Testing con Diagramas de Canicas

> *Esta guía se refiere al uso de los diagramas de canicas usando el nuevo `testScheduler.run(callback)`. Algunos detalles aquí expuestos no son aplicables al utilizar el `TestScheduler` de forma manual, sin el uso de la función auxiliar `run()`.*

Podemos hacer tests *síncronos* y de forma determinista a nuestro código RxJS *asíncrono*, mediante la virtualización del tiempo con un `TestScheduler`. Los diagramas de canicas ASCII proporcionan una forma visual de representar el comportamiento de un Observable. Se pueden utilizar para aseverar que un determinado Observable se comporta de forma esperada, además de para crear Observables fríos y calientes, que podemos utilizar como simulaciones.

En este momento, el `TestScheduler` puede utilizarse únicamente para hacer tests al código que utilice temporizadores, como delay/debounceTime etc. (ej: utiliza `AsyncScheduler` con `delays > 1`). Si el código consume una Promesa o lleva a cabo la planificación con `AsapScheduler`/`AnimationFrameScheduler` etc. no se pueden llevar a cabo tests con `TestScheduler` de forma fiable, sino que en su lugar se deberían hacer los test de una forma más tradicional. Ver la sección Problemas conocidos para obtener más detalles.

```
import { TestScheduler } from "rxjs/testing";

const testScheduler = new TestScheduler((actual, expected) => {
  // Aseverando que los dos objetos son iguales
  // Ej. usando chai.
  expect(actual).deep.equal(expected);
});

// Este test se ejecutará de forma "síncrona"
it("generate the stream correctly", () => {
  testScheduler.run((helpers) => {
    const { cold, expectObservable, expectSubscriptions } = helpers;
    const e1 = cold("-a--b--c---|");
    const subs = "^----------!";
    const expected = "-a-----c---|";

    expectObservable(e1.pipe(throttleTime(3, testScheduler))).toBe(expected)
    expectSubscriptions(e1.subscriptions).toBe(subs);
```

```
  });
});
```

API

La llamada a la función *callback* que se proporciona a `testScheduler.run(callback)` se realiza con el objeto `helpers` que contiene las funciones que se utilizan para escribir los tests.

> *Cuando el código contenido en la callback se está ejecutando, cualquier operador que utilice temporizadores/`AsyncScheduler` (como `delay`, `debounceTime` etc.) utilizará el `TestScheduler` de forma **automática**, para proporciona "tiempo virtual". Ya no es necesario proporcionarles el `TestScheduler`, como se hacía anteriormente.*

```
testScheduler.run((helpers) => {
  const { cold, hot, expectObservable, expectSubscriptions, flush } = helpers;
  // Aquí se pueden usar los helpers
});
```

A pesar de que `run()` se ejecuta de forma completamente síncrona, ¡Las funciones auxiliares de la función *callback* no! Estas funciones planifican aseveraciones que se ejecutan bien cuando la *callback_^esté completa, o cuando se llame explícitamente al método `flush()`. Se debe tener cuidado al llamar a las aseveraciones síncronas, como por ejemplo `expect`, de la biblioteca de testing que se haya elegido, desde el interior de la _callback*. Ver Aseveraciones Síncronas para obtener más información sobre cómo hacer esto.

`hot(marbleDiagram: string, values?: object, error?: any)` - crea un Observable "caliente" (como un Sujeto) que se comportará como si se estuviese "ejecutando" cuando el test comience. Una diferencia intersante es que las canicas `hot` permiten un carácter `^` para señalar la ubicación del '*frame* cero'. Este es el punto por defecto (se puede configurar - ver `expectObservable`) en el que comienza la suscripción a los Observables a los que se les esté realizando el test.

`cold(marbleDiagram: string, values?: object, error?: any)` - crea un Observable "frío" cuya suscripción comienza al empezar el test.

`expectObservable(actual: Observable<T>, subscriptionMarbles?: string).toBe(marbleDiagram: string, values?: object, error?: any)` - planifica un aserto para el momento en el que `TestScheduler` haga la llamada automática a la función `flush()`. Se puede cambiar la planificación de la suscripción y la cancelación de la suscripción mediante el parámetro `subscriptionMarbles`. Si no se proporciona el parámetro `subscriptionMarbles`, se realizará una suscripción al comienzo y nunca se llevará a cabo la cancelación. Ver Diagrama de canicas de suscripción.

`expectSubscriptions(actualSubscriptionLogs: SubscriptionLog[]).toBe(subscriptionMarbles: string)` - al igual que `expectObservable`, planifica un aserto para el momento en el que `TestScheduler` haga la llamada automática a la función `flush()`. Tanto `cold()` como `hot()` devuelven un Observable con una propiedad `subscriptions` de tipo `SubscriptionLog[]`. Se proporciona el parámetro `subscriptions` a `expectSubscriptions` para para aseverar si este coincide con el diagrama de canicas `subscriptionsMarbles` proporcionado en `toBe()`. Los diagramas de canicas de suscripción son ligeramente diferentes a los diagramas de canicas de observables.

`flush()` - el tiempo virtual comienza inmediatamente. No se suele utilizar, ya que `run()` hace la llamada a `flush()` de forma automática cuando la función *callback* retorna, pero en algunos casos es posible que se quiera llamar más de una vez, o tener más control.

Sintaxis de Canicas

En el contexto de `TestScheduler`, un diagrama de canicas es una cadena que contiene una sintaxis especial que representa eventos que ocurren a lo largo del tiempo virtual. El tiempo progresa en *frames*. El primer carácter de cualquier cadena de canicas siempre representa el *frame* cero, o el comienzo del tiempo. Dentro de `testScheduler.run(callback)`, el `frameTimeFactor` tiene un valor de 1, lo que quiere decir que un *frame* equivale a un milisegundo virtual.

Cuántos milisegundos virtuales represente cada *frame* depende del valor de `TestScheduler.frameTimeFactor`. Por razones de retrocompatibilidad, el valor de `framTimeFactor` es 1 únicamente cuando el código contenido en `testScheduler.run(callback)` se esté ejecutando. Fuera

de `testScheduler.run(callback)`, tiene un valor de 10. Esto es algo que probablemente cambie en una versión futura de RxJS para que siempre tenga un valor de 1.

IMPORTANTE: Esta guía de sintaxis se refiere al uso de diagramas de canicas en el nuevo `testScheduler.run(callback)`. La semántica de los diagramas de canicas al utilizar el `TestScheduler` de forma manual es distinta, y algunas funcionalidades, como la nueva sintaxis de progresión de tiempo, carecen de soporte.

`' '` espacio en blanco: el espacio en blanco horizontal se ignora, y puede utilizarse para alinear verticalmente múltiples diagramas de canicas.

`'-'` *frame*: representa el paso de un *frame* de tiempo virtual (ver la descripción anterior de *frame*).

`[0-9]+[ms|s|m]` progresión de tiempo: la sintaxis de progresión de tiempo permite progresar el tiempo virtual en una cantidad determinada. Es un número, seguido por una unidad de tiempo: `ms` (milisegundos), `s` (segundos) o `m` (minutos) sin espacio entre ellos. Ej: `a 10ms b`. Ver Sintaxis de Progresión de Tiempo para obtener más detalles.

`'|'` completación: representa la completación con éxito de un Observable. Es el Productor del Observable señalando `complete()`.

`'#'` error: representa un error finalizando el Observable. Es el Productor del Observable señalando `error()`.

[a-z0-9] Ej; `'a'` cualquier carácter alfanumérico: representa un valor emitido por el Productor señalando `next()`. Debe considerarse que se puede proyectar esto a un objeto o array, de la siguiente manera:

```
const expected = "400ms (a-b|)";
const values = {
  a: "value emitted",
  b: "another value emitter",
};

expectObservable(someStreamForTesting).toBe(expected, values);
// Esto también funcionaría
const expected = "400ms (0-1|)";
const values = ["value emitted", "another value emitted"];

expectObservable(someStreamForTesting).toBe(expected, values);
```

`'()'` agrupación síncrona: cuando varios eventos tienen que ocurrir en el mismo *frame* de forma síncrona, se utilizan los paréntesis para agrupar dichos eventos. Se pueden agrupar valores `next`, `complete` o `error` de esta manera. La posición de la `(` inicial determina el momento en el tiempo en el que estos valores se emiten. Aunque pueda ser poco intuitivo al principio, el tiempo virtual progresará en un número de frames igual al número de caracteres ASCII en el grupo, incluyendo los paréntesis, después de que todos los valores se hayan emitido de forma síncrona. Ej: `'(abc)'` emitirá los valores de `a`, `b` y `c` de forma síncrona en el mismo frame y avanzará el tiempo virtual en 5 frames, `(abc).length === 5`. Esto es debido a que suele ayudar a alinear verticalmente los diagramas de canicas, pero es un engorro al hacer tests en un escenario real. Ver más acerca de los problemas conocidos.

`'^'` punto de suscripción: (solo para Observables calientes) muestra el punto en el que los Observables a los que se les hace el test se suscriben al Observable caliente. Es el "frame cero" para el OBservable, cada *frame* anterior al `^` será negativo. El tiempo negativo puede parecer superfluo, pero hay ciertos casos avanzados en los que es necesario, normalmente al tratar con `ReplaySubjects`.

Sintaxis de Progresión de Tiempo

La nueva sintaxis de progresión de tiempo está inspirando en la sintaxis de duración de CSS. Es un número (int o float) inmediatamente seguido por una unidad: `ms` (milisegundos), `s` (segundos) o `m` (minutos). Ej: `100ms`, `1.4s` o `5.25m`.

Cuando no ocupa la primera posición en el diagrama, debe utilizarse un espacio en blanco a ambos lados para desambiguarlo de la serie de canicas. Ej: `a 1ms b` necesita los espacios en blanco, porque `a1msb` se interpretaría como `['a', '1', 'm', 's', 'b']`, donde cada uno de los caracteres es un valor que se enviaría como notificación `next()`.

NOTA: Es posible que haya que restar 1 milisegundo al tiempo que se quiera progresar, ya que las canicas alfanuméricas (que representan un valor emitido) *avanzan el tiempo en 1 frame virtual* al ser emitidas. Esto puede resultar poco intuitivo y frustrante, pero a día de hoy es así el funcionamiento.

```
const input = " -a-b-c|";
const expected = "-- 9ms a 9ms b 9ms (c|)";
/*
```

Copy

```
// Dependiendo de las preferencias de cada uno también podrían
// usarse guiones de frame para mantener la alineación vertical con el input
const input = ' -a-b-c|';
const expected = '------- 4ms a 9ms b 9ms (c|)';
// or
const expected = '-----------a 9ms b 9ms (c|)';

*/

const result = cold(input).pipe(concatMap((d) => of(d).pipe(delay(10))));

expectObservable(result).toBe(expected);
```

Ejemplos

`'-'` o `'------'`: Equivale a `never()` o a un Observable que nunca emite ni llega a completarse. `|`: Equivale a `empty()`. `#`: Equivale a `throwError()`. `'--a--'`: Un Observable que espera 2 *frames*, emite un valor `a` y nunca llega a completarse. `'--a--b--|'`: En el *frame* 2 emite `a`, en el *frame* 5 emite `b` y en el frame 8, `complete`. `'--a--b--#'`: En el *frame* 2 emite `a`, en el *frame* 5 emite `b` y en el frame 8, `error`. `'-a-^-b--|'`: Un Observable caliente, en el *frame* -2 emite `a`, en el *frame* 2 emite `b` y en el *frame* 5, `complete`. `'--(abc)-|'`: En el *frame* 2 emite `a`, `b` y `c`, en el *frame* 8, `complete`. `'-----(a|)'`: En el *frame* 5 emite `a` y `complete`. `'a 9ms b 9s c|'`: En el *frame* 0 emite `a`, en el *frame* 10 emite `b`, en el *frame* 10,012 emite `c` y en el *frame* 10.013, `complete`. `'--a 2.5m b'`: En el *frame* 2 emite `a`, en el *frame* 150,003 emite `b` y nunca llega a completarse.

Canicas de Suscripción

El auxiliar `expectSubscriptions` permite aseverar que se ha realizado/cancelado la suscripción a un Observable `cold()` o `hot()` que se haya creado, en el momento indicado. El parámetro `subscriptionMarbles` que se le proporciona a `expectObservable` permite que el test aplace la suscripción a un momento posterior en el tiempo virtual, y/o cancelar la suscripción aunque el Observable al que se le realiza el test no se haya completado.

La sintaxis de las canicas de suscripción es ligeramente distinta a la sintaxis convencional de canicas.

- `'-'` tiempo: El paso de un *frame* de tiempo.

`[0-9]+[ms|s|m]` progresión de tiempo: La sintaxis de progresión de tiempo permite progresar el tiempo virtual en una determinada cantidad. Es un número, seguido por una unidad de tiempo de `ms` (milisegundos), `s` (segundos) o `m` (minutos) sin ningún espacio entre ellos. Ej: `a 10ms b`. Ver Sintaxis de Progresión de Tiempo para obtener más detalles.
`'^'` punto de suscripción: Muestra el punto en el tiempo en el que se ha realizado una suscripción. `'!'` punto de cancelación de suscripción: ¡uestra el punto en el tiempo en el que se ha cancelado una suscripción.

Debería haber, como mucho, un único punto `^` en un diagrama de canicas de suscripción, y como mucho, un punto `!`. Además de esto, el carácter `-` es el único permitido en un diagrama de canicas de suscripción.

Ejemplos

`'-'` or `'------'`: No se ha realizado ninguna suscripción. `'--^--'`: Se ha realizado una suscripción después de haber pasado 2 *frames* de tiempo, y dicha suscripción no se ha llegado a cancelar. `'--^--!-'`: En el *frame* 2 se ha realizado una suscripción, que se ha cancelado en el *frame* 5. `'500ms ^ 1s !'`: En el *frame* 500 se ha realizado una suscripción, que se ha cancelado en el *frame* 1,501.

Dada una fuente caliente, se le hacen test a múltiples suscriptores que se suscriben en momentos diferentes:

```
testScheduler.run(({ hot, expectObservable }) => {
  const source = hot("--a--a--a--a--a--a--a--");
  const sub1 = "      --^-----------!";
  const sub2 = "      ---------^--------!";
  const expect1 = "   --a--a--a--a--";
  const expect2 = "   -----------a--a--a-";
  expectObservable(source, sub1).toBe(expect1);
  expectObservable(source, sub2).toBe(expect2);
});
```

Copy

Cancelación de suscripción manual de una fuente que nunca llega a completarse:

```
it("should repeat forever", () => {
  const testScheduler = createScheduler();
```

Copy

```
  testScheduler.run(({ expectObservable }) => {
    const foreverStream$ = interval(1).pipe(mapTo("a"));

    // Omitir este argumento puede causar un fallo en la suite de test
    const unsub = "------ !";

    expectObservable(foreverStream$, unsub).toBe("-aaaaa");
  });
});
```

Aseveración Síncrona

A veces, es necesario aseverar cambios en el estado *después* de que el flujo Observable se haya completado - como por ejemplo cuando un efecto colateral como `tap` actualiza una variable. En un contexto diferente al de realizar tests con `TestScheduler`, podríamos considerar esto como la creación de un retardo, o esperar cierto tiempo, antes de realizar nuestro aserto.

Por ejemplo:

```
let eventCount = 0;

const s1 = cold("--a--b|", { a: "x", b: "y" });

// Efecto colateral usando 'tap' para actualizar una variable
const result = s1.pipe(tap(() => eventCount++));

expectObservable(result).toBe("--a--b|", ["x", "y"]);

// flush - ejecuta el 'tiempo virtual' para completar todos los Observables
flush();

expect(eventCount).toBe(2);
```

En la situación anterior necesitamos que el flujo Observable se complete para poder comprobar si la variable contiene el valor adecuado. El `TestScheduler` se ejecuta en 'tiempo virtual' (de forma síncrona), pero no se suele ejecutar (y completar) hasta que

la *callback* testScheduler haya retornado. El método `flush()` dispara de forma manual el tiempo virtual para que se pueda comprobar el valor de la variable local después de que el Observable se haya completado.

Problemas Conocidos

No se pueden hacer tests sobre código RxJS que consuma Promesas o utilice cualquiera de los otros planificadores (Ej: AsapScheduler)

Si se tiene código RxJs que utilice cualquier forma de planificación asíncrona que no sea la de `AsyncScheduler`, como por ejemplo Promesas, `AsapScheduler` etc. no se pueden utilizar diagramas de canica de forma fiable para *ese código en particular*. Esto es debido a que todos los demás métodos de planificación no se podrán virtualizar, ni son reconocidos por `TestScheduler`.

La solución a eso es hacerle tests a ese fragmento de código aislado, con los métodos de testing tradicionales del framework de tests que se esté utilizando. Los detalles dependen del framework que se utilice, por lo que veremos un ejemplo en pseudo-código:

```
// Código RxJS que consume una Promesa, por lo que TestScheduler no será ca
// de virtualizar correctamente y el test será asíncrono
const myAsyncCode = () => from(Promise.resolve("algo"));

it("has async code", (done) => {
  myAsyncCode().subscribe((d) => {
    assertEqual(d, "algo");
    done();
  });
});
```

Un hecho relacionado es que, actualmente, no se pueden realizar asertos con retardos de cero, aunque se utilice `AsyncScheduler`. Ej: `delay(0)` es como decir `setTimeout(work, 0)`. Esto planifica una nueva "tarea" o "macrotarea", por lo que es asíncrono, pero sin un paso del tiempo explícito.

El comportamiento es distinto en el exterior de `testScheduler.run(callback)`

El `TestScheduler` existe desde v5, pero su uso estaba pensado más para que se llevasen a cabo los tests del framework RxJS por sus mantenedores, que para uso en aplicaciones de usuarios. Debido a esto, algunos de los comportamientos por defecto y funcionalidades del `TestScheduler` no funcionaban bien (o directamente no funcionaban) para los usuarios. En v6 se introdujo el método `testScheduler.run(callback)` que permitió proporcionar nuevos valores por defecto y funcionalidades manteniendo la compatibilidad, aunque sigue siendo posible utilizar el `TestScheduler` fuera de `testScheduler.run(callback)`. Es importante tener en cuenta que, en el caso de hacerlo, hay algunas diferencias importantes en su comportamiento:

- Los métodos auxiliares de `TestScheduler` tienen nombres más verbosos, como `testScheduler.createColdObservable()` en lugar de `cold()`.
- La instancia de `TestScheduler` NO se utiliza automáticamente por los operadores que utilizan `AsyncScheduler`. Ej: `delay`, `debounceTime` etc. por lo que hay que pasársela explícitamente.
- NO hay soporte para la sintaxis de progresión de tiempo. Ej: `-a 100ms b-|`.
- Un *frame* equivale a 10 milisegundos virtuales pr defecto.
- Ej: `TestScheduler.frameTimeFactor = 10`.
- Cada espacio equivale a un *frame*, igual que el guión `-`.
- Hay un máximo de número de *frames* con un valor de 750. Ej: `maxFrames = 750`. Después de 750, se ignoran silenciosamente.
- Hay que llamar a `flush()` de forma explícita.

Aunque en este momento el uso de `TestScheduler` fuera de `testScheduler.run(callback)` no haya sido declarado obsoleto de forma oficial, está desaconsejado, ya que es probable que cause confusión.