¿Qué es Rx?

Reactive extensión es una librería funcional y reactiva que empezó como iniciativa de Microsoft para .NET, y que pronto se extendió a muchos más lenguajes. La versión más popular de Rx es RxJS para Javascript (naturalmente).

Rx trata con colecciones asíncronas de llamadas a servidor o eventos de usuario, intentando hacerlas más fácil de trabajar con ellos.

Rx se basa en paradigmas de programación funcional y reactiva.

Del paradigma funcional toma la idea de funciones que no dependen de estado externo ni generan efectos secundarios, el resultado sólo depende de los parámetros. También el concepto de inmutabilidad, donde las funciones devuelven nuevos datos y objetos, dado que no deben modificar el estado de elementos externos a la función. Los métodos funcionales de Rx están creados con composición en mente, resultando en expresiones complejas compuestas de simples métodos uno detrás de otro.

Del paradigma reactivo toma la idea de que el estado de la aplicación es resultado directo del input del usuario. Aunque no hace falta escribir aplicaciones reactivas con Rx, se puede usar como una librería adicional en cualquier programa.

En el centro de Rx están los Observables, colecciones en el tiempo que lanzan sus valores al observador para que éste los maneje. Este es la idea principal de Rx; la diferencia entre iterar sobre un array (u otra colección) y sacar (patrón pull) valores manualmente, y un observable que lanza (patrón push) su valores cuando llega su tiempo.

Ventajas sobre promesas:

-Promesas son un bloque de código que se ejecuta inmediatamente y se usan cuando resuelven un valor. Si algo falla y tienes que repetir esa promesa, o quieres ejecutar la misma promesa en dos momentos del programa, tienes que repetir el código dentro de la promesa; que se ejecutará dos veces. Si no necesitas una promesa que has ejecutado, tienes que esperar a que acabe e ignorar su resultado.

-Observables, en cambio, sólo se ejecutan cuando te subscribes a ellos; tienen evaluación perezosa. Puedes declarar una colección infinita sin problemas pues no se evaluará hasta la subscripción, y entonces sólo se evaluará cada valor uno a uno. Te puedes subscribir varias veces al mismo observable sin tener que declararlo varias veces. También puedes componer varios observables, pausar, reintentar o cancelarlos; ahorrando tiempo y recursos.

Ventajas sobre eventos:

-Eventos son más verbosos, requieren más línas de código. Son bastante rígidos, debes añadirlos a un elemento y debes removerlos cuando no los necesites para evitar *memory leaks* (una de las causas frecuentes de *memory leaks* son eventListeners olvidados).

-Observables son funcionales, más fluidos y concisos. Al terminar un observable de eventos se remueve automáticamente, o se pueden definir fines alternativos para ellos; de modo que te aseguras de que siempre acabarán.

Diagrama de canicas (marble diagram):

El diagrama de canicas es la forma establecida de visualizar un stream de observables en el tiempo. Y sea con imágenes o con simples caracteres ASCII, este tipo de diagramas es la más usada a la hora de representar observables.

La flecha dirigida de izquierda a derecha representa el tiempo.

Los nodos esparcidos en la flecha son los valores de la colección de observables, y la distancia entre ellos es el intervalo transcurrido entre la recepción de un valor y el siguiente. Si varios valores son recibidos al mismo tiempo, se dibujan en el mismo nodo o en nodos contiguos.

Una X en la línea temporal significa un error en la generación o recepción de un valor observable. Normalmente, el stream termina en cuanto se produce un error.

Una barra vertical significa el final del stream. Al finalizar un stream la subscripción al observable termina automáticamente.

Elementos básicos de Rx:

-Observables: El elemento central de Rx. Observables son colecciones asíncronas de cero o más valores. Se pueden generar a mano o se pueden convertir promesas, eventos, etc en observables.

-Observers: Objeto con funciones onNext(), onError() y onComplete() que escucha la subscripción a un observable y se encarga de los valores lanzados.

-Subscriptions: Al subscribirse a un observable se obtiene un objeto subscripción, que contiene una función dispose() que se puede llamar para cancelar la subscripción al observable y dejar de escuchar.

-Subjects: Un objeto genérico proporcionado por la librería Rx. Puede utilizarse tanto como observable como observer, y tiene varios usos.

-Schedulers: Algunos métodos de Rx aceptan schedulers como extra parámetros. Schedulers sirven para definir concurrencia en múltiples threads al trabajar con observables.

Crear Observables:

-create(*function*) es el método para crear observables manualmente. Se le pasa una función que define que valores (o errores) van a formar parte de la colección, y si va a completarse y cuando. Esta función puede retornar una función que será llamada en caso de llamar dispose() sobre este observable.

-empty() crea un observable sin valores que termina inmediatamente. Usado principalmente en tests.

-for(*source, function*) recibe un array (*source*) y crea un observable donde cada valor es el resultado de aplicar la función recibida a cada valor del array.

-fromEvent(*element, event*) crea un observable con cada evento generado al producirse el evento recibido como parámetro string (*event*) en el elemento dado (*element*).

-fromPromise(*promise*) convierte una promesa en un observable.

-interval(*period*) admite un numero representando un periodo en milisegundos y genera un observable que lanza valores enteros empezando desde 1 espaciados ese periodo.

-just(*value*) o return(*value*) devuelve un observable que lanza el valor recibido como parametro e inmediatamente finaliza.

-never() crea un observable sin valores y que no termina nunca. Usado en tests.

-of(*arguments*) genera un observable con los argumentos como valores. Admite múltiples tipos de argumentos, incluso diferentes tipos en el mismo observable.

-range(*start, count*) genera un observable con valores enteros empezando por el inicio (*start*) y con un numero de valores igual a la cuenta (*count*). Finaliza inmediatamente al acabar.

Controlar Observables:

-subscribe() o forEach() se ejecuta sobre un observable y llama al objeto observer dado como parámetro cada vez que el observable lanza un valor, error, o finaliza.Tambien se puede suministrar funciones, que serán llamada cada vez que el observable lanza un valor, un error, o finaliza (en ese orden). Otros métodos aplicados al observable antes de la subscripción no se ejecutarán hasta que te subscribas. Si el observable es frío\*, el observable empezará en el momento de subscibirse. Este es el método final llamado sobre un observable.

-startWith() genera un nuevo observable concatenando el valor o observable dado como parámetro antes del observable sobre el que se llama este método.

-do() se usa como un map(), llamando la función suministrada para cada valor del observable. La diferencia es que do() está pensado para funciones que no modifican el observable, sino que actúa fuera del observable. Devuelve el mismo observable inicial.

-finally() es un método que define una función que se ejecutará al finalizar el observable, al igual que el método onCompleted() del observer.

-debounce() recibe un numero de milisegundos y genera un observable formado por los valores del observable inicial que sean precedidos por ese numero de milisegundos de silencio. Sirve para filtrar una gran cantidad de valores posiblemente repetidos.

-timeinterval() coge los valores del observable inicial y genera un nuevo observable con objetos, de propiedades *value* (el valor cogido) y *interval* (el tiempo en milisegundo de distancia de este valor con el anterior.

-timeStamp() es similar al anterior, pero en vez de *interval* genera *timestamp*, el timestamp de cada valor.

-retry() reintenta el observable el numero proporcionado de veces. Normalmente un observable termina al encontrar un error, este método sirve para evitar perder una colección entera por un error puntual de servidor o un timeout ocasional.

Comprobar Observables:

-every() ejecuta la condición parámetro sobre cada valor del observable y devuelve un booleano que depende de si todos los valores cumplen la condición.

-some() es igual que every() pero devuelve true si cualquiera de los valores cumple la condición.

-includes() recibe un valor como parámetro y lo busca en el observable, devolviendo un booleano.

-isEmpty() comprueba si el observable está vacío o si contiene algún valor.

Calcular Observables:

-average() devuelve un observable con un valor, la media de todos los valores del observable inicial. El observable inicial debe terminar para que este método funcione.

-sum() devuelve un observable con un valor, la suma de todos los valores del observable inicial. El observable inicial debe terminar para que este método funcione.

-count() devuelve un observable con un valor, la cantidad de valores del observable inicial. El observable inicial debe terminar para que este método funcione.

-reduce() tiene como parámetro una función que *reduce* los valores del observable y devuelve un observable con ese valor. Como segundo parámetro opcional se le puede pasar el valor inicial de la función *reduce*. El observable inicial debe terminar para que este método funcione.

-scan() actúa como el reduce(); pero en vez de devolver un solo valor al final, scan devuelve un observable con cada valor intermedio del cálculo y NO devuelve el valor final. No requiere que el observable inicial termine para funcionar.

Filtrar Observables:

-filter() o where() funciona como el filter normal de javascript, recibe una función y filtra el observable de acuerdo a dicha función.

-slice() igual que el slice normal, devuelve una parte del observable.

-take() toma el numero deseado de elementos desde el principio.

-takeLast() es igual que take, pero toma valores del final. El observable debe acabar.

-first() toma el primer valor.

-last() toma el último valor. El observable debe terminar.

-skip() ignora el número deseado de valores del observable.

-skipLast() es igual que skip() pero ignorando valores del final del observable. Dicho observable debe finalizar.

-distinct() filtra valores repetidos, creando un observable con todos los valores diferentes.

-distinctUntilChanged() filtra valores repetidos contiguos, iguales al valor inmediatamente anterior. Sirve para no realizar el mismo calculo con el mismo dato dos veces seguidas.

-takeWhile() recibe una función condición y coge valores que cumplen la condición desde el principio, terminando en cuanto uno no la cumpla.

-takeUntil() recibe un observable (o una promesa) y toma valores del observable inicial hasta que el observable inicial emite algún valor.

Transformar observables:

-map() o select() funciona como un map normal.

-toArray() convierte el observable en un array con sus valores.

-defaultIfEmpty() devuelve el mismo observable. Excepto si está vacío, donde devuleve un observable con un solo valor por defecto (dado como parámetro).

-groupBy() recibe una condición y agrupa los valores de acuerdo a esa condición. Devuelve un observable de observables.

Combinar Observables:

-concat() crea un nuevo observable concatenando el observable parámetro al observable inicial.

-merge() devuelve un observable con todos los valores de ambos observables en sus tiempos correspondientes.

-combineLatest() toma dos observables y cada vez que alguno lanza un valor, devuelve el último valor lanzado por ambos. Se puede suministrar una función opcional para manejar esos dos valores.

-zip() toma dos observables y devuelve un observable con parejas de valores. Sólo toma un único valor de cada observable en cuanto hay pareja disponible. Se puede suministrar una función opcional para manejar esos dos valores.

Caliente y frío:

Los observables se distinguen entre fríos y calientes.

Los observables fríos son aquellos que se crean al subscribirse y cada subscripción es independiente y generan un observable nuevo diferente. Por ejemplo al crear observables manualmente (con interval) cada vez que te subscribas, recibes una nueva colección con los mismos valores desde el principio.

Los observables calientes son aquellos que reciben valores aunque aún no te hayas subscrito, y al subscribirte recibes una colección con valores desde el momento de subscripción. Si realizas dos subscripciones recibirás la misma colección dos veces. Por ejemplo; al definir un observable capturando eventos, los eventos ocurren aunque no haya nada escuchando, y cuando te subscibes, empiezas a recibir los eventos siguientes.

Los observables fríos se pueden convertir en calientes por conveniencia, usando métodos como share (entre otros); principalmente para conseguir dos subscripciones con los mismos datos.

Ejemplos prácticos I:

-Capturador de movimientos del ratón:

Creando un observable desde el evento ‘mousemove’ en todo el documento capturamos todos los movimientos del ratón y los transformamos en texto que mostramos en pantalla al subscribirnos. Además, a este observable le añadimos el método pausable() para hacer posible el pausar la captura de eventos.

Otro observable desde el evento ‘click’ de un botón, lo convertimos en la cantidad de clicks realizados y lo convertimos otra vez a un booleano, depende de si el número es par o impar. Al subscribirnos cogemos ese booleano y llamamos a pause() o resume() sobre el observable de movimientos del ratón para pausarlo o continuarlo.

Un último observable para terminar la captura de movimientos en el evento ‘click’ del div donde aparece el texto con los movimientos del ratón, que llama al método dispose() de la subscripción del primer observable.

-Autocompletado desde Wikipedia:

Un observable que recoge los eventos ‘keyup’ de una caja de texto, coge el valor dentro de esa caja de texto, filtra sólo los strings mayores de 3 caracteres, llama debounce() para retardar el resto del observable, llama distinctUntilChanged() para coger valores que cambian, usa esos valores en una búsqueda a wikipedia, reintenta 3 veces por si acaso la conexión falla, llama switchlatest() (o switch a secas) para coger sólo el conjunto de valores más reciente, y llama timeInterval() para calcular cuánto ha tardado el proceso. Al subscribirnos a este observable mostramos la palabra que buscamos, el tiempo transcurrido en la búsqueda, la primera línea de descripción de ese artículo en Wikipedia, y un enlace a la página con el artículo entero. Esto para los 10 primeros resultados de cada búsqueda.

Otro observable recoge los eventos ‘keyup’ de la misma caja de texto, coge el valor igual que antes, filtra los strings menores de 2 caracteres y llama a un método propio que limpia la página; de modo que si borras el texto de la caja de texto, los resultados desaparecen.

Ejemplos Prácticos II:

-DDRx:

Esta aplicación tiene el siguiente ciclo de vida:

Empieza con el top 5 récords (inicialmente sólo hay 3 entradas) y un botón para empezar. Al pulsar el botón empieza el juego.

El juego consiste en flechas que aparecen en pantalla y que el jugador tiene que pulsar en el teclado en el mismo orden. Si la flecha pulsada corresponde con la mostrada en pantalla se consiguen 1000 puntos, en caso contrario se pierden 250. Hay un marcador que muestra la puntuación total. Las flechas en pantalla aparecen cada vez más y más rápido hasta el final, donde dejan de aparecer flechas y la aplicación espera que el jugador recuerde (o adivine) el orden de flechas y las pulse. Cuando todas las flechas se han pulsado aparece una caja de texto que te pregunta tu nombre. Lo introduces, das la botón y la aplicación vuelve a la página principal con el top 5; con tu nombre añadido si tu puntuación lo merece. Estos récords se guardan en el localStorage, la memoria del navegador.

No sólo el juego está hecho con observables, sino que el ciclo de funcionamiento de la aplicación también usa observables.

Primero se establecen los récords en el localStorage y se llama a la función main().

Main() coge los récords del localStorage y los muestra en pantalla junto a un botón. Este botón tiene un observable asociado a su evento ‘click’ que acepta un único click y ejecuta la función play().

Play() tiene un observable que en un intervalo decreciente genera direcciones aleatorias y al subscribirse las muestra en pantalla. Otro observable captura los eventos ‘keyup’ de teclado y los filtra para quedarse con las flechas direccionales. Un tercer observable une con zip() los otros dos observables, comprueba si son iguales para dar o quitar puntos, muestra en pantalla los puntos ganados o perdidos, los acumula con scan(), muestra en pantalla la puntuación total, y coge el último valor. Al subscribirse se lanza el método result() con ese valor.

Result() dibuja una caja de texto y un botón y genera un observable desde el evento ‘click’ del botón, que recibe la entrada en la caja de texto, filtra ese nombre para que no esté vacío, guarda ese nombre en los récords del localStorage, y finaliza al tomar un solo valor. Al subscribirse a este observable se lanza el método main() de nuevo, cerrando el ciclo de vida de DDRx.

Más Rx:

Reactive extensions está extendido a un gran número de lenguajes y adaptado a múltiples librerías (incluído react). Además, como no tiene ninguna dependencia externa, es fácil y seguro de implementar; y funciona para todas las aplicaciones posibles, reactivas o no, online o no.