# Tutorial para pruebas unitarias con Karma y Jasmine.

Contenido

[Introducción. 2](#_Toc16161820)

[Versiones del proyecto. 2](#_Toc16161821)

[1. ¿Qué es Jasmine? 2](#_Toc16161822)

[2. ¿Qué es karma? 2](#_Toc16161823)

[3. ¿Cómo instalamos y utilizamos estos 2 Frameworks? 2](#_Toc16161824)

[3.1. ¿Como creamos una prueba unitaria? 3](#_Toc16161825)

[4. Estructura básica de un componente de pruebas utilizando Jasmine. 3](#_Toc16161826)

[5. Lanzando las pruebas y entendiendo el menú. 5](#_Toc16161827)

[6. Ciclos de vida de las pruebas unitarias. 7](#_Toc16161828)

[6.1. BeforeEach. 8](#_Toc16161829)

[6.2. AfterEach. 8](#_Toc16161830)

[6.3. BeforeAll. 8](#_Toc16161831)

[6.4. AfterAll. 8](#_Toc16161832)

[6.5. Resumen. 9](#_Toc16161833)

[7. Creando nuestras primeras pruebas. 9](#_Toc16161834)

[8. Manuales, documentación y otros conceptos. 11](#_Toc16161835)

[9. Creación de pruebas unitarias en servicios con conexiones HTTP. 11](#_Toc16161836)

[9.1. Resumen del servicio creado y como se implementa. 12](#_Toc16161837)

[9.2. Importaciones de librerías necesarias. 13](#_Toc16161838)

[9.3. Creación de variables y asignaciones. 14](#_Toc16161839)

[9.4. Integridad después de cada prueba. 14](#_Toc16161840)

[9.5. Haciendo nuestras pruebas unitarias. 14](#_Toc16161841)

[10. Espías en nuestro código. 17](#_Toc16161842)

[11. Omisión, exclusión y forzar foco de pruebas. 18](#_Toc16161843)

[12. Cobertura de nuestras pruebas unitarias. 21](#_Toc16161844)

[13. Enlaces de interés. 23](#_Toc16161845)

## Introducción.

Cuando desarrollamos una aplicación, es necesario generar test sobre los distintos elementos de nuestra App para comprobar que cada uno de los elementos mínimos de código funcionan de manera correcta.

Hay que diferenciar entre prueba funcional y prueba unitaria. La prueba funcional nos permite comprobar si se cumple una especificación, un requisito o una funcionalidad de la aplicación, aunque está utilice diferentes componentes, servicios y capas (Ej: para hacer el log necesitaremos un servicio, un componente y algunos métodos).

La prueba unitaria se utiliza para probar una unidad mínima funcional de código (Ej: comprobar si un método nos devuelve un número o una cadena de caracteres, si existe una etiqueta o si ante un evento se genera código específico en el DOM). Es muy importante ser plenamente consciente de que cada prueba debe quedar totalmente encapsulada y no requerir ningún elemento externo a la propia prueba, es decir, si un método tiene valores de entrada, estos deberán ser proporcionados por la persona que realiza la prueba y no enviárselos desde otro método/servicio.

### Versiones del proyecto.

Angular 8.0.1

Angular CLI 8.0.3

Karma 4.1.0

Jasmine 3.0.4

## ¿Qué es Jasmine?

Jasmine es un Framework para pruebas unitarias basado en JavaScript el cual nos permite hacer pruebas, si no también obtener los resultados de estas de una manera visual y con lenguaje humano.

## ¿Qué es karma?

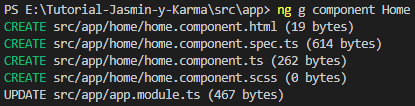
Karma es lo que se denomina un “test runner”, es decir, nos permite automatizar las pruebas creadas por otros Frameworks (como Jasmine).

## ¿Cómo instalamos y utilizamos estos 2 Frameworks?

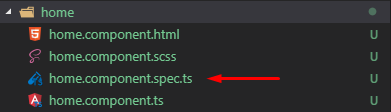
Ambos entornos vienen instalados en la creación básica de un proyecto Angular, por lo que no vamos a necesitar hacer ninguna instalación o configuración inicial.

### ¿Como creamos una prueba unitaria?

Si estamos utilizando Angular CLI para la creación de componentes (Si no se está haciendo así, empieza a hacerlo), cuando generas un componente nuevo se creara automáticamente un fichero con el nombre del documento pero con la especificación “**.spec.ts**”.



\*1



\*2

En este fichero realizaremos todas las pruebas unitarias sobre el componente homónimo.

En caso de querer hacer pruebas unitarias sobre un servicio (no lo generamos como un componente, si no como un “service” será necesario generar el fichero de manera manual, generando el código básico de un componente de pruebas que se explicará a continuación.

## Estructura básica de un componente de pruebas utilizando Jasmine.

Cuando generamos el componente mediante el CLI de Angular, este ya viene construido con una estructura y un par de comprobaciones básicas cuya estructura es la siguiente.

\*3

1. Zona de importaciones: como se ha mencionado antes, todas las pruebas deben de ser autónomas e independientes, por lo que seguramente tengamos que añadir directivas como las de “Angular Material”, el componente o librerías propias de Jasmine. Para hacer la importación es exactamente igual que en el resto de los componentes Angular.

Aquí tenemos que destacar (por que probablemente no hayas trabajado nunca con ellas) tas librerías “TestBed” y “ComponentFixture”.

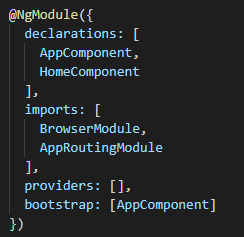
- TestBed: Esta librería nos va a permitir generar una clase “@NgModule” para este componente de manera automática. Se utilizará en la siguiente sección de este punto.

- ComponentFixture: Está librería nos va a permitir crear una instancia del componente sobre el que vamos a realizar todas nuestras pruebas.

1. Definición de variables: En esta sección podemos definir variables que utilizaremos a lo largo de nuestras pruebas unitarias.
2. Módulo principal: También generaremos el módulo de la aplicación gracias a la librería TestBed.

Como podemos ver, sigue la misma estructura que los módulos utilizados en Angular, por lo que tendremos las mismas secciones en caso de ser necesarias.

\*Ejemplo del modulo principal de la aplicación.

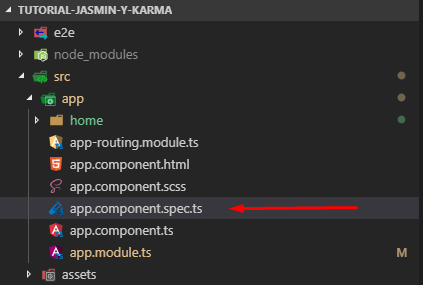


\*4

1. Comprobación de creación de componente: este método nos va a permitir crear una instancia del componente sobre el que vamos a hacer todas las pruebas. Permite crear un método que detecte cambios es nuestro componente para poder actualizar la batería de pruebas de manera constante.
2. Pruebas unitarias: aquí ya podemos comenzar a generar nuestras pruebas funcionales. Tema que trataremos en este manual más adelante.

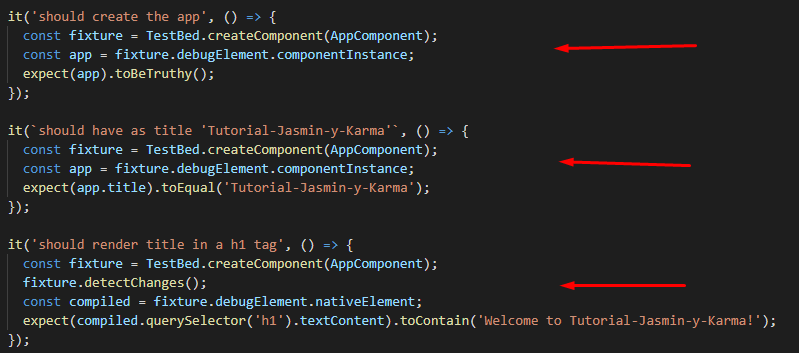
## Lanzando las pruebas y entendiendo el menú.

Ahora que ya tenemos conocimientos generales sobre que son las pruebas unitarias y como generar sus componentes, vamos a comenzar a utilizarlas. Para empezar, vamos a lanzar las pruebas unitarias básicas creadas por Angular para el componente “App” del proyecto de ejemplo.



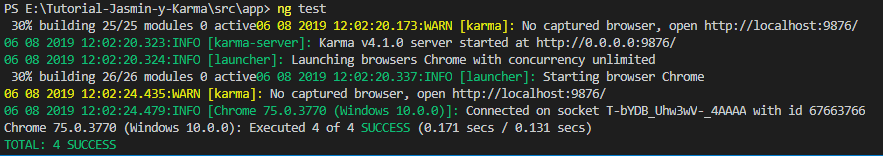
\*5

Como se puede ver, en este fichero encontramos 3 pruebas unitarias distintas, las cuales veremos en profundidad mas adelante, ahora solo nos interesa como ejecutar estas pruebas y ver qué información nos devuelven.



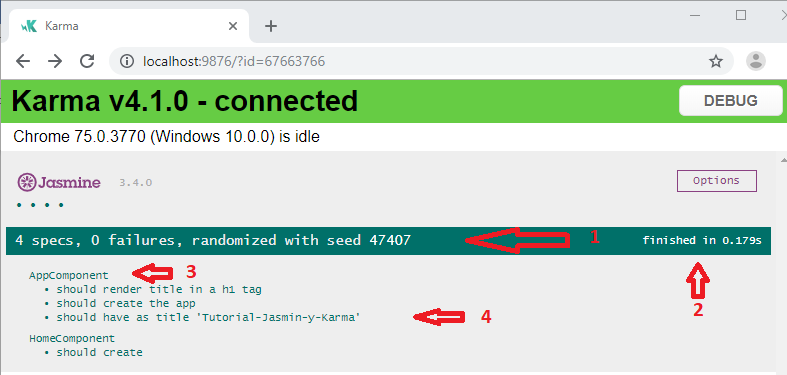
\*6

Para lanzar las pruebas debemos ejecutar el comando “**ng test**” desde la terminal, de manera similar a cuando hacemos el “ng serve” para lanzar nuestro proyecto.



\*7

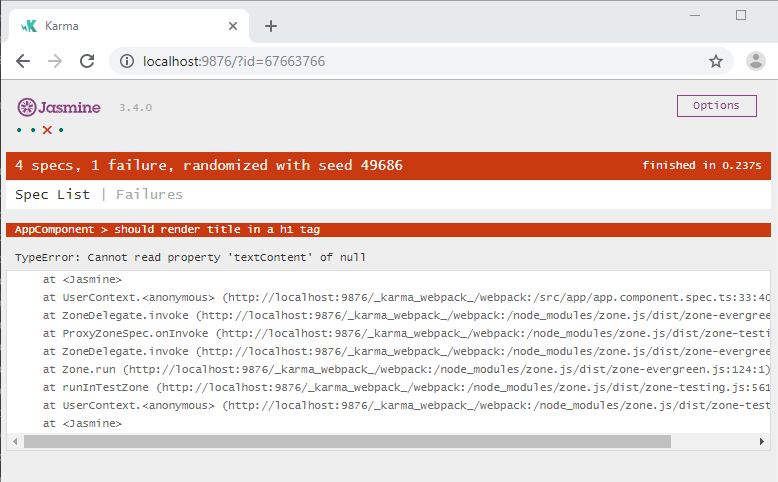
Cuando el servidor en localhost se haya lanzado completamente nos aparecerá una ventana en nuestro navegador similar a esta si es correcta:



\*8

1. Total de pruebas unitarias realizadas: hay 4 pruebas unitarias que hemos realizado. En las líneas de abajo podemos ir viendo cuales se han realizado y a que componente pertenecían. También se especifica el numero origen o semilla que se ha utilizado para realizar estas pruebas de manera aleatoria.
2. Tiempo que han tardado en realizarse: en cuantas mas pruebas se incluyan, mas tiempo se tardará en realizar, esto te permite conocer cuanto es exactamente (para que, si de media son 5 minutos, no te quedes mirando a la pantalla).
3. Componentes: esta es la lista de componentes sobre los que se han realizado todas las pruebas unitarias.
4. Acciones: acciones que debería haber realizado la prueba y si se ha hecho de manera correcta o no.

Ahora vamos a ver como es la salida de una prueba unitaria errónea:



\*9

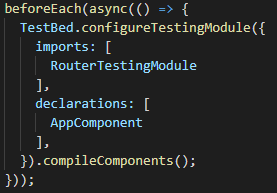
Como se puede ver, la estructura es muy similar, con la diferencia que ahora se especifica la prueba que no ha sido superada correctamente y el componente al que pertenece dicha prueba, además del código de error correspondiente a la prueba que nos permitirá ubicar más fácilmente el fallo.

## Ciclos de vida de las pruebas unitarias.

Cuando hablamos de pruebas unitarias y vemos un poco de sucódigo, es fácil creer que su funcionamiento, ejecución y ciclo de vida es como el de un componente de Angular normal, pero esto no es así, ya que el ciclo de vida de un componente de pruebas es fragmentado. Cada método o prueba se ejecuta de manera independiente, pudiendo especificar métodos que harán tareas antes o después de cada prueba, al inicio de todas las pruebas, al final de todas…

### 6.1. BeforeEach.

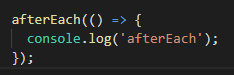
Este ciclo de vida nos lo encontramos en todos los ficheros de pruebas unitarias cuando son creados por Angular CLI. Esto nos va a permitir ejecutar todo el código que esté dentro de este método antes de cata una de las pruebas que hagamos (los métodos que empiezan con “It(‘prueba’)).



\*9.1

### 6.2. AfterEach.

Este método va a ejecutarse después de cada uno de los métodos “It”



\*9.2

### 6.3. BeforeAll.

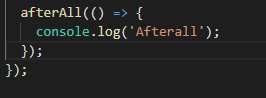
Este método se va a ejecutar al inicio de la ejecución del componente, es decir, se ejecuta solo 1 vez justo después de la instanciación del componente de pruebas. Hay que destacar que no es necesario escribirlo justo debajo, ya que independientemente de donde se declare, este va a ejecutarse siempre el primero.



\*9.3

### 6.4. AfterAll.

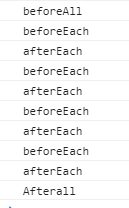
Este método va a ejecutarse siempre al final de la ejecución e todas las pruebas, es decir, será el último método que se ejecute antes de finalizar las pruebas del compoente.



\*9.4

### 6.5. Resumen.

En la foto que hay a continuación se puede ver mas claramente como se van ejecutando los distintos ciclos de vida de las pruebas unitarias.



\*9.5

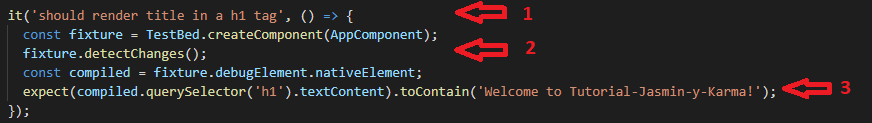
Como se puede ver el primero es beforeAll, seguido de tantos beforeEach/afterEach como pruebas unitarias tenga la aplicación y para finalizar un afterAll.

## Creando nuestras primeras pruebas.

Ahora que ya sabemos para que valen, que estructura tienen sus componentes, como se ejecutan y como utilizar sus resultados, vamos a crear nuestras pruebas unitarias personalizadas.

Lo primero es distinguir entre pruebas unitarias del componente(typescript) y de la parte HTML(DOM), ya que, aunque su formato, estructura y métodos son iguales, existen diferencias a la hora de instanciar las variables que vamos a utilizar.

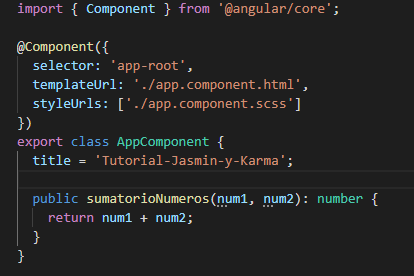
Para ello vamos a ver su estructura, en este caso, la prueba unitaria comprueba si la página contiene una etiqueta “**<h1>**” y si el contenido de esta coincide con el que se muestra en la foto.



\*10

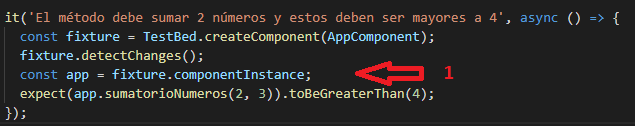
1. Método contenedor de la prueba y descripción de la prueba unitaria: aquí vamos a especificar mediante una cadena de texto, cual es el objetivo de la prueba, por lo que debería ser lo mas específico posible. Este texto es el que se muestra en la pantalla de resultados de las pruebas (Foto número 8, punto 4).
2. Generación del componente y subscripción a cambios: aquí generamos la instancia al componente y nos subscribimos a sus cambios. Esto ya se ha explicado en la foto 3, punto 4.
3. Prueba unitaria: aquí es donde realmente hacemos la prueba unitaria.
   1. Lo primero que hacemos es usar el comando “expect” para decir al método que es lo que debería hacer esta prueba.
   2. Utilizamos la variable “compiled” creada anteriormente (a la que hemos asignado el valor de “fixture.debugElement.nativeElement” porque vamos a trabajar con el DOM) para empezar a poner nuestras exigencias, en este caso queremos que posea una etiqueta “**<h1>**”.
   3. También queremos exigir que esta etiqueta dentro de nuestro código posea una cadena de caracteres concreta que especificamos dentro de la función “**toContain ()**”.

Ahora vamos a hacer lo mismo, pero con una prueba que se encargue de trabajar con la parte del componente(typescript) que sume 2 números y devuelva un número mayor que 4. Para esto, lo primero que tenemos que hacer es crear este método en el componente (en nuestro caso, lo estamos creando en “app.component.ts”.



\*11

Una vez ya hemos creado el método que queremos depurar con nuestras pruebas unitarias, tenemos que crear nuestra prueba. Para ello vamos a nuestro componente de pruebas (app.component.spec.ts) y creamos el método que hará la prueba.



\*12

Como podemos ver, ahora ya no utilizamos el contenido de la variable “fixture.debugElement.nativeElement”, si no que ahora estamos utilizando “fixture.componentInstance”, es decir, la instancia del componente sobre el que hacemos la pruebas.

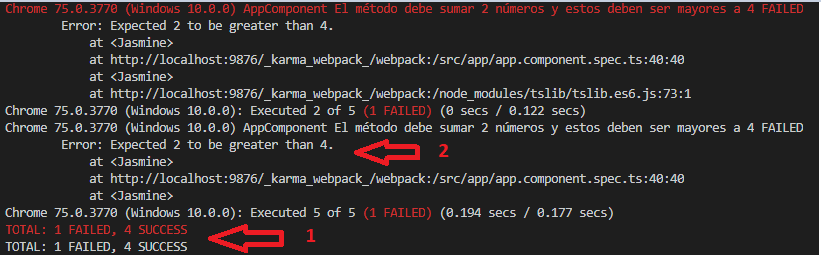
En este caso, desde el método “expect()” accedemos al método que hemos creado anteriormente en el componente (app.component.ts) y le pasamos los valores que nosotros queramos. Después le especificamos que queremos que el resultado de ese método debe ser superior a 4.

Si hemos hecho todo correctamente, al guardar los cambios, la terminal debería informarnos de que se ha añadido una prueba unitaria más y que esta se está realizando correctamente.



\*13

Podemos comprobar que la prueba cambiando el número que le pasamos como parámetro a 0, de tal manera que no se cumpla la condición.



\*14

Como podemos ver en el punto 1, la terminal nos informa que se han realizado 5 pruebas unitarias, pero que 1 de ellas ha fallado. En el punto 2 vemos como nos especifica el error, diciendo que nos ha devuelto un 2, el cual debería ser mayor a 4.

## Manuales, documentación y otros conceptos.

Es muy probable que después de haber visto esto, entiendas como funciona una prueba unitaria, pero no sepas que puedes llegar a comprobar con ellas, por lo que, para no extenderme mucho y hacer un manual de cientos de páginas, te dejo las referencias a la API de Jasmine, en la que te vienen explicados todos los métodos que posee este gran Framework de pruebas.

Es recomendable que antes de seguir con este manual, le des un vistazo a estos enlaces para que tu aprendizaje sobre la utilización mas “avanzada” de Jasmine sea correcta.

1. <https://jasmine.github.io/2.0/introduction.html>
2. <https://jasmine.github.io/tutorials/custom_argument_matchers>
3. <https://jasmine.github.io/tutorials/async>

## Creación de pruebas unitarias en servicios con conexiones HTTP.

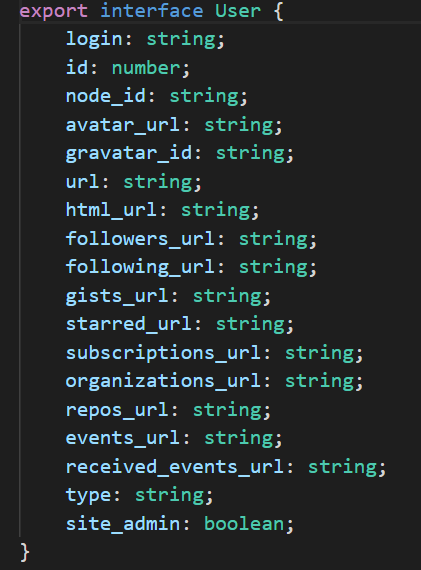
Ahora que ya sabemos para que valen, que estructura tienen y como se ejecutan, vamos a ir un paso más adelante y vamos a comenzar a ver cómo hacer pruebas unitarias sobre servicios que atacan a una API externa mediante el protocolo Http.

Llegados a este punto es necesario destacar que se considera que el lector tiene conocimientos sobre servicios, implementación de librerías y creación de servicios. Si no se posee alguno de estos conocimientos es mejor que primero aprendas esos conceptos antes de continuar con este punto.

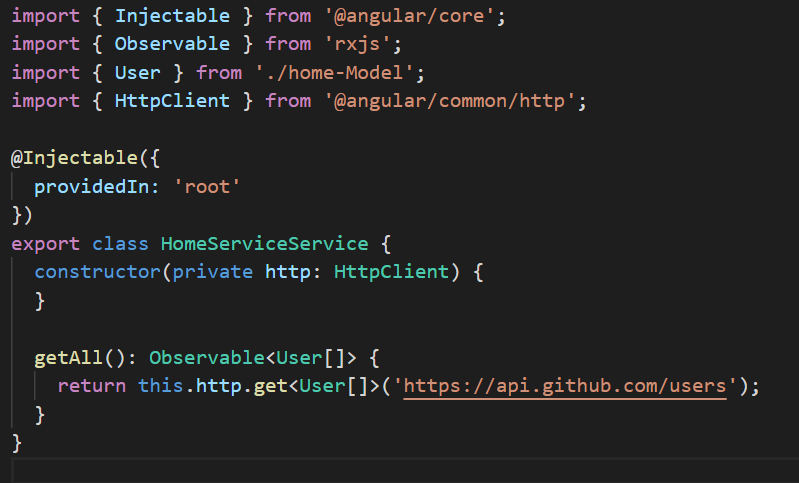
Por último, es necesario aclarar, que las pruebas unitarias de un servicio jamás, y debo hacer mucho hincapié en esta palabra “**JAMÁS**” se harán sobre la BBDD real, ya que en una aplicación pequeña no suponen mucho problema, pero si se hacen en una gran aplicación, tantas llamadas simultaneas pueden perjudicar el rendimiento del servidor

### Resumen del servicio creado y como se implementa.

Aquí os voy a dejar un pequeño resumen del servicio que hemos creado, a donde ataca y que modelo usa.



\*15. Interfaz

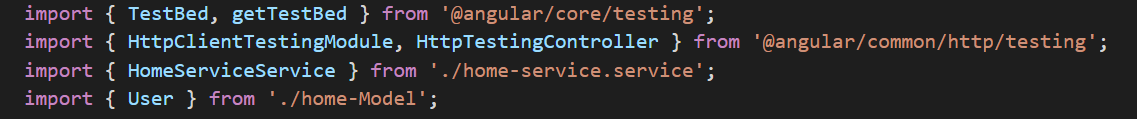


\*16. Servicio

En cualquier caso, Esto se encuentra en el proyecto de ejemplo que acompaña a este tutorial dentro de la carpeta “home/Servicio”.

Ahora vamos a comenzar con el fichero para las pruebas unitarias que genera de manera automática el CLI de Angular.

### Importaciones de librerías necesarias.



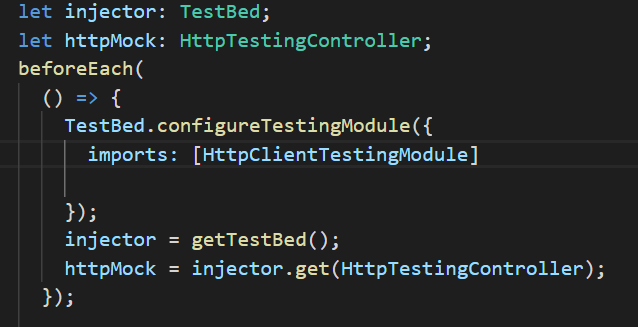
\*17

La primera importación que hacemos es “TestBed” que ya la hemos visto anteriormente y “getTestBed”, la cual nos va a permitir crear la instancia del módulo principal.

La segunda importación son las librerías “HttpClientTestingModule”, la cual, nos va a permitir simular las llamadas Http y “HttpTestingController” que es con la que haremos el mock de los datos para simular las llamadas a la BBDD.

Las 2 últimas importaciones son el servicio y la interfaz que vamos a utilizar para el ejemplo.

### Creación de variables y asignaciones.



\*18

A nivel del componente de pruebas, debemos declarar 2 variables que nos ayudarán a trabajar con las pruebas.

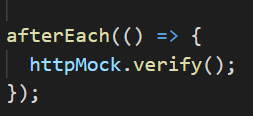
1. Injector: obtendrá las configuraciones del módulo para poder utilizar las librerías importadas.
2. httpMock objeto que nos va a permitir simular nuestra llamada a la BBDD.

¡Importante!: recuerda que el módulo de test unitarios no utiliza en “app.module” que utiliza el componente real, por lo que debemos importar la clase “HttpClientTestingModule” en la configuración del módulo.

### Integridad después de cada prueba.

Como las llamadas a un servicio son siempre asíncronas, ya que no podemos saber cuánto tiempo vamos a tardar en obtener la respuesta, es necesario comprobar después de cada llamada, que esta ha finalizado. El cómo ha terminado lo validaremos en cada una de nuestras pruebas, pero después de cada una, es necesario comprobar que de verdad han terminado y su conexión ha sido completamente cerrada.

Para esto vamos a utilizar la función “**verify()**” de la clase “HttpTestingController” la cual hemos almacenado en “**httpMock”.**

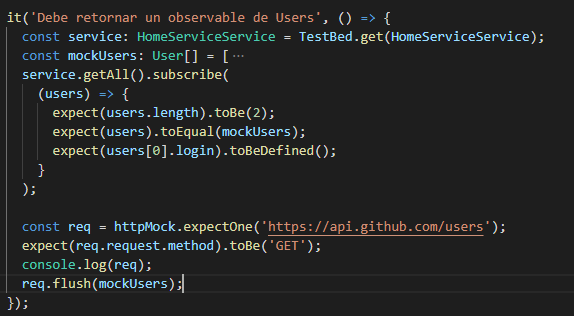


\*19

### 9.5. Haciendo nuestras pruebas unitarias.

Ahora vamos a comenzar a hacer nuestras pruebas. Ya tenemos todo configurado y listo para empezar a simular nuestras conexiones.

Aunque a lo largo de este punto iremos explicando para que vale cada uno, aquí dejo una foto para que te hagas una idea general de cómo va a quedar el método.



\*20

Lo primero que debemos hacer es crear un modelo de datos que coincida con la interfaz que vamos a utilizar en la llamada real con datos reales. Para ello utilizaremos un array de del tipo “User” con datos rellenados.



\*21

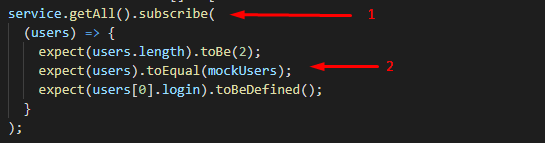
Esta declaración se debe hacer dentro del método, ya que a pesar de que se pueda hacer fuera, y su funcionamiento sea igual, es recomendable hacer que cada una de las pruebas unitarias sea totalmente independiente al resto.

Ahora vamos a instanciar el servicio que emulará nuestra petición. Esto lo haremos a través de la librería TestBed con su método “GET”, el cual recibe la instancia del servicio que vamos a emular.



\*22

Ahora que ya tenemos nuestro modelo de datos y el servicio emulado, podemos comenzar a hacer las “llamadas” al mismo y empezar con la prueba en sí.



\*23

1- Usando la variable que tiene la instancia de nuestro servicio, hacemos una llamada al método sobre el que queramos hacer nuestra prueba unitaria de la misma manera que lo haríamos en nuestro componente, es decir, llamamos al método, nos subscribimos para esperar la respuesta y trabajamos con los datos devueltos.

2- En esta sección, debemos hacer todas las comprobaciones que consideremos necesarias, eso quiere decir que podemos usar todos los métodos que hemos aprendido en el punto 7 sobre la variable “users”. Hay que recordar que esto es una prueba unitaria, por lo que lo normal es comprobar si lo que nos devuelve tiene el formato correcto, los datos tienen los tipos que deben …

Una vez hemos realizado todas las pruebas que consideremos, hay que realizar una última (no es obligatorio, pero si es altamente recomendable, sobre todo si vamos a trabajar con distintos entornos). Esta prueba consiste en comprobar si la URL a la que estamos accediendo es la correcta.



\*24

Gracias a estas 2 pruebas, podemos comprobar si el método que estamos realizando es un “GET” y si se hace a la dirección correcta. Esto nos va a evitar muchos problemas en el caso de que se nos haya olvidado cambiar el entorno antes de proteger nuestra rama y evitar subir al entorno de pruebas/ desarrollo/producción llamadas de otros entornos.

Por último, es necesario hacer que todos los datos que hemos “mockeado” puedan ser enviados al servicio, por lo que para hacer esto utilizaremos la siguiente sentencia.



\*25

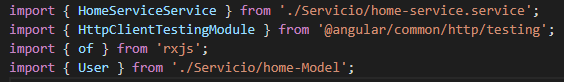
Ahora mismo, ya puedes comenzar a realizar tus pruebas unitarias sobre un servicio Http, sin embargo, con esto solo podemos comprobar que el servicio funciona como nosotros queremos, pero no que este servicio es correctamente llamado desde un componente. Para esto tenemos a los “espías”, los cuales veremos con detalle en el siguiente punto.

## Espías en nuestro código.

Son fragmentos de códigos que nos permiten verificar que se está llamando correctamente a un método y que la conexión entre estos se realiza de manera correcta.

Para comenzar, es necesario aclarar que esto ya no se va a realizar en el servicio, si no que se hará en el componente que llama a ese servicio, que en este caso se hará desde “home.component.ts”.

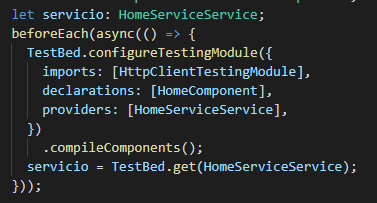
Lo primero que vamos a tener que hacer, como siempre, es hacer todas las importaciones necesarias para poder operar. En este caso será el cliente Http de pruebas (“HttpClientTestingModule”) la interfaz que usamos de modelo (“User”), nuestro servicio (“HomseServiceService”) y la librería of de rxjs(“Of”).



\*26

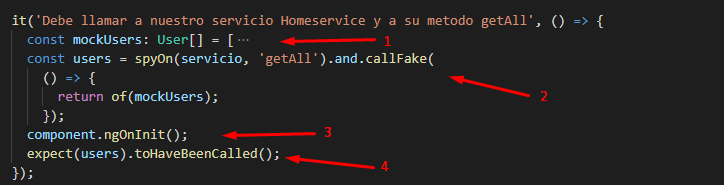
El resto de las librerías ya han sido explicadas (como por ejemplo TestBed) por lo que no vamos a hacer hincapié en su uso en este punto.

Nuestro siguiente paso es crear el servicio como una variable y asignarle un valor para poder trabajar con él.



\*27

Ahora ya podemos comenzar a realizar las pruebas que nosotros queramos.



\*28

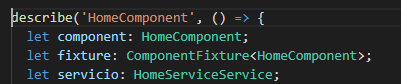
1. Vamos a necesitar tener un modelo de datos, en este caso vamos a utilizar el mismo que en el punto anterior (Imagen 21).
2. Creamos una variable en la que vamos a almacenar la respuesta que nos de nuestro emulador de conexiones Http. Para esto utilizaremos el método “**spyOn()**”, el cual, recibirá como parámetros la variable en la que tenemos almacenado nuestro servicio y el método al que vamos a llamar como un string.
   1. Una vez tenemos esto llamamos al método “**and.callFake()**”, el cual, mediante una función de flecha, nos devolverá el array de usuarios que hemos creado anteriormente.
3. Llamamos al método que queremos probar, en este caso “**ngOnInit()**”, ya que es aquí donde llamamos al servicio.
4. Ahora debemos hacer la comprobación real, en la cual decimos que la variable users(en la que hemos almacenado los datos que debería devolver el servicio) debe ser llamada.

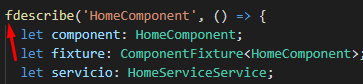
Con esto, ya tendríamos la comprobación de si un servicio se está llamando. Es normal que vuelva a asaltarte la misma duda que en el punto anterior, ¿pero los datos no son reales, esto es correcto hacerlo así? La respuesta es la misma que en el punto anterior, si, por que, si en vez de simular datos hacemos llamadas al servidor, las pruebas unitarias tendrían un impacto terrible en nuestro servidor, además de poder llegar a tardar mucho en realizar todas las pruebas.

## Omisión, exclusión y forzar foco de pruebas.

Según la aplicación desarrollada crezca, aumentaremos el número de pruebas unitarias y su tiempo de ejecución comenzará a ser mayor. Si la aplicación es muy grande, la cobertura de pruebas unitarias es muy alta o los métodos son muy pesados y costosos, hacer un cambio y ejecutar todas las pruebas de la aplicación puede llegar a suponer un gran problema, además, no tiene mucho sentido que, si modificas solo un método, tengas que volver a hacer pruebas unitarias del resto de métodos, los cuales no has tocado y sabes que cuando se desarrollaron ya pasaron las pruebas unitarias. Para esto Jasmine nos da la oportunidad de hacer foco en un componente o en un solo método, ahorrándonos una enorme cantidad de tiempo.

Para hacer foco sobre un conjunto de elementos, utilizaremos la letra f antes de la declaración del método o del componente.





\*28

Si ejecutamos las pruebas con la f en la descripción del componente podremos ver algo parecido a esto.

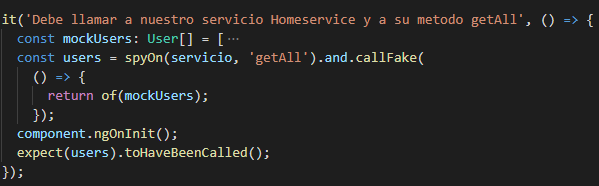


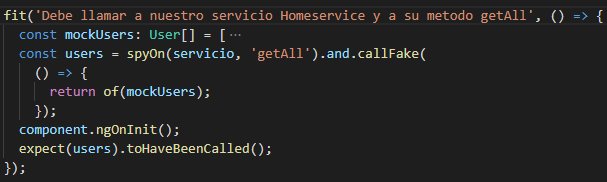
\*29

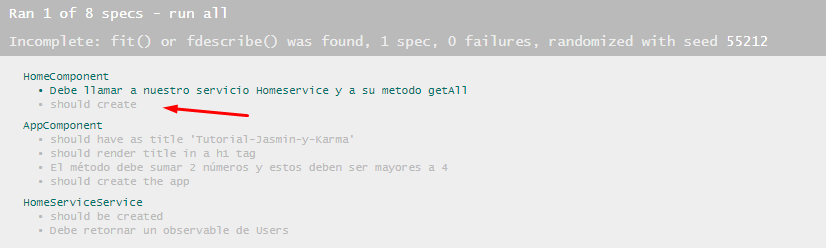
Como se puede ver, solo aparecen en verde, es decir, que se han ejecutado, los métodos del “HomeComponent” mientras que el resto, han sido ignorados.

Además, en la parte superior vemos como nos dicen que solo se han ejecutado 2 pruebas de las 8 que hay programadas en todo nuestro proyecto.

Como se ha dicho antes, esto también puede aplicarse a uno o varios métodos dentro de uno o varios componentes.



\*30



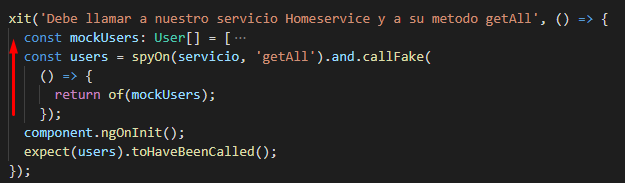
\*31

También pueden hacerse en métodos de distintos componentes.

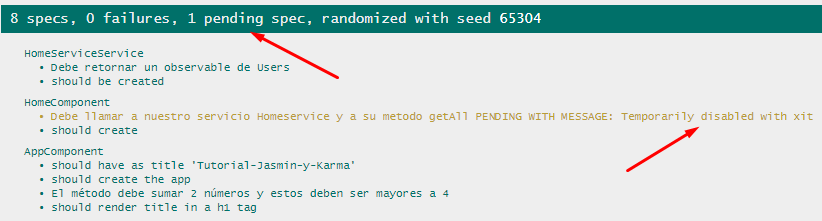


\*32

Esto es muy útil, pero… ¿qué pasaría si queremos lo contrario?, es decir, tengo 50 pruebas, pero quiero ejecutar 45 de ellas, e ignorar otras 5. Seria muy absurdo tener que dedicar tiempo a poner la letra f delante de cada uno de los métodos que quieres, por lo que para eso tenemos la letra x, cuyo funcionamiento es exactamente igual y se aplica en los mismos casos y de la misma manera que “f” pero en este caso nos vale para excluir.



\*33



\*34

Como podemos ver, ahora nos dice que se han realizado todas las pruebas a excepción de una, que esta en espera gracias al comando “xit”.

## Cobertura de nuestras pruebas unitarias.

Llegados a este punto, ya sabes lo que son las pruebas unitarias, como se estructuran, crean y se visualizan, un montón de métodos para garantizar el correcto funcionamiento de tu app… incluso eres capaz de simular servicios y llamadas Http para hacer tus pruebas sin que el encargado del Backend te intente agredir por saturar su servidor con tus pruebas unitarias, pero… ¿Cómo sabemos si nuestras pruebas son suficientes, si nos hemos dejado posibles casos sin cubrir?.

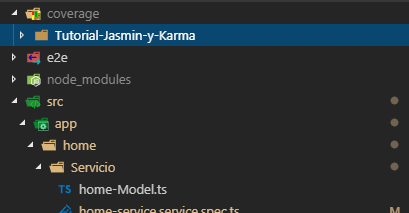
Jasmine nos brinda una herramienta para generar informes sobre pruebas que nos van a permitir conocer exactamente que casos hemos cubierto y que porcentaje supone respecto al total de posibilidades de una manera rápida, sencilla y, sobre todo, visual.

Para esto vamos a utilizar el comando “**ng test --code-coverage**” en nuestra terminal.



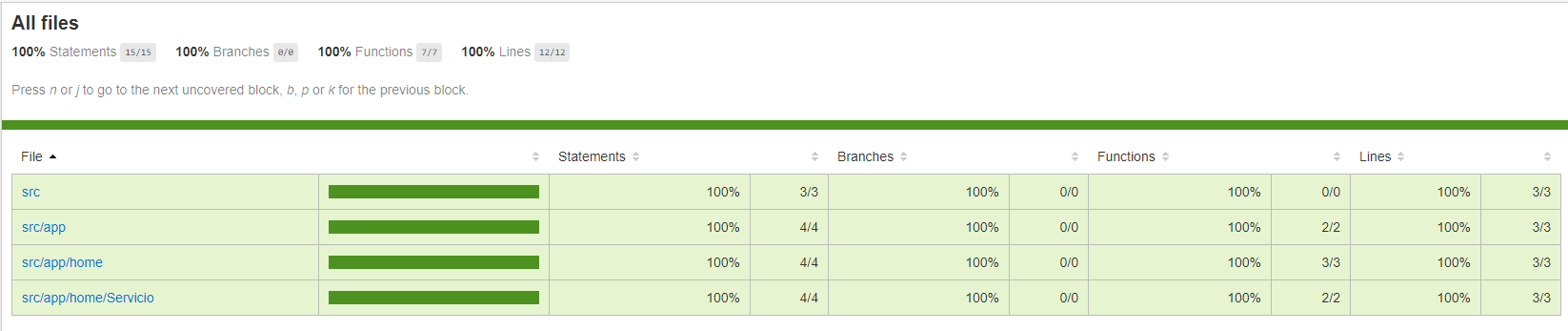
\*35

Este comando nos generara una carpeta nueva en nuestra estructura del proyecto con nombre “coverage”.



\*36

Dentro de esta carpeta encontraremos un documento “index.html”, el cual, podremos abrir para ver la cobertura de todos nuestros casos de usos.

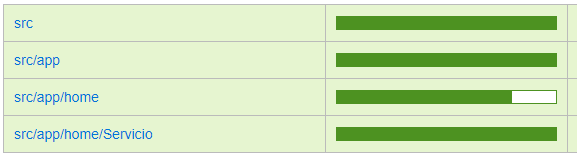


\*37

Como las pruebas que hemos realizado son muy sencillas, tenemos una cobertura del 100%, pero es necesario recordar que, en un proyecto real cuyos componentes tienen muchos métodos con muchas entradas y muchas salidas, es imposible (o increíblemente difícil) alcanzar el 100% de cobertura.

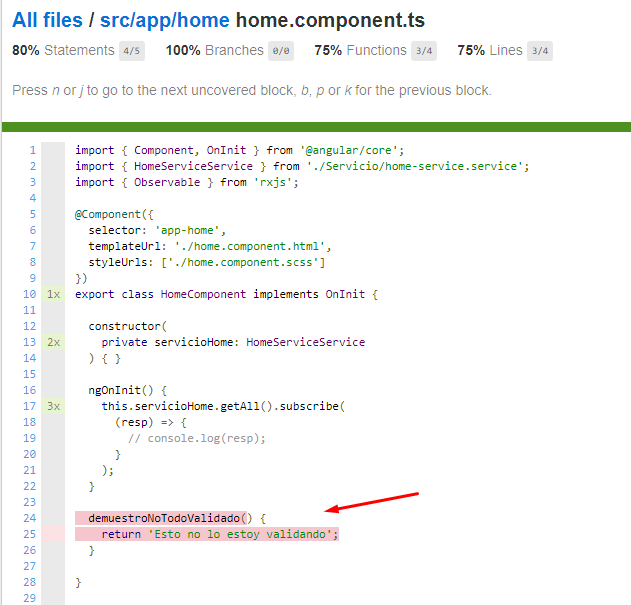
Ahora hemos añadido un método nuevo en el componente “HomeComponent” el cual no hemos validado, por lo que nuestro index cambiará y nos aportará información

sobre que estamos haciendo mal o mejor dicho, que no estamos comprobando.



\*38

Si entramos a este componente podremos ver de manera mas detallada que métodos, variables, funciones… no están siendo probadas, lo que nos va a permitir hacer esas pruebas y volver a ejecutar el comando para generar de nuevo el documento.



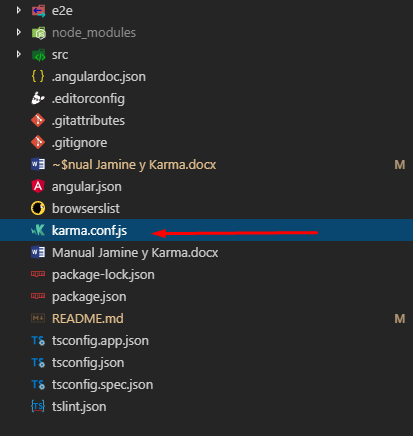
\*39

## Karma y automatización de pruebas.

Hasta ahora, nos hemos concentrado en Jasmine (estructura, metodología, funcionalidades…) per no nos hemos parado a pensar como es que con un simple comando “ng test” todo esto funciona de manera automática. Aquí es donde entra karma.

Karma nos permite generar un entorno para realizar pruebas de manera automatizada y facilitarnos la integración continua de nuestra aplicación, la cual, veremos más adelante.

Karma surgió originalmente en el equipo de desarrollo de Angular, y aunque más adelante creció y se extendió a otros proyectos, sigue teniendo una buena integración con Angular, por lo que el CLI de Angular nos configura el entorno para lanzar pruebas unitarias con Karma de manera automática. Si deseamos ver estas configuraciones tendremos que ir al fichero “karma.conf.js”, el cual se encuentra en la raíz del proyecto.



\*40

En este fichero podemos modificar todas las configuraciones de Karma, pero demomento, orientándonos a la ejecución de la integración continua, solo vamos a cambiar la variable “**singleRun**” a true. Esto va a hacer que cuando ejecutemos el comando “**ng test --code-coverage**” el servidor no se lance. Esto es necesario ya que, con la integración continua, será el servidor el que se encargue de realizar estos trabajos, y no nuestro ordenador.

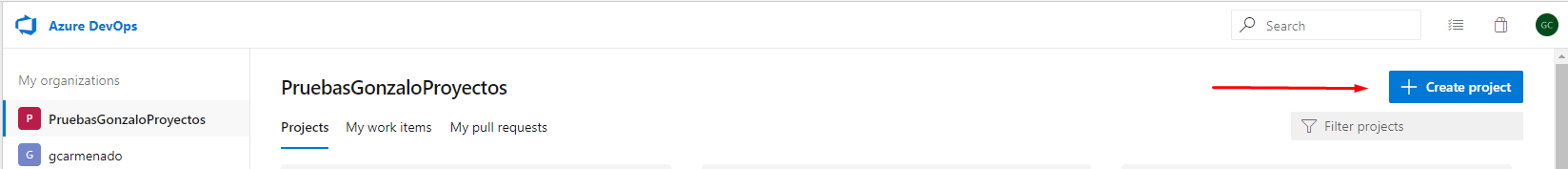
## Azure DevOps, manejo, creación y configuración.

### Creación de cuenta y repositorio.

Azure DevOps es una plataforma de Microsoft que nos va a permitir, entre otras muchas funciones, mantener un proyecto basado en la integración continua.

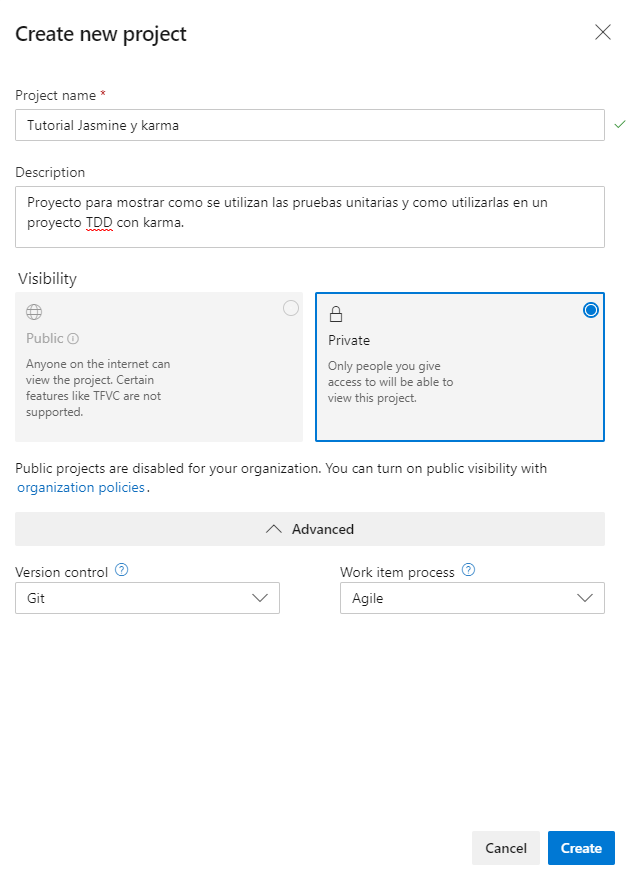
Para ello accedemos a la página de Azure DevOps y nos creamos/hacemos login en una cuenta.

Después vamos a crear un nuevo proyecto.



\*41

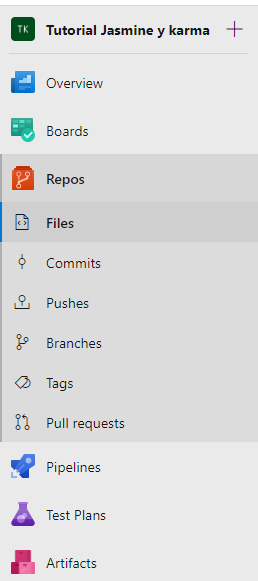
Rellenamos los datos y creamos nuestro nuevo proyecto.



\*42

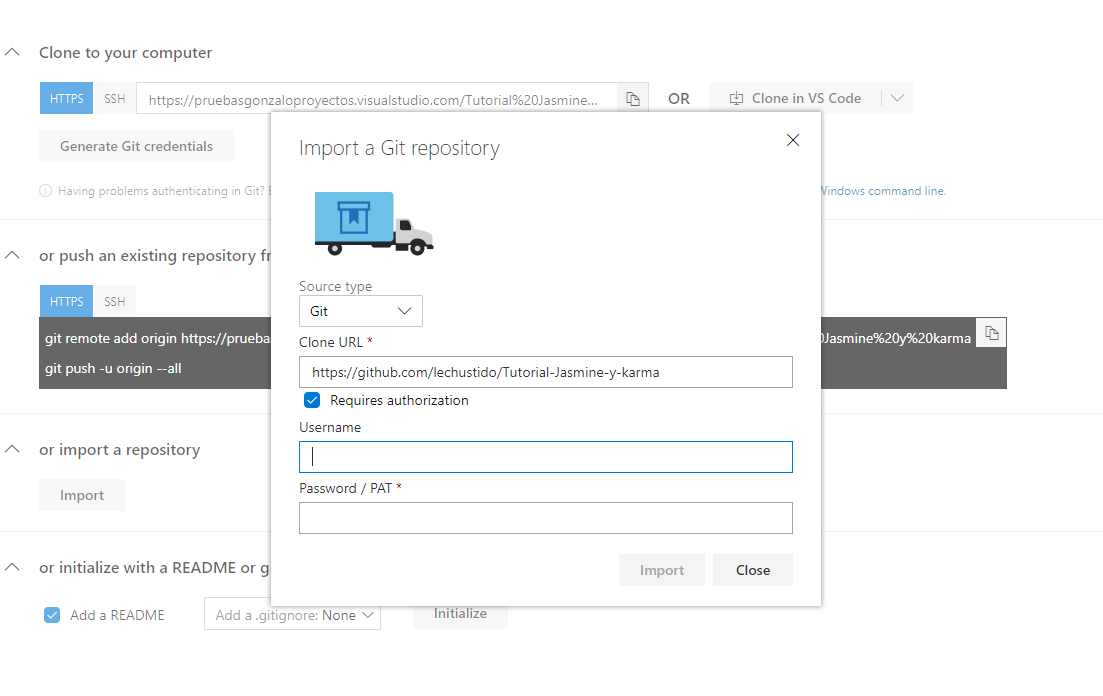
### Migración de proyecto a Azure DevOps.

Si tenemos nuestro proyecto subido a otra plataforma de control de código, o simplemente queremos subirlo pro primera vez vamos a tener que ir a la opción Repos > Files.



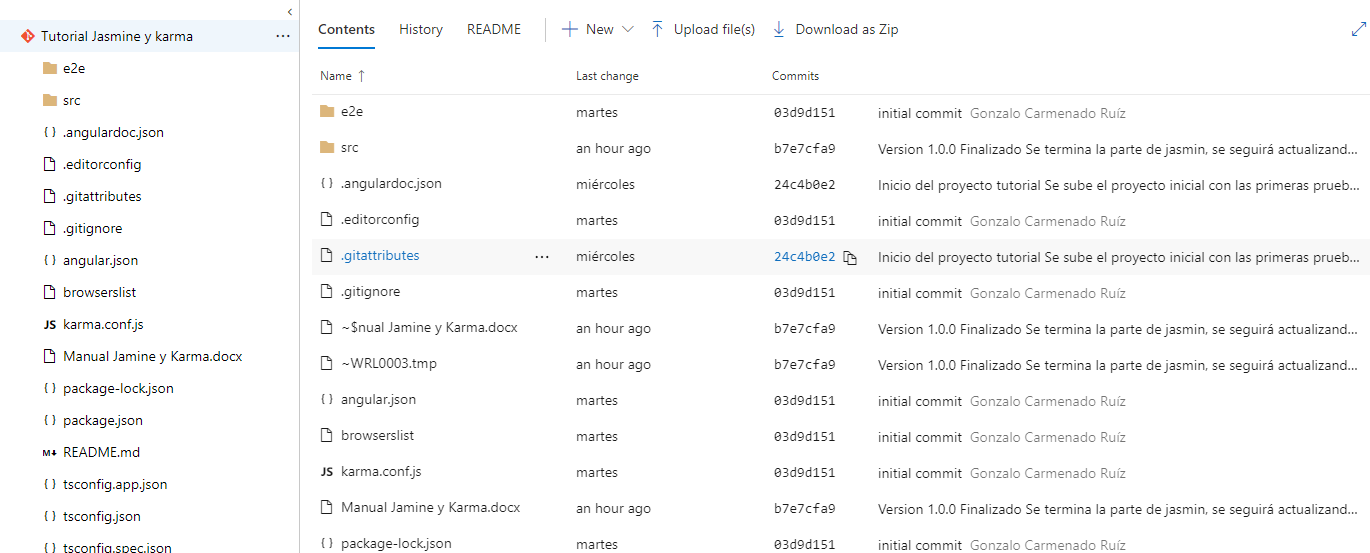
\*43

Y desde esta ventana podemos hacer la carga del proyecto que nosotros queramos, en mi caso será desde GitHub.



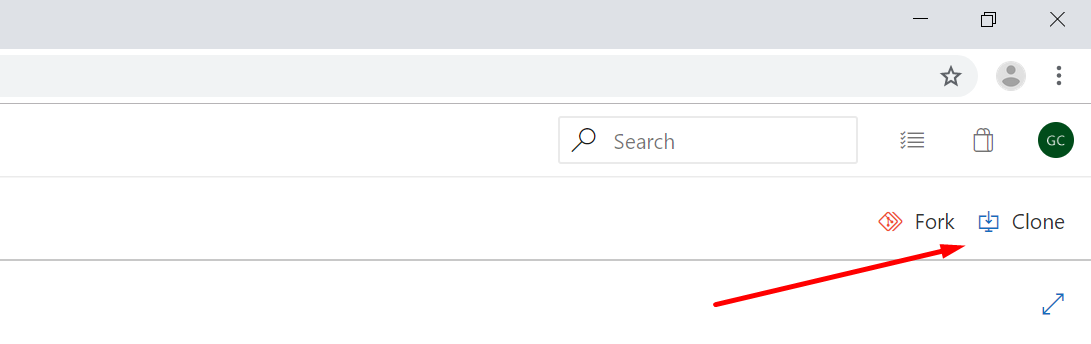
\*44

Si todo ha ido bien nos mostrará un mensaje diciendo que se ha realizado correctamente y nos mostrara la estructura de carpetas de nuestro proyecto.



\*45

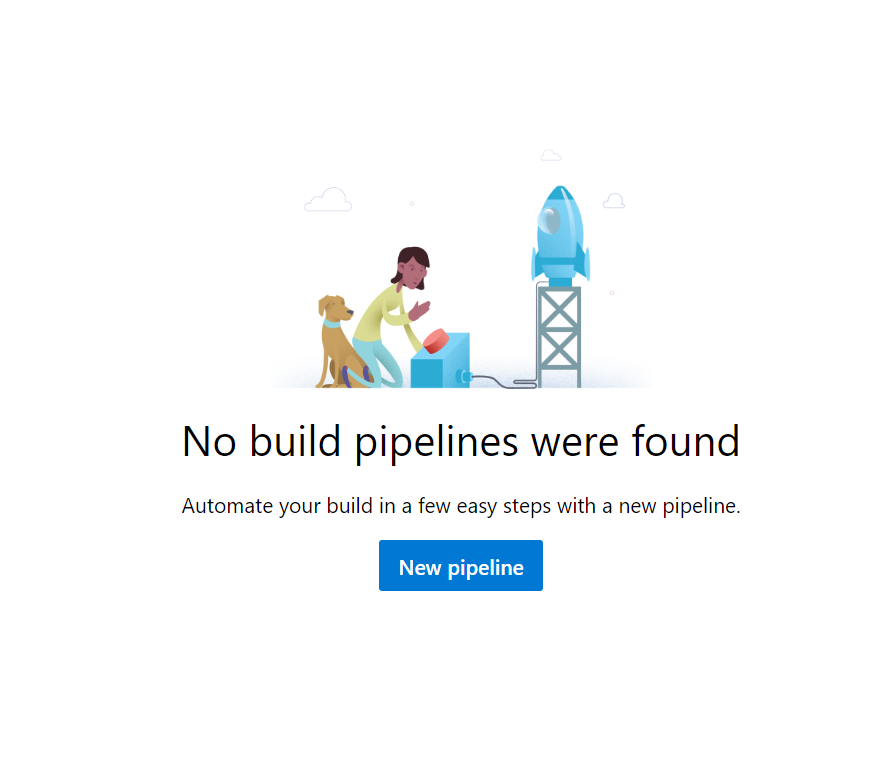
Ahora tenemos que descargarnos el proyecto, ya que así vendrá todo configurado directamente para hacer los pull/push. Para esto nos vamos a Repos > Files > Clone, situado en la parte superior derecha.



Una vez tenemos el proyecto en local, es necesario comenzar a crear nuestras Pipeline para generar cadenas de tareas cada vez que e detecte un cambio en una rama.

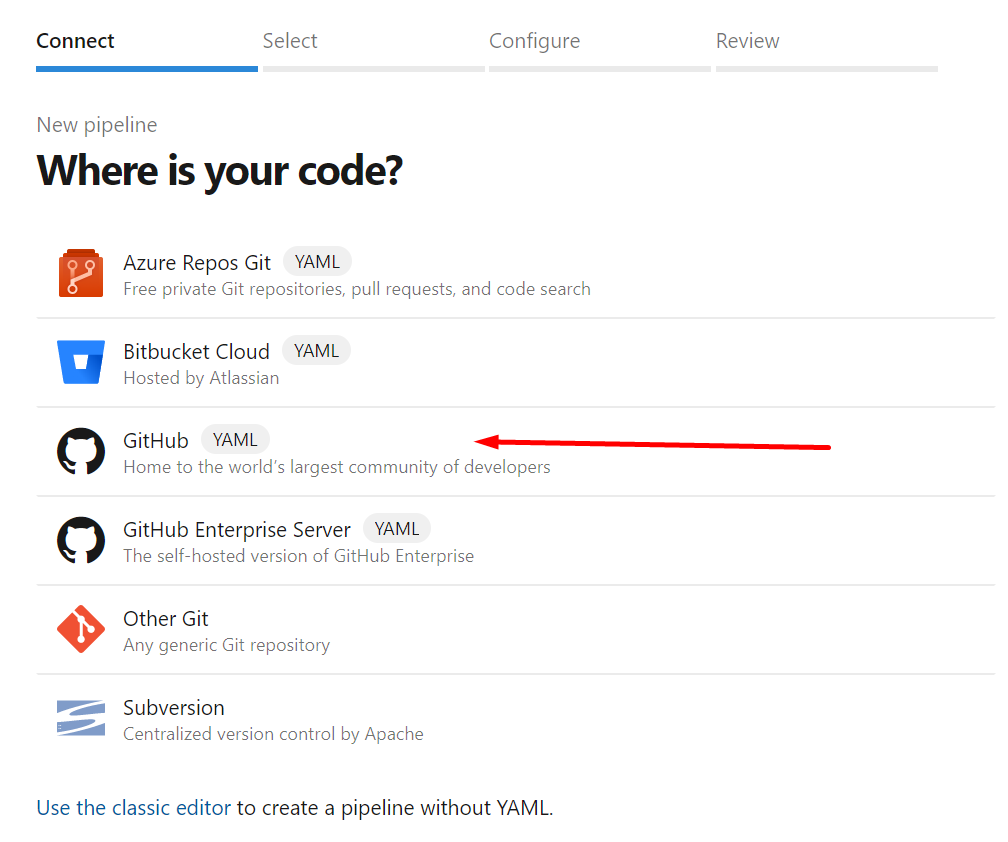
### 14.1. Pipelines.

Para crear una Pipeline tenemos que irnos en el menú de la izquierda a la opción Pipeline y después al botón “new Pipeline”.



\*46

Ahora nos aparecerá una ventana en la cual elegir que repositorio vamos a utilizar, en este caso yo voy a utilizar GitHub, pero cada uno puede usar el que desee en función de donde prefiera/deba tener alojado su proyecto.



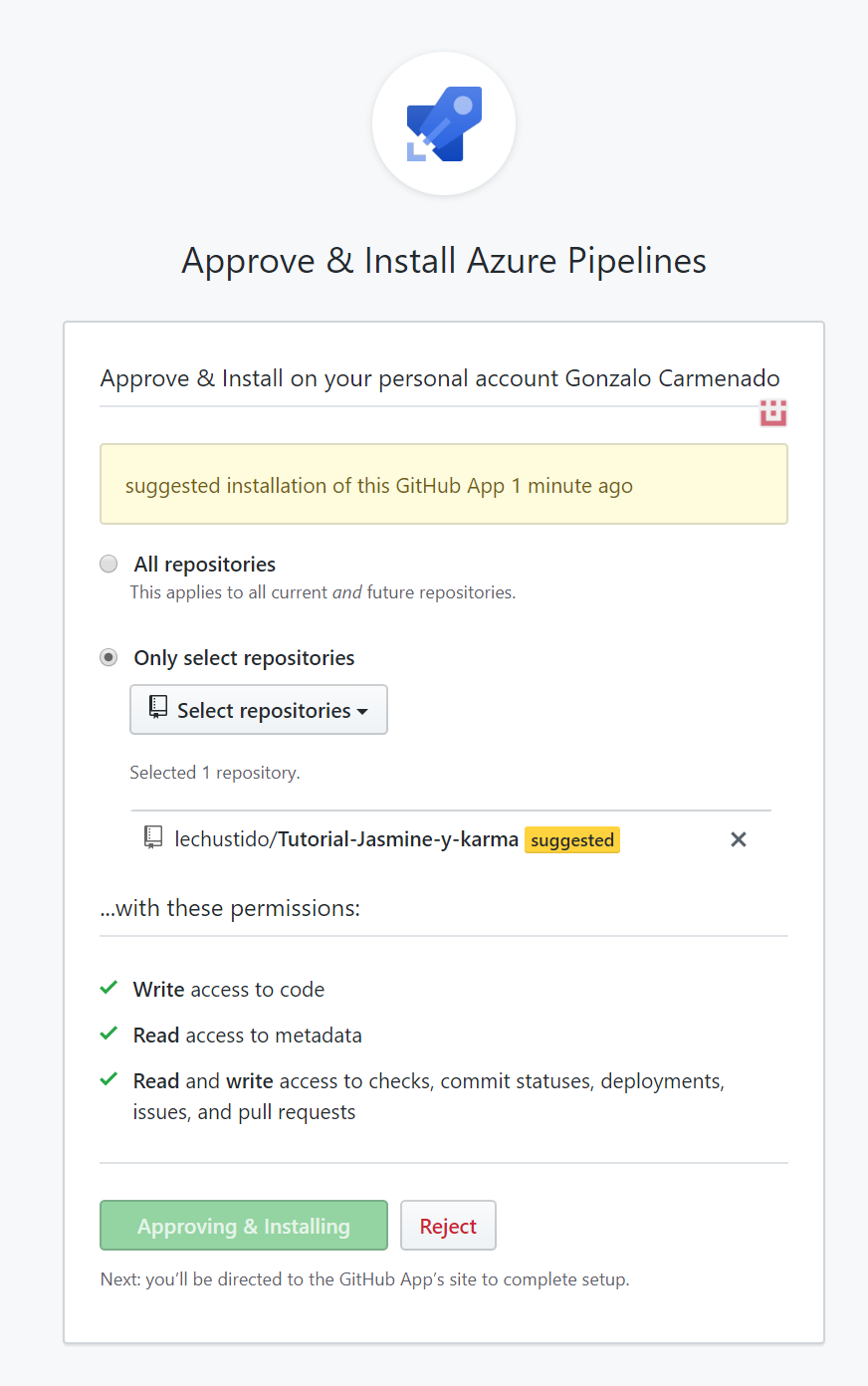
\*47

En la ventana que nos aparece ahora será necesario elegir nuestro proyecto.



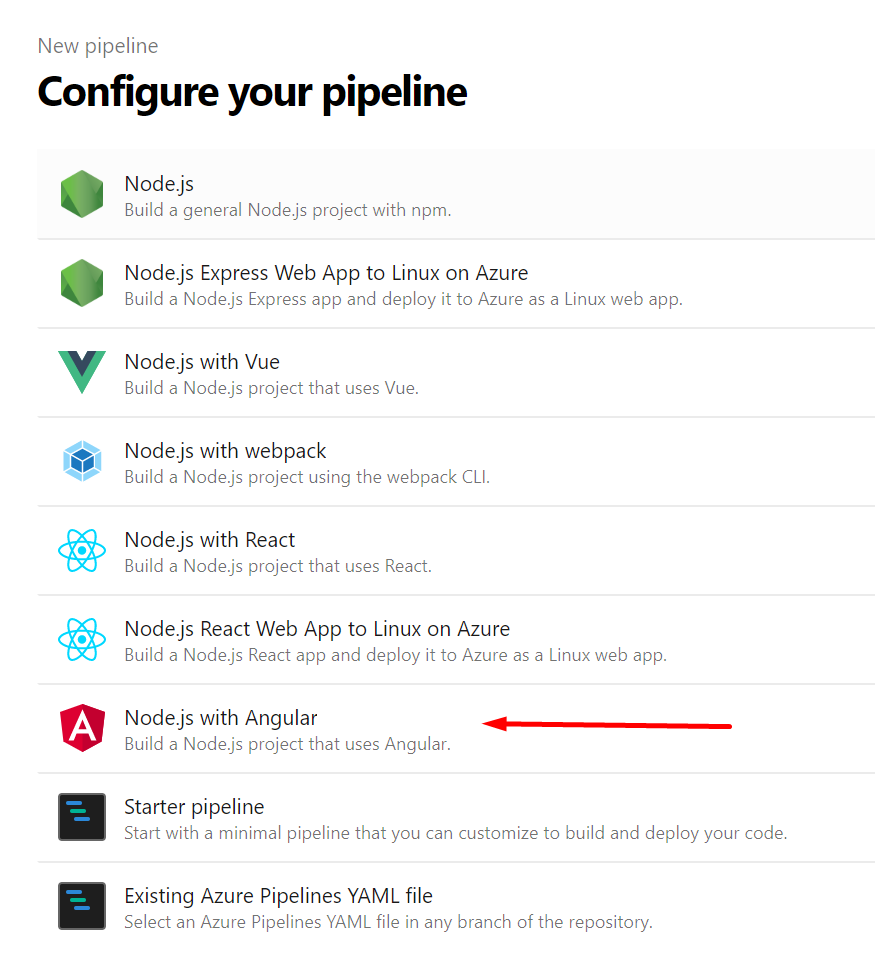
\*48

Llegados a este punto, si ho habéis elegido la opción “Azure Repos” es muy probable que os empiecen a pedir permisos para poder acceder a vuestra cuenta. Es totalmente necesario que aceptéis esto, ya que, si no, Azure no será capaz de detectar cambios y ejecutar las pruebas.



\*49

Cuando todo este aceptado, tendremos que elegir qué tipo de configuración de Pipeline vamos a utilizar. Nos aparecerá una lista enorme de tipos, pero de momento vamos a utilizar una ya configurada especialmente para Angular.



\*50

Aquí es necesario destacar, que elegir cualquiera de estas no te vincula a nada, es decir, es solo una plantilla con algunas configuraciones básicas, las cuales puedes borrar, añadir o modificar en cualquier momento y sin ningún problema.

Una vez hemos seleccionado la de Angular, nos aparecerá una ventana con el código de ejecución de nuestra Pipe.



\*51

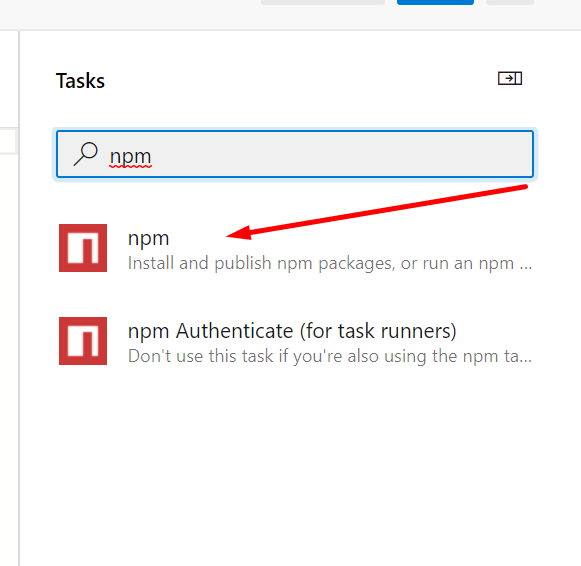
1-Trigger: aquí debemos indicar el nombre de las ramas sobre las que vamos a desarrollar la integración continua.

2-Pool: aquí especificaremos el servidor que se va a utilizar para realizar los procesos, para ver todos los servidores que puedes utilizar y las características de cada uno te dejo este enlace a la página oficial de Microsoft, la cual estará siempre actualizada y te aportará toda la información que necesitas, pero demomento vamos a dejar el que viene por defecto (<https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/pools-queues?view=azure-devops> ).

3-Steps: aquí podremos especificar tareas previas y necesarias para nuestra Pipe. De momento vamos a dejar la que viene, que instalara nuestro servidor “node.js” para poder realizar el resto de tareas.

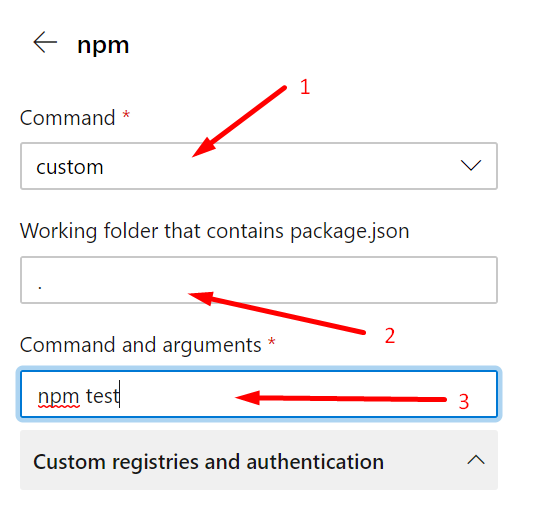
4-Scripts: aquí podremos poner de manera secuencial todas las tareas que deberá hacer nuestra Pipe. Por defecto viene lo que hay en la foto 51, pero nosotros debemos añadir una tarea nueva, que efectivamente es la que nos va a permitir hacer las pruebas unitarias.

Para esto tenemos que ir a las opciones de la derecha y pulsar sobre nueva tarea, buscar o filtrar las opciones por “npm”



\*52

Una vez tenemos esto, pinchamos en la opción “npm “ y rellenamos la pantalla con los datos.



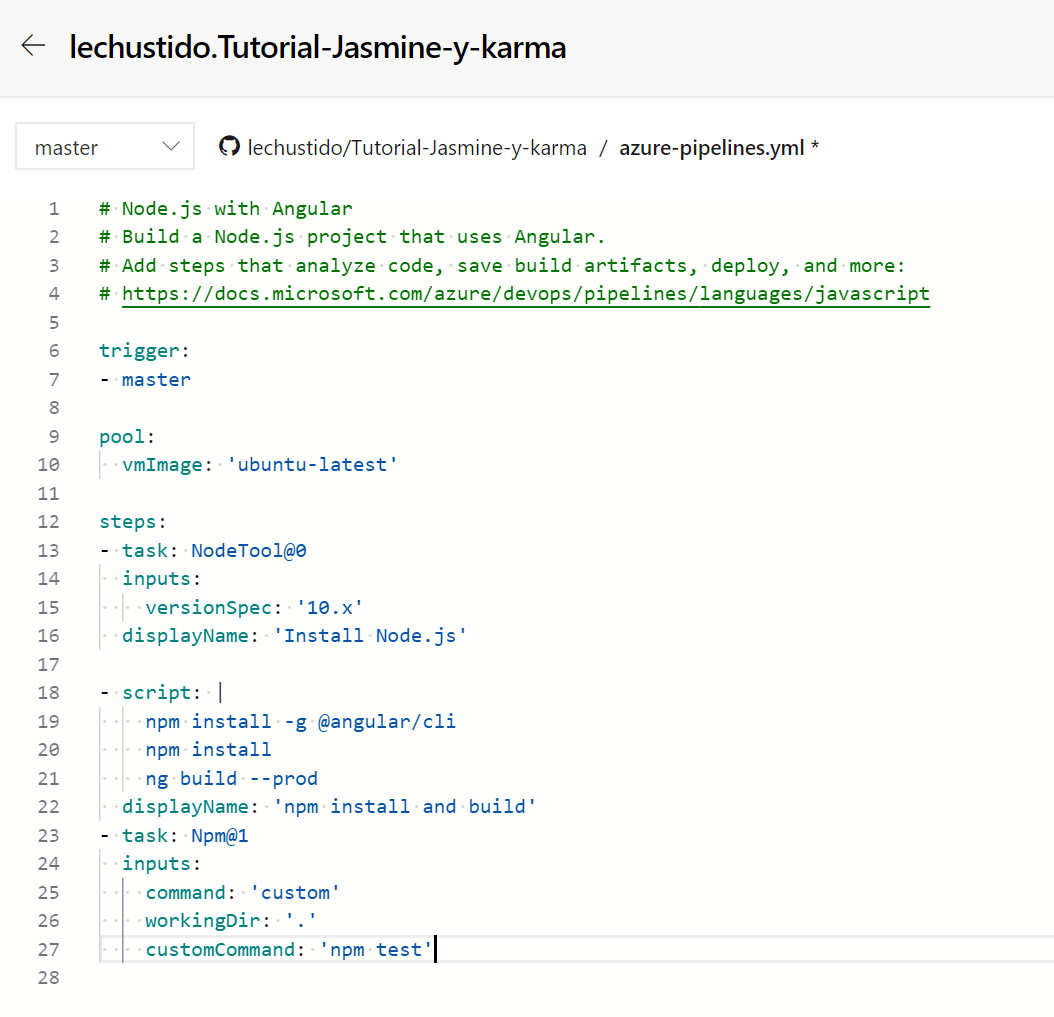
\*53

1- Elegimos la opción “custom”, ya que no vamos ni a hacer una instalación ni una publicación.

2- Aquí debemos indicar la ruta en la que se encuentra nuestro packaje.json, por lo que ponemos un “.” Para indicarle que es en la raíz.

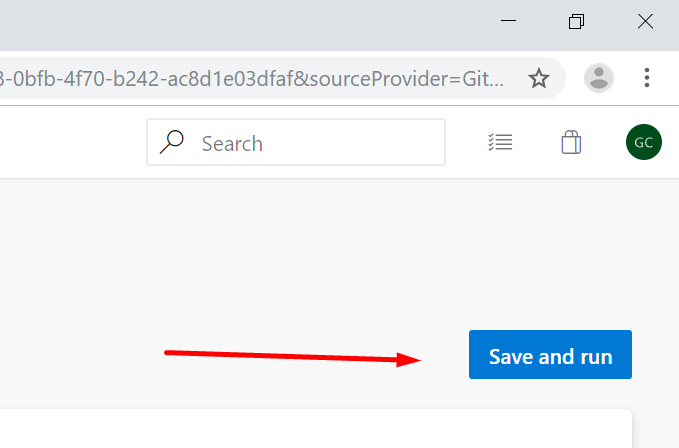
3- Aquí ponemos el comando que deseamos que ejecute nuestra Pipe.

Esto dejará un script de compilación similar a este.



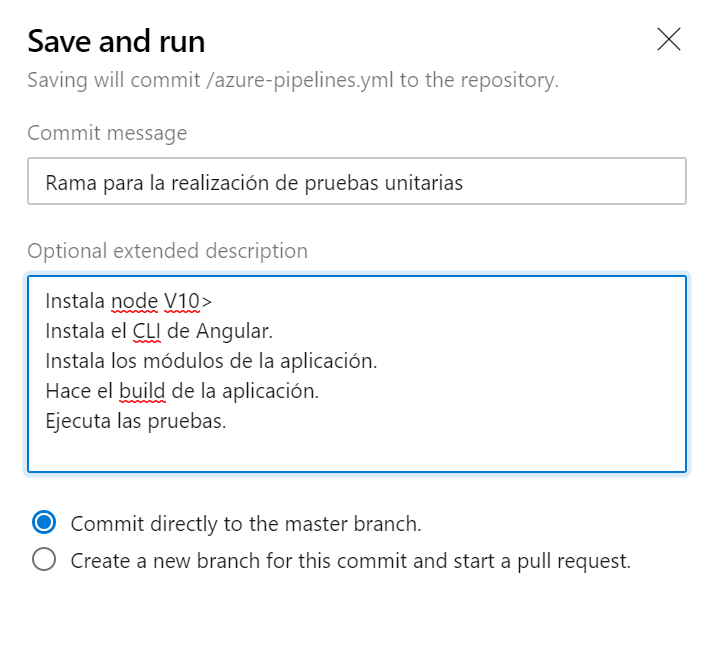
\*54

Después, le damos al botón “Save and run” ubicado en la parte superior derecha.



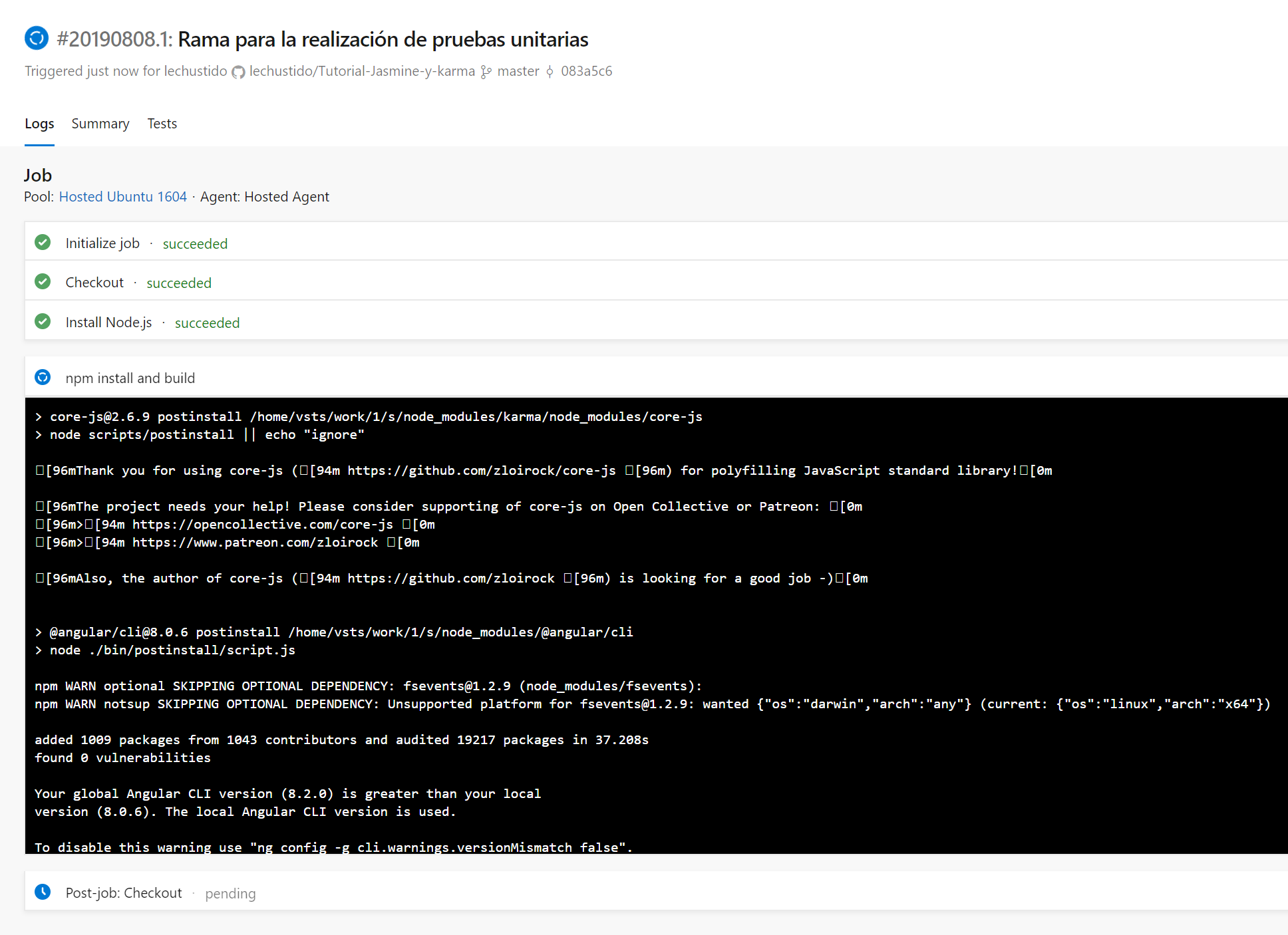
\*53

Ahora nos saltará un mensaje emergente en el cual nos pedirá un mensaje para hacer el comit. En este tipo de mensajes siempre recomiendo poner la mayor cantidad de información posible respecto a lo que se desea que haga esta rama, ya que apenas nos consume tiempo y dejará muy claro a futuros programadores para que vale.



\*54

Después de esto, el navegador avanzará a una página web que contendrá un Log de los procesos que está haciendo Azure.



\*55

Como se puede ver, lo que está haciendo ahora mismo nuestra Pipe, es ejecutar uno por uno y en orden todos los comandos que le hemos especificado en las fotos 51 y 52.

\*56

## Enlaces de interés.

Espero que este tutorial te haya valido para comprender un poco que son las pruebas unitarias y como validar cada uno de los métodos, servicios y el DOM de tu aplicación.

Aquí dejo un par de enlaces que te van a ayudar a seguir ampliando tus conocimientos sobre Jasmine y karma.

DOC Jasmine: <https://jasmine.github.io/pages/docs_home.html>

GitHub para descargarse el proyecto: <https://github.com/lechustido/Tutorial-Jasmine-y-karma>