamos a utilizar aquí. Solo estoy cambiando aquellos valores que por defecto tienen otra configuración previa. Es decir, si nadie dice nada, **Flat** hubiera sido **False**. Si nadie dice nada, **InlineTemplate** hubiera sido **False**, etc.

Ahora que tenemos configurados los **Schematics** a nuestro gusto, ya vamos a crear dos **componentes** un poquito útiles, no como esos que hacía a modo prueba.

Voy a crear un **componente** para lo que será la cabecera de mi aplicación y otro para lo que será el pie. Y de paso veremos cómo utilizarlos.

Voy a pedirle, por favor, **ng Generate Component** y podría poner ya directamente **header**, o si ahora quiero agruparlos de alguna manera un poco especial, por ejemplo, los **componentes** que voy a utilizar una sola vez y para cosas muy serias como podría ser la cabecera y el pie, suelo meterlos en una carpeta llamada **core**.

Esto es siguiendo un convenio desde tiempos inmemoriales de **Angular** y que me permite tenerlo así agrupado. Fijaos que no es lo mismo que el **Flat** porque esta carpeta ya no se llama como el **componente** y me permitirá, por ejemplo, tener varios **componentes** agrupados en la misma carpeta porque tengan un sentido, vamos a decir, funcional.

2.2 Anatomía de un componente: plantillas y lógica.

Vale, pues una vez que hemos visto cómo es un **componente** por dentro, vamos a ver ahora cómo es su uso incrustado dentro de otro **componente**. Y este podría tener otro dentro, podríamos crear nuestros **componentes** como nos parezca bien.

Para ello sólo necesitamos dos cosas. Una, en el **HTML** tenemos que invocar al **selector**, que te recuerdo, que es el prefijo más el nombre. Pero para que eso sea conocido, vamos a decir que en la parte decoradora del **componente** habrá un array de importaciones, y en este array de importaciones tendremos que incrustar el nombre de la clase, **Class Name**.

Con esto podemos usar un **componente** dentro de otro, invocando su **selector**, pero para que sea conocido, previamente habremos importado también su **Class Name**.

Todo lo que hagamos en los **componentes** será con la intención de vincular dos mundos, el de las **templates**, el de las plantillas, con el de los controladores, las clases en **HTML**.

¿Y qué vivirán cada uno de ellos? Pues en las clases, esencialmente, vivirán propiedades, que tendrán nombre, en este caso, genérico **propiedad**, o métodos, que tendrán también un nombre y una implementación, en este caso se llama **método**.

¿Y qué hay en el lado de las **plantillas**? Bueno, pues en el lado de las **plantillas**, por un lado, tendremos una simbología un poquito rara, como es esta de las dobles llaves, en donde vincularemos una **propiedad**. De esa manera, el valor de esa **propiedad** se incrustará, aparecerá dentro del **HTML**.

Alternativamente, para cuando lo que necesitemos sea dar un valor a un atributo, usaremos los corchetes **[]**. De forma que un atributo del **HTML** se asociará con el valor de una **propiedad**.

¿Y qué hay de los métodos? Pues los métodos me servirán para vincular eventos del **HTML**, un evento, con una invocación, esto es importante, el método tendrá que invocarlo, con los argumentos que fuese necesario, o con paréntesis vacíos, si no los lleva.

Pero de esta manera, lo que se dice es que hay una comunicación desde la **plantilla** hacia la **clase**, hacia el **controlador**, mientras que por aquí lo que hay es una comunicación desde el **controlador** hacia la **plantilla**.

Datos de propiedades e invocaciones a métodos, con esta simbología, llaves **{}**, corchetes **[]** y paréntesis **()**.

2.2.1 Plantillas para las vistas

A partir de ahora voy a procurar tener siempre dos terminales abiertas. Puedo tenerlo con la pantalla dividida en dos, o aquí para el curso me va a ser más cómodo, en lugar de dividirlo así en dos, agregar una segunda terminal para mantener todo el ancho posible. ¿Por qué? Porque en una de ellas voy a mantener siempre lanzada la aplicación con **npm start**, de forma que se compile y pueda ver también el resultado de dicha compilación.

¿Qué veo aquí? Pues la aplicación tal cual ha arrancado. Aquí puedo tener acceso a los valores actuales del contenido y a cualquier cosa de la consola. A partir de ahora tendremos a un lado el código y al otro la ejecución. En cuanto al código, habíamos dejado dos **componentes** creados, el **header** y el **footer**, y vamos a empezar a utilizar estos conceptos de plantillas unos dentro de otros.

La idea es que si este componente raíz dispone de una cabecera, esa cabecera la sustituya por este contenido de aquí. Así que ahora mismo lo que voy a crear dentro de esta cabecera va a ser un **HTML** que no necesita mucha explicación, y lo que sí necesita explicación es ver su uso dentro de otro componente. Voy a sustituir esta parte de aquí por una invocación a este **selector**.

La invocación al **selector** teóricamente se haría simplemente poniendo **lab-header**, que es como se llama, pero veré que inmediatamente el Visual Studio ya me avisa poniéndome esto en rojo y aquí con un aviso también de rojo, y la compilación falla y la reejecución falla porque dice, oye, el **lab-header** no es algo estándar ni conocido.

Y tiene razón. Para que sea conocido, porque estándar no lo es, lo que tendré que hacer será previamente importarlo en esta sección de la decoración en la que voy a indicar todos y cada uno de los subcomponentes que voy a utilizar. Por ejemplo, en este caso será el **HeaderComponent**.

Atención que lo que aquí se importa es el nombre de la clase, mientras que lo que aquí se usa es el **selector**. Entonces, **selector**, clase. Voy a repetir lo mismo para el **footer**, solo que ahora ya sabiendo esto, pues pondré el **footer** antes de nada. No importa el orden aquí en este array de componentes.

Y lo que puedo hacer es ya simplemente poner aquí el **lab-footer**. Deciros que en versiones modernas de **Angular** se prefiere poner de esta manera abreviada los componentes y vemos que aquí tenemos tanto el **header** como el **footer**.

Y en cuanto al contenido del **footer**, puedo cambiarlo en cualquier momento. Ahora mismo no tiene más que lo que me generó **Angular**, pero puedo sustituirlo por algo un poquito con mayor intención, como por ejemplo el copyright y las cookies.

Vemos aquí en el navegador que el contenido se ha macrosustituido de alguna manera. El **lab-footer** es sustituido, realmente no se sustituye porque se mantiene el **lab-footer** con el contenido que tengo aquí, que es este **footer**, **nav**, **a** y botón.

2.2.2 Propiedades y métodos

Vale, pues vamos a centrar ahora nuestra atención pasando de las **plantillas** hacia los **componentes**. Ya en el primer **componente** generado aparecía aquí una rareza que eran unas llaves **{}** haciendo uso de una **propiedad** que ahora mismo tenemos sin utilizar aquí en el **AppComponent**, pero que lo voy a trasladar, voy a trasladarle esa funcionalidad. Simplemente me llevo esta **propiedad**, podría llamarse de cualquier manera, con que sea pública, que por defecto en **TypeScript** todo lo es. Vale, estas **propiedades** públicas se pueden reutilizar entre dobles llaves **{{}}**, que es lo que llamaremos en **Angular** una expresión de interpolación, un nombre muy raro, para decir que simplemente sustituya el valor que ponga entre las llaves por los contenidos que tengan estas **propiedades** en los **componentes**.

Esto evolucionará, pero por ahora vemos que lo hace. Este **ActivityBookings** tal cual, o lo puedo separar para que se vea mejor, y el caso es que aquí perfectamente me aparece este valor de esta **propiedad** en esta expresión de interpolación.

Siguiendo con lo mismo y llegando al **footer** veo que me encuentro aquí, por ejemplo, con un ancla para que me lleve a la página web y al nombre del autor. La página web y el nombre del autor realmente podrían estar como **propiedades** sueltas o aparecer como **propiedades** de un objeto un poquito más complejo. De nuevo esto es **TypeScript** puro y duro. Lo que sí que puedo hacer en este caso es acceder a las subpropiedades, es decir, puedo llegar a **author.name**. Muy bien, con lo cual esto también sigue funcionando. Voy a poner aquí un punto para que se vea que estas cosas cambian inmediatamente.

¿Qué ocurre con el **href**? Bueno, en el **href** obviamente yo podría tener el valor de la URL a la que quiero ir a visitar. Si hago clic aquí, pues esto aparecerá en esa página web, pero la intención es de nuevo hacerlo dinámico. Para ello lo que tengo que ver es que cuando tengo un atributo de este estilo puedo seguir con la misma estrategia de poner dobles llaves **{{}}** y ahora meter aquí pues **author.homepage**. Inmediatamente, si inspecciono veo que efectivamente el **href** se ha macrosustituido correctamente y que por cierto funciona correctamente, pero cuando tengo que dar valores a los atributos **Angular** me ofrece otra opción, que es la opción de los corchetes **[]**. Es una opción que queda un poquito más limpia y que es la que se recomienda. En esta situación, al meter un atributo cualquiera entre corchetes **[]**, convierto inmediatamente, sin necesidad de las dobles llaves **{{}}**, convierto su valor en una expresión evaluable. Es decir, no necesito ponerle ya las dobles llaves **{{}}** aquí porque he puesto aquí los corchetes **[]**. De esta manera este atributo se dice que toma un valor evaluado que en este caso es exactamente lo mismo. ¿Podemos hacer más cosas? Pues claro que sí. Por ejemplo, aquí tenemos el año 2024 y una idea bastante razonable sería que ese año se calculase automáticamente.

Así que efectivamente las **propiedades** podrían ser funciones de alguna manera. Así que yo ahora voy a poner aquí que esto es la **propiedad year** que se ha rellenado con estos valores automáticamente.

Y aquí tenemos que sigue manteniendo el 2024. Si yo quisiese comprobar que esto efectivamente funciona, pues ya lo veis, que ahí tendría un año más de lo que le corresponde en el momento de la grabación.

Bueno, muy bien. ¿Y qué ocurre con este botón de aceptar **cookies**? Bueno, pues por ahora vamos a ver que podríamos tener cualquier método, como por ejemplo el **onCookiesAccepted**, que simplemente ponga en la consola **cookiesAccepted**.

Yo lo que necesito ahora es vincular la ejecución de este método con un evento en la **plantilla**. Los eventos son los estándar de **HTML**, como por ejemplo el click,

pero ahora el convenio me lleva a ponerlos entre paréntesis **()** en lugar de corchetes **[]**.

Corchetes **[]** es para evaluar atributos y paréntesis **()** es para invocar métodos. El método simplemente tendría que hacer la llamada con los paréntesis adecuados, porque esto es una invocación a un método.

Y a partir de este momento, cuando yo haga click en **Accept Cookies**, debería poner **cookies accepted**, que es lo que ejecuta este método. Así que hemos visto lo básico de los **componentes** y sus **plantillas**, que es poner datos que aparezcan en sus clases o invocar a métodos ante los eventos de usuario.

2.3 Presentación de datos

A la hora de presentar los datos, puede que no tengan el formato perfecto que a mí me gustaría. Puede que yo tenga aquí una fecha que ponga, pero que a mí me gustaría presentarlo de otra manera. Cuando invoco a esta propiedad, puedo aplicarle una transformación a partir de algo que llamaremos un **pipe**, una tubería que se pone con el símbolo genérico este del | (alt gr+1), que me permite invocar ahora aquí a una función que tiene que estar importada previamente y predefinida, como pueda ser la función **date**, que después pueda admitir argumentos.

Estas tuberías me permiten modificar cómo se va a ver algo. Lo que saldrá de aquí será 2025 sin necesidad de que esto haya sido modificado. Es decir, las tuberías que están predefinidas en **Angular** y se pueden crear muchas más. Por ejemplo, podremos tenerlas para monedas, para pasar a **upper** o **lowercase**... Bueno, todas estas tuberías lo que hacen es manipular el cómo se va a ver el resultado sin modificar el origen.

Bueno, hablar de **HTML** y no hablar de estilos parece que no es posible, ¿verdad? Así que vamos a hablar de lo básico que tenéis que saber o que tienes que saber a la hora de aplicar **CSS** a tu **HTML**. Lo primero es que estos **CSS** pueden venir de manera genérica para la aplicación de dos maneras. Desde un fichero llamado **styles.css**, que es hermano del **index.html**, y esto funciona de la manera más clásica posible. Y es que estos estilos se aplican automáticamente a toda la aplicación y a todas partes de este **index.html**.

Pueden venir también de algo que hayas instalado, de un framework de terceros, que hayas metido quizá en **node-modules**, que hayas instalado con **npm install**, y que habrá que configurar dentro del **angular.json**. El **angular.json** me permite que una serie de assets, cosas que tenemos, se incorporen tal cual. La novedad la verás en que cualquier componente pueda tener su propio **CSS** local, que se aplicará exclusivamente a este componente.

Se pueden hacer cosas para que se hereden en subcomponentes y demás, pero en principio este es el funcionamiento por defecto, que lo que hace es que formen una unidad este componente con su estilo, de manera que puedan ser reutilizados en varios sitios. Es la parte que modifica la presentación del componente.

2.3.1 Datos y transformación

Bueno, pues vamos a darle algo de funcionalidad a esta aplicación, ¡¡que por ahora en su core solo me muestra aquí el Angular works!!. Y para ello voy a generar un nuevo componente, en este caso le voy a llamar **bookings**, y como me va a permitir generar varias reservas, o voy a hacer varias cosas con las reservas, voy a crear otro componente dentro de esa carpeta, es decir, no estoy utilizando el flat porque esta carpeta va a albergar más cosas que el **bookings.component**.

El **bookings.component** lo voy a poner aquí abajo. Ahora mismo, si lo quiero utilizar, solo debo hacer una cosa, que es importarlo, y ya puedo sustituir esto por una selección del **lab-bookings**. A partir de este momento ya tengo aquí el lugar del "Angular works", pues el "bookings works". Claro que esta no es una gran mejora, ya puedo deshacerme del **app.component** en este momento, y voy a empezar a pensar en qué voy a poner aquí.

Lo que voy a poner para empezar va a ser datos sobre la actividad a la cual quiero hacerle la reserva. Normalmente esto será un objeto complejo y debería tener esto fuertemente tipado. Para tipificarlo, creo siempre un fichero, o los que necesite, en una carpeta que a mí me gusta llamar **domain**. Te he copiado aquí un fichero, que destino uno para cada tipo de dato con el que trabajo, pero esto es puro **TypeScript**, así que entiendo que tú lo entenderás. A partir de este momento puedo tipificar esta propiedad y rellenarla con datos adecuados. Bien, a partir de este momento puedo utilizar los datos de esa actividad y presentarlos inmediatamente.

Pero no hemos llegado hasta aquí para hacer tan poquita cosa.

Por ejemplo, vamos a suponer que tengo otra línea más abajo, esta vez por ejemplo con un **div**, y en la cual voy a presentar una serie de informaciones. Para empezar, podría poner, por supuesto, la localización. Si quisiera continuar mostrando información, como por ejemplo el precio, bastante interesante, ¿verdad? Lo haría así, y aquí me aparecen los 100 que aparecen por aquí, que no se sabe muy bien qué son. Bueno, yo ahora, sin entrar en problemas de internacionalización, os quiero presentar una técnica de **Angular** para modificar el cómo se ve la información, y es la técnica de los **pipes**. Para utilizar un **pipe**

necesito hacer tres cosas. La primera, poner el símbolo del **pipe** |, que se pone con alt y el 1, igual que con el 2 pones la @ y con el 3 el #. Esto pondrás el **pipe**. Después invocarás a un **pipe**, que tienes que saberte que existe. Por ahora simplemente te explico cómo funciona **Angular**, no tanto toda la sintaxis disponible, pero como cualquier otra cosa, antes de usarla necesitaré importarla. A partir de este momento **Angular** lo que hace es ejecutar una función que toma como entrada este valor y produce una salida sin modificar este valor, que aquí sigue siendo un número 100, ahora me aparece con un $100.00.

Repito, por ahora no vamos a internacionalizar, pero os imaginaréis que para las fechas hay cosas similares. Entonces tendré el **pipe date**, que de nuevo también tengo que importar. Una vez hecho eso, la fecha aparece ya más o menos formateada. ¿Se puede mejorar ese formato? Sí. Igual que el del current, sí. Se pueden mejorar, por ejemplo, mediante atributos que van después de dos puntos y después del nombre del **pipe**. Así que ahora mismo ya tengo 15 agosto, sin letras, y el año 2025. Sólo para que veáis que esto tiene más posibilidades, voy a poner aquí también, por ejemplo, el status de la actividad.

Pues, por ejemplo, con el **uppercase**, os podéis imaginar que existe **lowercase**, **titlecase**, etc. Y ahí aparece. Publish it.

2.3.2 Custom pipes

Pues, después de ver cómo utilizar **pipes** ofrecidos por el **framework**, vamos a ver cómo crear los nuestros propios. Para ello, voy a dividir aquí la **terminal** en dos, manteniendo la **aplicación** ejecutándose. Y voy a lanzar un nuevo comando, que será **ng generate pipe**, porque así se generan todas las cosas en **Angular**. **Angular**, genérame un **pipe**.

Pero antes de ello, voy a visitar aquí a nuestro viejo conocido, el **angular.json**, que tiene la posibilidad de personalizar también los **schematics**. Voy a agregar uno para los **pipes**, para las tuberías custom que hagamos, diciéndole que, por favor, por ahora no me genere **test**.

Una vez que lanzo el comando, aparecerá un **archivo** que, si lo pongo aquí al lado, verás que se parece bastante a la definición de cualquier **componente**, es una **clase**.

Su apellido, en lugar de **component**, será **pipe**. Tiene aquí una rareza de **programación orientada a objetos**, y es que nos obligan a implementar una **interfaz**. Una **interfaz** que, por otro lado, viene ya pre-implementada, con un único método. Un método público, que viene ya pre-implementado, con un

argumento obligatorio y otros opcionales. Todos ellos vienen pre-asignados como desconocidos, pero esto lo vamos a cambiar.

Mi intención con esta tubería es que el título de la actividad mejore, por ejemplo, adjuntándole la localización. Imagínate que la localización, en lugar de tenerla aquí, quisiese colocarla ahí. Bueno, mis opciones podrían empezar por mezclar **HTML** con expresiones, pero esto, vaya, no queda muy limpio. Por otro lado, podría intentar dejar todo esto en una única expresión. Tampoco me va a quedar muy bien, porque necesitaré, en algún momento, concatenar un texto. Y esta concatenación, pues bueno, limpia, tampoco me queda.

Y una opción que sí que me gusta, sería llevarme esta lógica aquí, a esta transformación, y sustituir todo esto por un **pipe**. Ahora mismo esto ha quedado así un poquito temblando, pero lo vamos a corregir. Lo primero, este método de transformación que recibe un valor que, digamos, es obligatorio, ese valor es lo que está a la izquierda de la tubería. Y yo debería tipificarlo un poquito mejor. Por ejemplo, si yo ya sé que esto va a ser una **activity**, pues lo pongo.

Una vez importado el tipo **activity**, ya puedo empezar a programar. Lo siguiente, no siempre va a ser así, pero es muy frecuente que la respuesta sea una cadena de texto, así que también la pongo. Obviamente ahora se queja porque esto no estoy retornando de ninguna manera una cadena de texto. Vale, voy a hacerlo, por lo menos para que no empiece a protestar. Y a partir de ahí, y para hacerle caso porque no estoy utilizando aún el valor, pues voy a concatenar el texto que tenía pensado hacer antes. De esta manera, ya empieza a cobrar forma.

Sabiendo que **value** es una actividad, no me cuesta nada renombrarlo y decir que esto es una **activity**. De esta manera queda un poquito más homogéneo. Y para finalizar, los argumentos que podrían modificar el cómo se ve esto. Esto sería el equivalente a esta manera en la que llamamos al **date**, pasándole estos argumentos, por ahí nos entrarían. Si no los necesito, también los puedo quitar, no pasa nada.

Me queda ahora la tubería perfectamente definida. Su nombre actúa un poco como el selector, no lleva prefijo, así que sería **activity title**. Este sería su nombre. Como cualquier otra cosa que utilizo con los **componentes standalone**, su contexto lo tienen que definir directamente en **imports**. Así que tengo que importar mi tubería como si de un subcomponente se tratase.

Tanto si empiezas en **Angular** como si ya tienes experiencia previa con **Angular Modular**. Quiero recalcar que los **componentes standalone** necesitan que su

contexto se defina explícitamente en su array de **importaciones**. Quizás sea redundante con las importaciones del fichero, puede ser, pero en cualquier caso el hecho de que en este fichero se haga explícito todas las necesidades, todas las dependencias de un **componente**, me parece notable y una muy buena idea para que desaparezca la magia que aportaban a veces los **módulos**.

El resultado de esto en ejecución es que ahora el título de la actividad incluye ya la localización. En definitiva, hemos visto que podemos ampliar o extender el **framework** creando nuestras propias tuberías que como cualquier otra cosa en **Angular** se generan usando el **CLI**, que pueden personalizarse esa experiencia mediante el **angular.json** y que después se utilizarán en las plantillas siempre previamente declarando esa dependencia como una **importación** para que el contexto sea explícito.

2.3.3 Estilos

Vale. Nuestra **aplicación** ya presenta la información, pero la verdad es que es un poquito fea, ¿verdad? ¿Qué podemos hacer para mejorarlo? Bueno, desde luego agregarle **CSS**.

Yo voy a utilizar aquí el **framework** más minimalista que conozco, que se llama **PicoCSS**, pico de pícolo o de la cosa más pequeña posible. Y para ello también me sirve un poco para mostraros cómo se le agregan estilos a **Angular**. Así que lo primero, si quiero instalar algo de terceros, pues lo hago con **npm install**.

Cuando termina, después de haberse metido en **node_modules**, yo lo puedo agregar en esta sección del constructor, en la cual se le pueden agregar estilos a lo que ya viene. La ruta donde está, en este caso viene desde **node_modules**, y lo mismo podría hacer si necesitase scripts tipo **jQuery**, cosa que casi nunca vamos a utilizar, pero si fuese el caso de esos scripts sueltos de **JavaScript**, se meterían en esta carpeta de scripts. El sistema lo que hace con los scripts y los styles es incorporarlos tal cual. Es lo que se conoce como **asset**, algo que tienes y que se usa tal cual.

Por supuesto, también tenemos aquí un fichero de estilos genérico. Para no aburriros, os copio ya lo que voy a utilizar yo durante todo este tiempo en la **aplicación**. Y una cosa interesante es que estos cambios no se ven reflejados inmediatamente, sino que tengo que relanzar la **aplicación**. ¿Por qué? Porque no se consideran cambios en el **source**, en la carpeta **src**, sino que forman parte de lo que hay alrededor.

Ahora sí, ya tengo esto con esos estilos aplicados. Bien, podemos mejorar esto un poquito. Y le agregaremos más cosas, por ejemplo, un botón donde confirmaremos la reserva cuando la hagamos. En el **main**, un poquito más de información como, por ejemplo, cuántos participantes ya hay actualmente en esta **actividad**.

Solo para que tengamos aquí un poquito estas tres secciones. El **header**, el **main** y el **footer** de este artículo. Pero una cosa que os quería mostrar, llegados hasta aquí, es que podemos tener estilos específicos para cada componente. Esto es para lo que vale la propiedad **styles**. En este caso, yo me he creado aquí unas propiedades que utilizaré según el estado en el que esté esta **actividad**. Esta **actividad** ahora mismo está en estado **published** y su estilo será este color más o menos verdoso.

¿Cómo aplico eso? Muy fácil. Hay que pensar que tal como **href** fue un atributo al que le hemos puesto un valor, pues **class** también puede ser lo mismo. Solo tengo que incluir el atributo entre corchetes y poner aquí el valor que va a ocurrir, ahí lo tenéis, que la **actividad** se haya puesto con el estilo adecuado. Si la **actividad** estuviese, por ejemplo, en **draft**, pues el estilo cambiaría.

Con lo cual, hemos aprendido que podemos incorporar nuevos elementos a **Angular**, tal como hemos hecho con la instalación de un **framework CSS**. Hemos aprendido que se puede incorporar contenido sin compilar, simplemente contenido tal cual, como podrían ser estilos y scripts. Hemos aprendido que tenemos un **styles CSS** donde cambiar estilos para toda la **aplicación**, o que podemos también tener estilos locales para cada componente y que no se heredan, por lo cual hace que esto sea muy fácil de reutilizar. Por otra parte, los estilos, como cualquier otro dato, se puede macrosustituir mediante el corchete, por ejemplo, para el atributo **class**.