Arquitecturas de Frontend

Arquitecturas Reactivas de Streams

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Octubre 2014

I. ¿Quién Soy?

Licenciado en informática por la Universidad Politécnica de Madrid (UPM) desde el año 2001 y doctor en informática por la Universidad Nacional de Educación a Distancia (UNED) desde el año 2009, Javier es investigador y está especializado en el diseño y análisis de la colaboración. Esta labor la compagina con actividades de evangelización, consultoría, mentoring y formación especializada para empresas dentro del sector IT. Inquieto, ávido lector y seguidor cercano de las innovaciones en tecnología.

javier.velez.reyes@gmail.com

@javiervelezreye

linkedin.com/in/javiervelezreyes

gplus.to/javiervelezreyes

jvelez77

javiervelezreyes

youtube.com/user/javiervelezreyes

II. ¿A Qué Me Dedico?

Desarrollado Front/Back

Evangelización Web

Arquitectura Software

Formación & Consultoría IT

E-learning

Diseño de Sistemas de Colaboración

Learning Analytics

Gamificación Colaborativa

Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

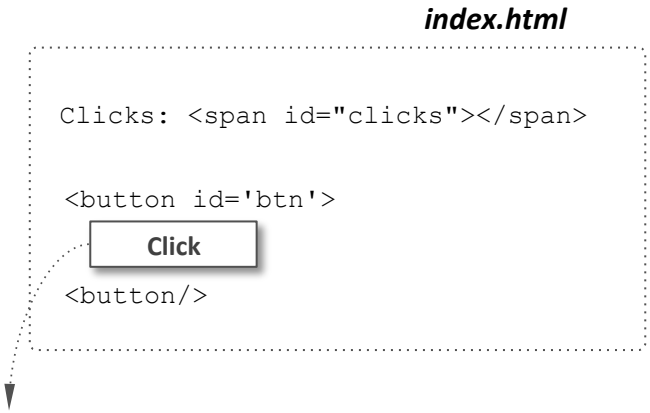
1 Introducción

- La Web Como Modelo Reactivo
- Arquitecturas centradas en el modelo
- Arquitecturas de Programación Funcional Reactiva
- Principios de las Arquitecturas Reactivas

I. El Camino Hacia las Arquitecturas Reactivas

A. La Web Como Modelo Reactivo

La Web, como plataforma de interacción, es un sistema reactivo. El árbol DOM que representa en memoria el documento de la página en curso es un modelo de datos sujeto a cambios sobre el que pueden registrarse funciones escuchadoras para cada uno de los eventos que el sistema lanza como señalización de dichos cambios. La programación basada en escuchadores es el más básico y nuclear de los modelos de construcción de aplicativos Web.

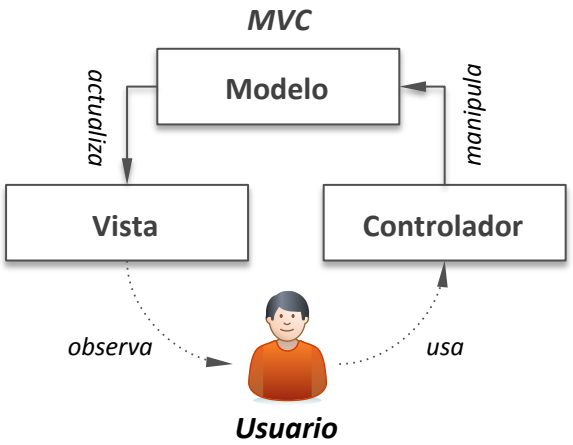



```
var btn = document.querySelector('#btn');
var sClicks = document.querySelector('#clicks');
var nClicks = 0;
btn.addEventListener('click', function () {
    clicks.innerHTML = ++nClicks;
});
```

I. El Camino Hacia las Arquitecturas Reactivas

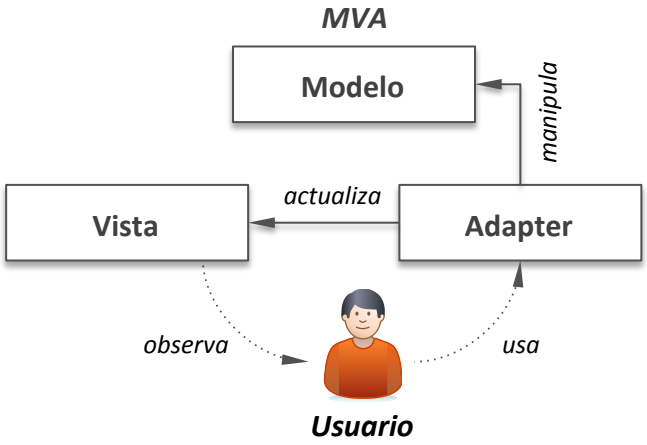
B. Arquitecturas Centradas en el Modelo

Como evolución del modelo de construcción basado en programación de escuchadores, emergieron una serie de arquitecturas que se recogen bajo el acrónico MV* por sus similitudes entre sí. Todas ellas comparten la premisa de que están centradas en un modelo interno cuyos cambios son atendidos por la vista a través de cierta lógica de control. En general este paraguas recoge diversas variantes arquitectónicas.



Model View Controller

El controlador manipula la vista en respuesta a las interacciones del usuario y el modelo actualiza la vista



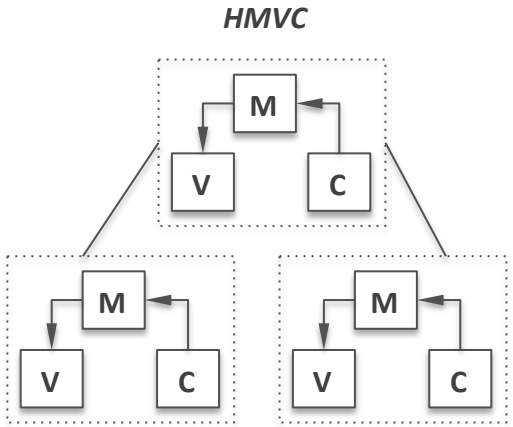
Model View Adapter

El modelo no se comunica con la vista. Es el adaptador el encargado de propagar los cambios en ambas partes

I. El Camino Hacia las Arquitecturas Reactivas

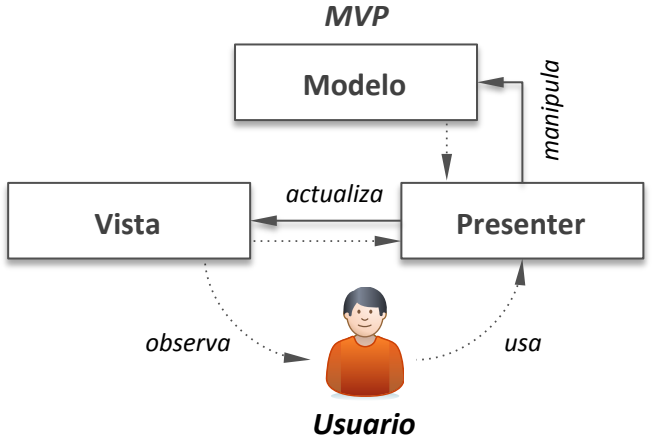
B. Arquitecturas Centradas en el Modelo

Como evolución del modelo de construcción basado en programación de escuchadores, emergieron una serie de arquitecturas que se recogen bajo el acrónico MV* por sus similitudes entre sí. Todas ellas comparten la premisa de que están centradas en un modelo interno cuyos cambios son atendidos por la vista a través de cierta lógica de control. En general este paraguas recoge diversas variantes arquitectónicas.



Model View Controller Jerárquico

Los modelos MVC se distribuyen entre el aplicativo para cada componente que se organizan agregativamente



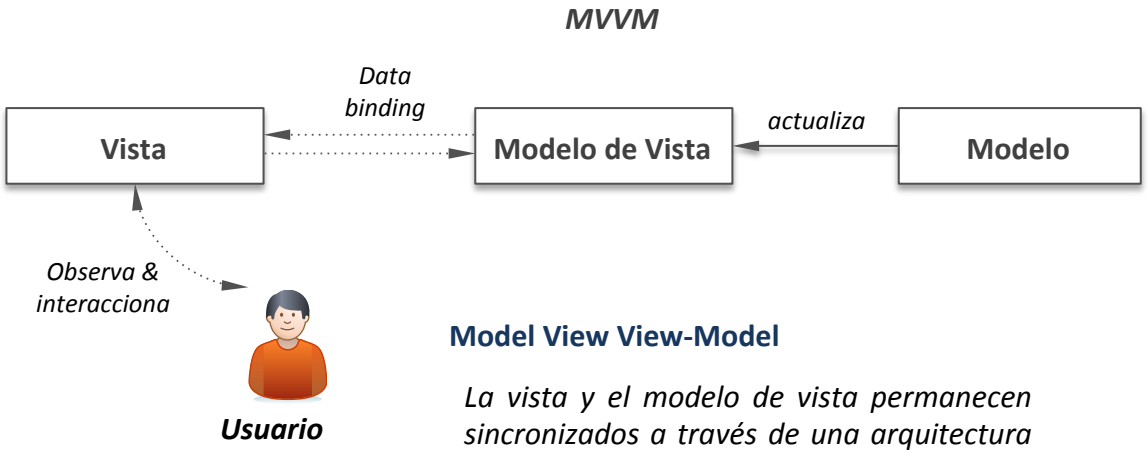
Model ViewPresenter

Como extensión del MVA, el patrón MVP hace que el controlador se mantenga a las escucha de los cambios en la vista y el modelo

I. El Camino Hacia las Arquitecturas Reactivas

B. Arquitecturas Centradas en el Modelo

Como evolución del modelo de construcción basado en programación de escuchadores, emergieron una serie de arquitecturas que se recogen bajo el acrónico MV* por sus similitudes entre sí. Todas ellas comparten la premisa de que están centradas en un modelo interno cuyos cambios son atendidos por la vista a través de cierta lógica de control. En general este paraguas recoge diversas variantes arquitectónicas.



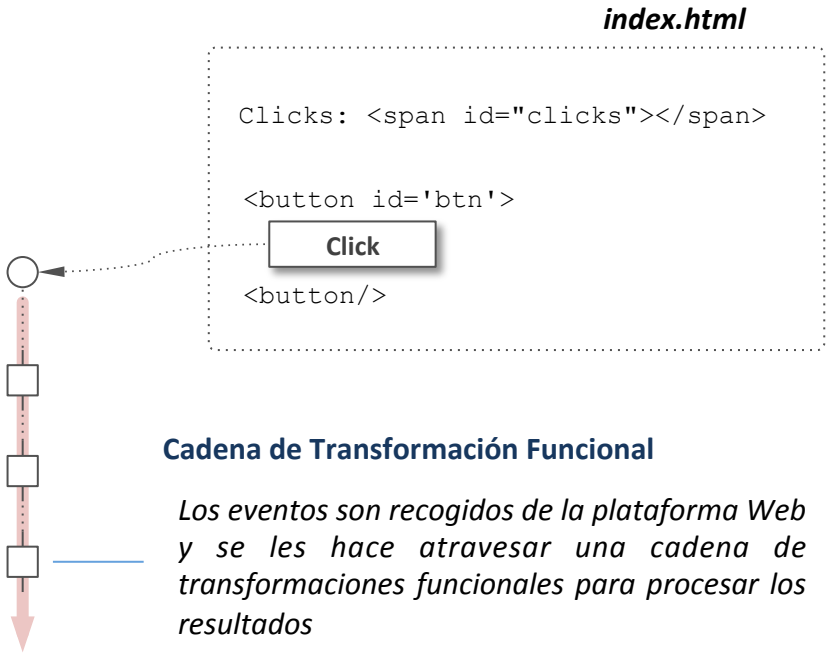
Model View View-Model

La vista y el modelo de vista permanecen sincronizados a través de una arquitectura de observadores y mutadores

I. El Camino Hacia las Arquitecturas Reactivas

C. Arquitecturas de Programación Funcional Reactiva

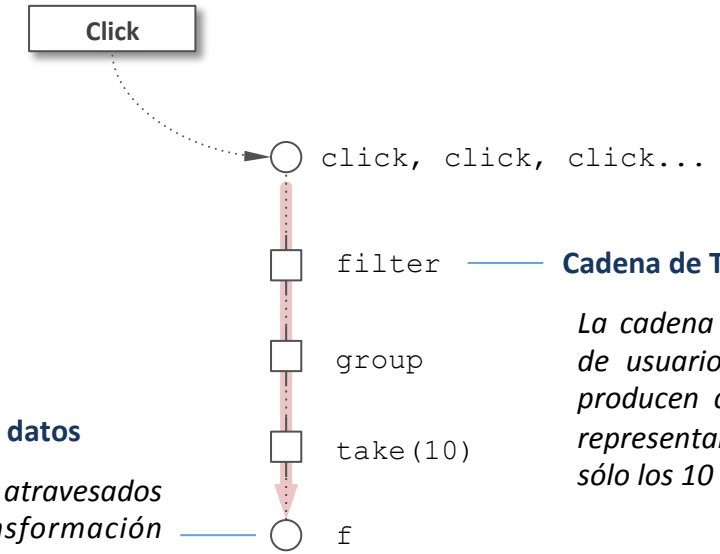
En contraposición a las propuestas anteriores que articulan su funcionamiento en base a un proceso de escucha y transformación de un modelo de datos interno más o menos explicito que representa el estado de la aplicación aparecen las arquitecturas de programación reactiva donde los eventos son gestionados por cadenas de transformación funcional que se encargan de procesar la interacción del usuario de una manera declarativa, inmutable y sin estado.



II. Principios de Programación Reactiva

A. Dirigida por los Datos

Las arquitecturas de programación funcional reactiva definen cadenas de transformación funcional que son atravesadas por flujos de datos para su procesamiento. Se dice que este tipo de soluciones está dirigida por los datos por que en ellos descansa gran parte de la lógica de la aplicación y el propio comportamiento reactivo del sistema.



Cadena de Transformación

La cadena de transformación filtra los eventos de usuario, los agrupa por pares si éstos se producen con cierta frecuencia temporal para representar el doble click y finalmente absorbe sólo los 10 primeros

Comportamiento dirigido por datos

La naturaleza de los datos atravesados por la cadena de transformación condiciona cuándo se ejecutará f

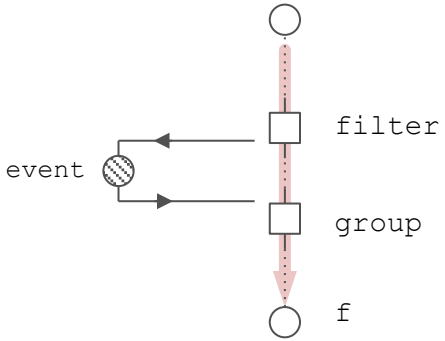
II. Principios de Programación Reactiva

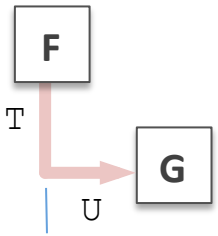
B. Centrada en la Composición

Las operaciones que forman parte del procesamiento de datos en programación reactiva se encadenan de forma compositiva de manera que el resultado de cada transformación es la entrada para la siguiente transformación. Para poder articular adecuadamente esta composición debemos tener en cuenta dos reglas de transformación funcional.

I. Dominio Simple

Dado que las funciones devuelven un único valor de retorno, el número de parámetros de entrada que puede aceptar una función en una composición debe ser exactamente uno. En el caso de la programación reactiva, la información se encapsula en una estructura de datos que representa un evento lo que permite cumplir esta condición





El tipo de salida T debe ser compatible con el tipo de entrada U. En términos OOP diríamos que T debe ser subtipo de U

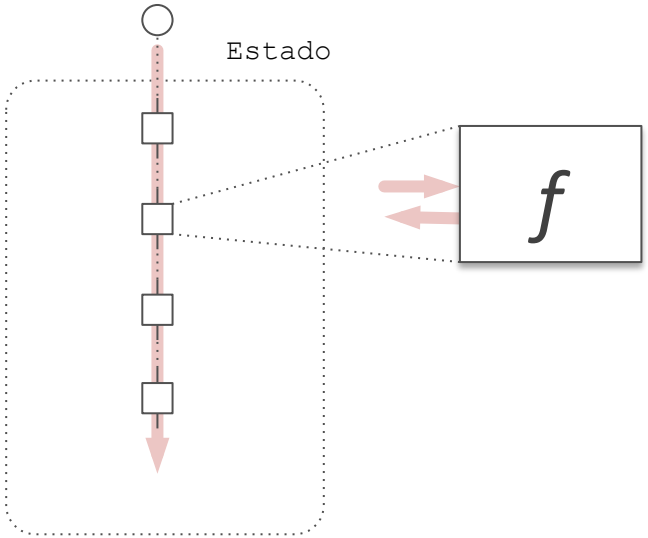
II. Compatibilidad Rango-Dominio

El tipo del parámetro de entrada de una función debe ser compatible con el tipo del valor de retorno devuelto por la función anterior para que la composición pueda articularse con éxito. Las arquitecturas reactivas también cumplen este requisito puesto que reciben eventos como entrada y generan eventos de salida

II. Principios de Programación Reactiva

C. Transparencia Referencial

El principio de transparencia referencial predica que en toda especificación funcional correcta, cualquier función debe poder substituirse en cualquier ámbito de aplicación por su expresión funcional sin que ello afecte al resultado obtenido. Equivalentemente, esto implica que el valor de retorno de las funciones, como cualquier transformación matemática, sólo puede depender de sus parámetros de entrada y no de condiciones ambientales.



Funciones Puras

Se dice que las operaciones de transformación son funciones puras en tanto que garantizan el mantenimiento de la transparencia referencial. Así, su valor de retorno sólo depende de los parámetros de entrada y no del estado ambiental (variables globales, variables retenidas en ámbito léxico, operaciones de E/S, etc.)

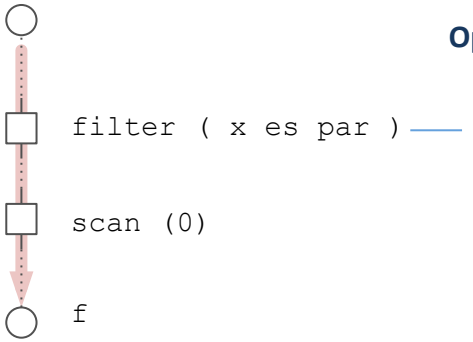
II. Principios de Programación Reactiva

D. Comportamiento Idempotente

Un principio en estrecha relación con el de transparencia referencial es de de comportamiento idempotente según el cual cada función de transformación debe devolver siempre el mismo resultado para los mismos parámetros de entrada. Esto permite razonar algebraicamente sobre la validez del procesamiento reactivo.

Operación No Idempotente

La operación scan cuenta el número de eventos que atraviesan la atraviesa y devuelve un evento con el valor de dicho contador en cada invocación. Se trata de una operación no idempotente porque el valor devuelto depende del número de invocaciones anteriores



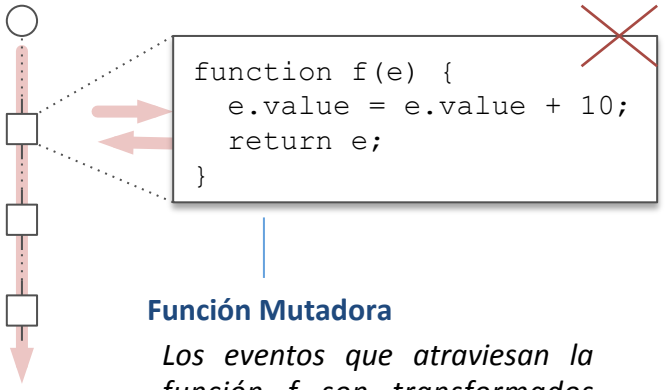
Operación Idempotente

La operación filter se encarga de dejar pasar a la cadena compositiva sólo aquellos eventos en los que x tiene un valor par. Se trata de una operación idempotente porque para el mismo evento de entrada siempre procede igual

II. Principios de Programación Reactiva

E. Inmutabilidad de Datos

Un último principio heredado de la programación funcional que debemos tener en cuenta en la construcción de arquitecturas reactivas es el principio de inmutabilidad de datos. En términos prácticos la aplicación se este principio se traduce en que las funciones nunca deben actualizar los parámetros de entrada como resultado de su ejecución sino solamente generar a partir de ellos resultados de salida.

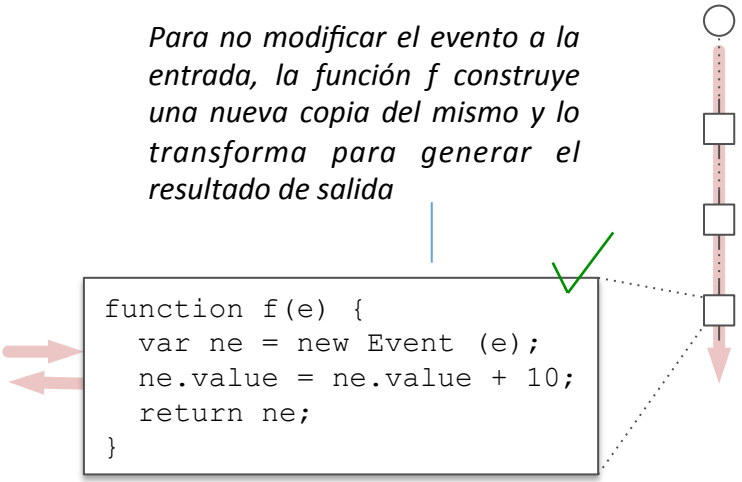


Función Mutadora

Los eventos que atraviesan la función f son transformados para generar un resultado de salida

Función No Mutadora

Para no modificar el evento a la entrada, la función f construye una nueva copia del mismo y lo transforma para generar el resultado de salida



Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

2 Conceptos Esenciales

- Arquitecturas Web Reactivas
- Visión Estática
- Visión Dinámica

I. Arquitecturas Web Reactivas

La programación funcional reactiva es un modelo de programación que pretende dar respuesta a la demanda creciente de nuevos sistemas más flexibles, elásticos y tolerantes a fallos a partir de los principios propios de la programación funcional. Esta aproximación resulta muy ventajosa en tanto que produce desarrollos más declarativos y explícitos centrados en las transformaciones que deben sufrir los datos para su procesamiento.

> *Las **Arquitectura Web Reactivas** son sistemas software que operan en reacción a los eventos ocurridos sobre el modelo de datos DOM de la Web. Cada evento atraviesa una cadena de transformación funcional para su procesamiento.*

I. Construcción Compositiva

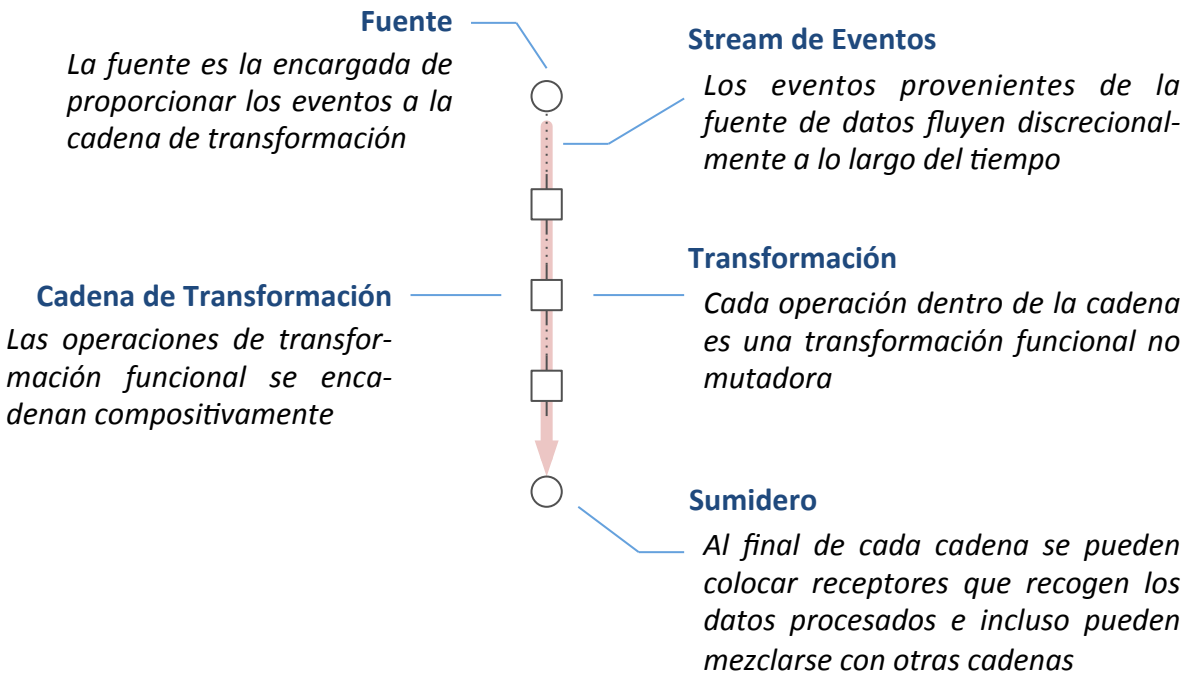
La lógica de negocio de una arquitectura reactiva queda capturada en la cadena de transformaciones que es atravesada por cada evento. Estas transformaciones son en general inmutables, idempotentes y agnósticas de las condiciones ambientales

II. Modelado Explicito del Tiempo

Paradójicamente, las operaciones de transformación que articulan las arquitecturas reactivas están en estrecha relación con el tiempo ya que éste es un factor determinante a la hora de construir modelos de comportamiento reactivo

II. Visión Estática

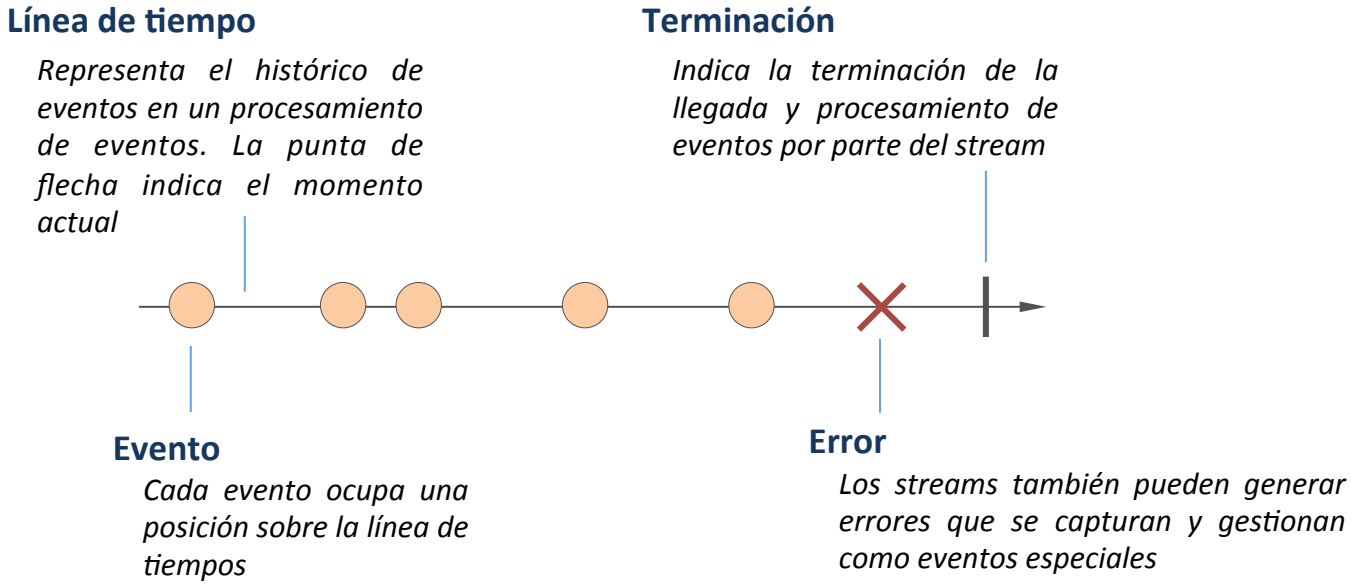
Desde un punto de vista estructural, las arquitecturas de programación funcional reactiva están caracterizadas por una colección de abstracciones funcionales. Estas abstracciones se encadenan secuencialmente para que lleven a cabo las tareas de procesamiento y transformación de eventos desde la fuente emisora a los sumideros receptores de manera que toda la lógica de negocio se expresa de forma declarativa y distribuida a lo largo de la cadena.



III. Visión Dinámica

A. Línea de Tiempo

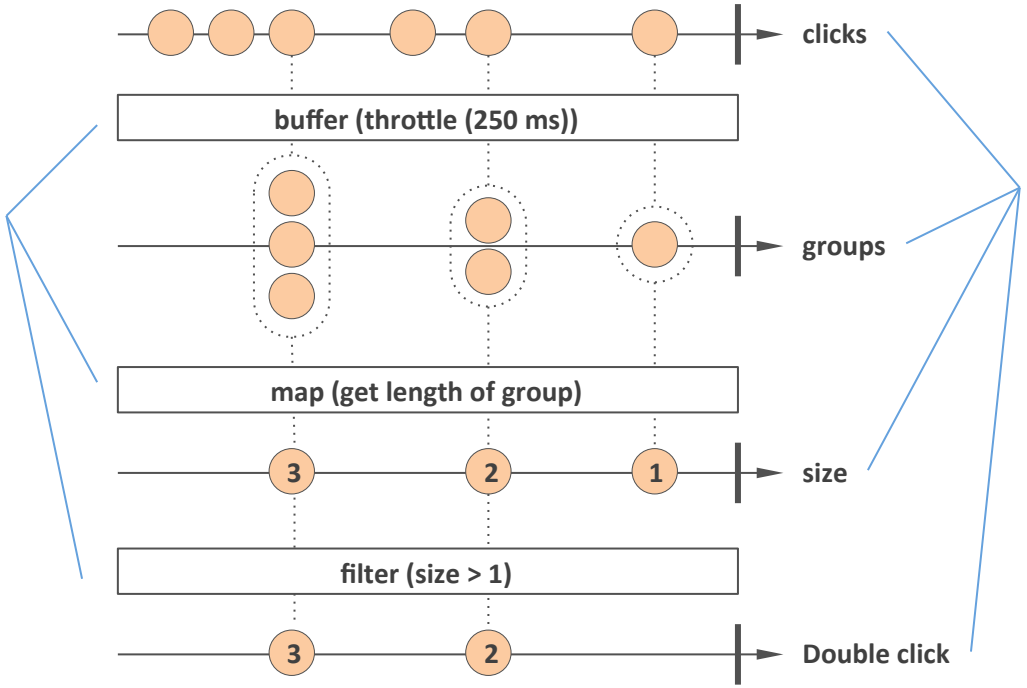
El comportamiento de las arquitecturas reactivas de programación funcional se encuentra en estrecha relación con el factor tiempo. En efecto, la lógica de las operaciones que forman parte de la cadena de transformación depende comúnmente de aspectos tales como el número de eventos procesados hasta el momento, su frecuencia de aparición o su periodo. Esto permite modelar fácilmente comportamientos sistémicos que dependen del tiempo.



III. Visión Dinámica

B. Diagramas de Reactividad

La mejor manera de razonar sobre las arquitectura reactivas es por medio del uso de diagramas de reactividad. Un diagrama de reactividad es una representación alterna de líneas de tiempo y operaciones de transformación funcional. Cada una de estas operaciones recibe como entrada la línea de tiempo anterior y genera como salida la línea subsiguiente.



Operaciones

Cada operación recibe una línea de tiempo como entrada y genera a su salída otra línea de tiempo con eventos nuevos

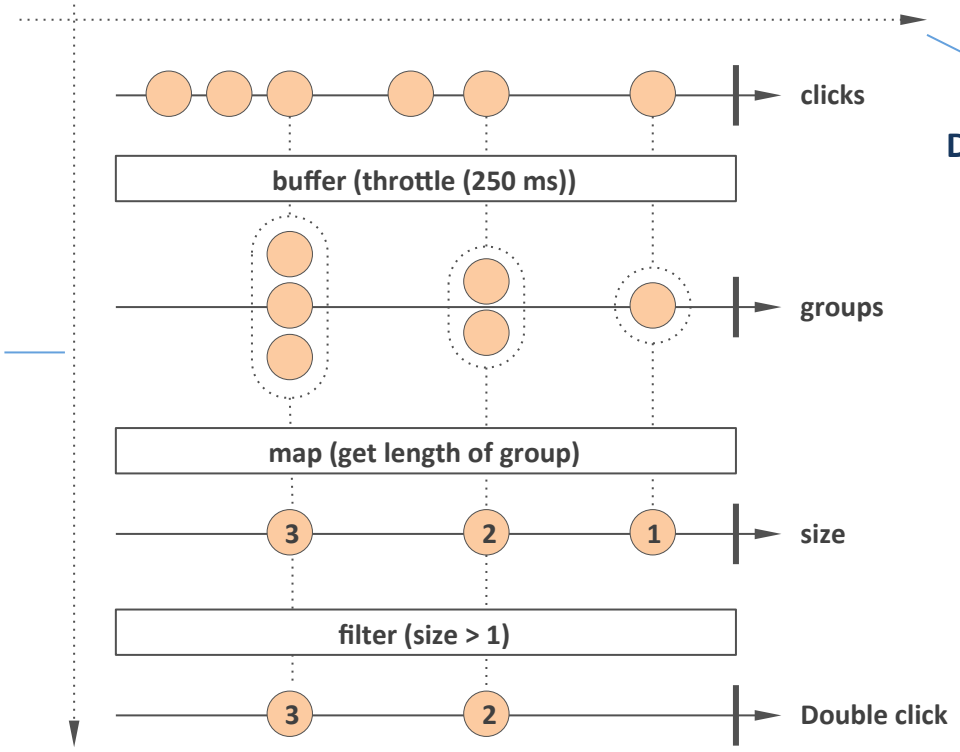
Líneas de Tiempo

Cada línea es una proyección a lo largo del tiempo de lo que ocurre en el sistema en un punto determinado de la cadena

III. Visión Dinámica

B. Diagramas de Reactividad

Los diagramas de reactividad ofrecen una visión del comportamiento de las arquitecturas reactivas en torno a dos perspectivas ortogonales. En cada vertical se puede comprobar cuál es el proceso de transformación que sufre un evento en un momento del tiempo. Horizontalmente, se contempla el comportamiento de cada fase a lo largo del tiempo



Dimensión Horizontal

Se razona acerca de cómo se comporta cada fase y el proceso independientemente de la dimensión tiempo

Dimensión Vertical

Se razona sobre lo que ocurre en un instante de tiempo con el procesa-miento de un evento concreto a lo largo de la cadena

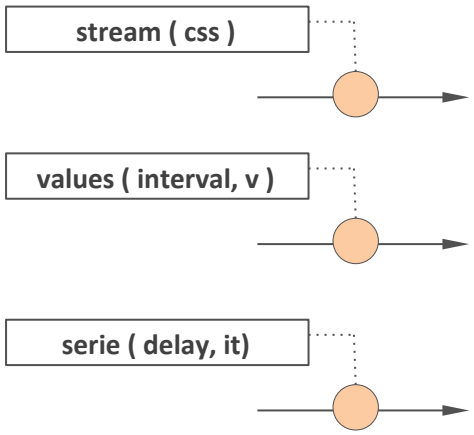
III. Visión Dinámica

C. Operaciones Reactivas

Para completar la descripción del comportamiento dinámico es necesario conocer las principales operaciones que pueden aplicarse para definir cadenas de transformación. El léxico de las mismas cambia de una librería a otra pero en esencia todas proporcionan el mismo juego de facilidades declarativas.

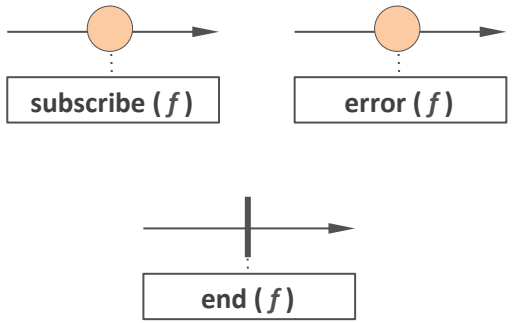
Operaciones de Creación

Las operaciones de creación permiten crear distintas fuentes de eventos en relación a distintos orígenes asíncronos de datos



Operaciones de Terminación

Las operaciones de terminación permiten definir sumideros al final de una cadena reactiva



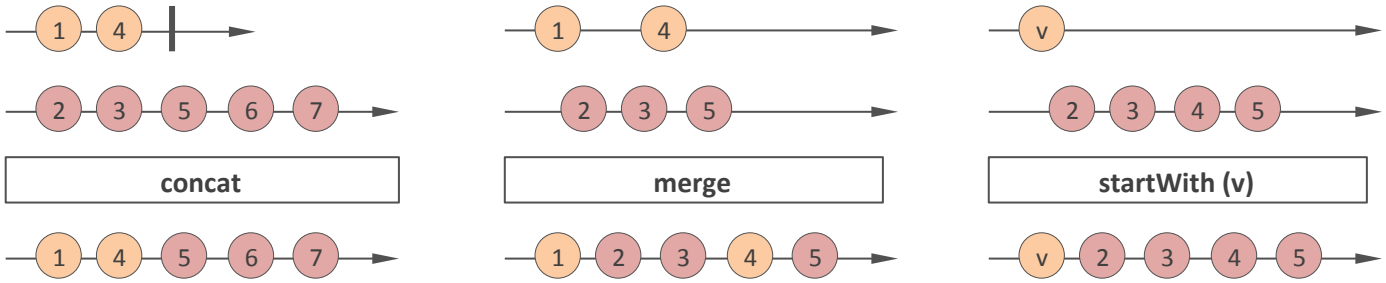
III. Visión Dinámica

C. Operaciones Reactivas

Para completar la descripción del comportamiento dinámico es necesario conocer las principales operaciones que pueden aplicarse para definir cadenas de transformación. El léxico de las mismas cambia de una librería a otra pero en esencia todas proporcionan el mismo juego de facilidades declarativas.

Operaciones de Mezcla de Streams

Las operaciones de entrelazado y mezcla de streams de eventos permiten fusionar varios streams de eventos para crear streams de eventos compuestos. Esto permite que un mismo stream pueda ser procesado por más de una cadena de composición de forma simultanea



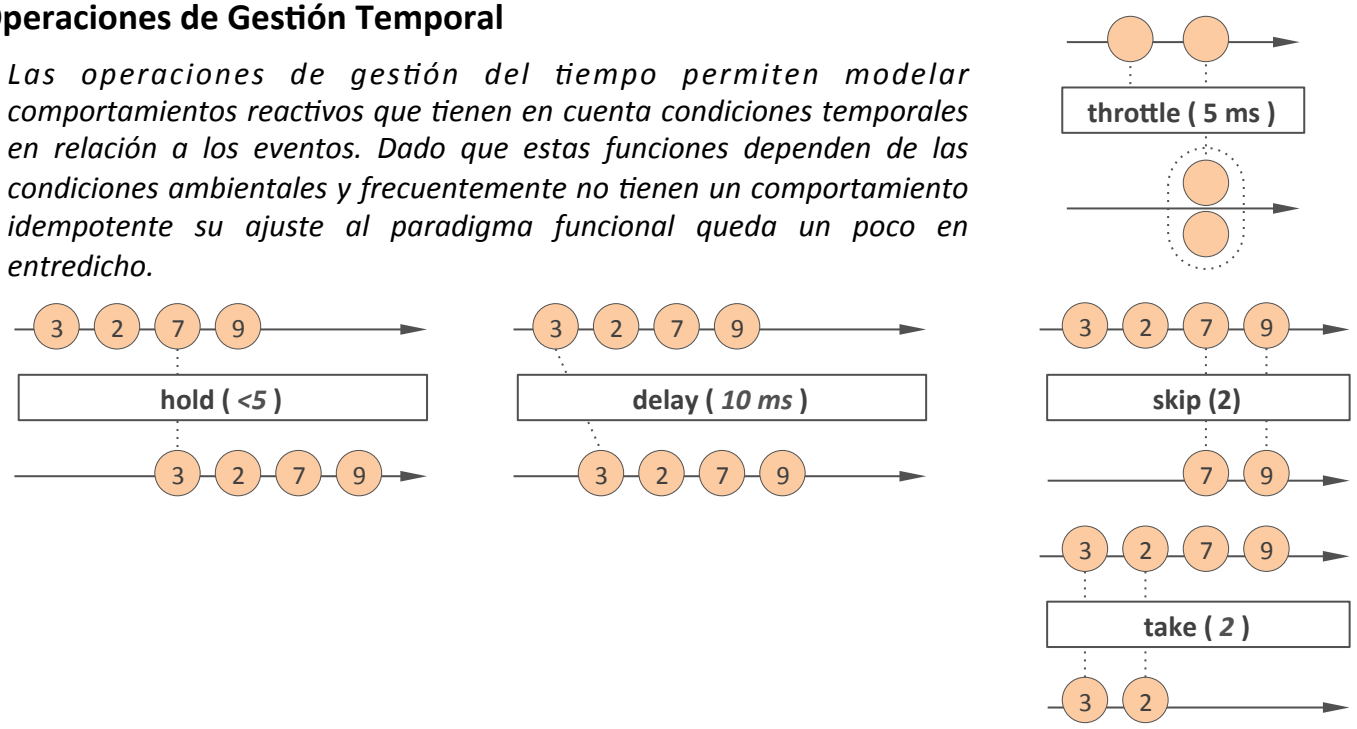
III. Visión Dinámica

C. Operaciones Reactivas

Del capítulo anterior ya ha quedado patente que las arquitecturas reactivas presentan una clara formulación funcional. Se trata en esencia de definir cadenas de transformación que serán atravesadas por los datos según van generándose. Sin embargo en términos más prácticos existen una serie de carácterísticas diferenciales que resumimos a continuación.

Operaciones de Gestión Temporal

Las operaciones de gestión del tiempo permiten modelar comportamientos reactivos que tienen en cuenta condiciones temporales en relación a los eventos. Dado que estas funciones dependen de las condiciones ambientales y frecuentemente no tienen un comportamiento idempotente su ajuste al paradigma funcional queda un poco en entredicho.



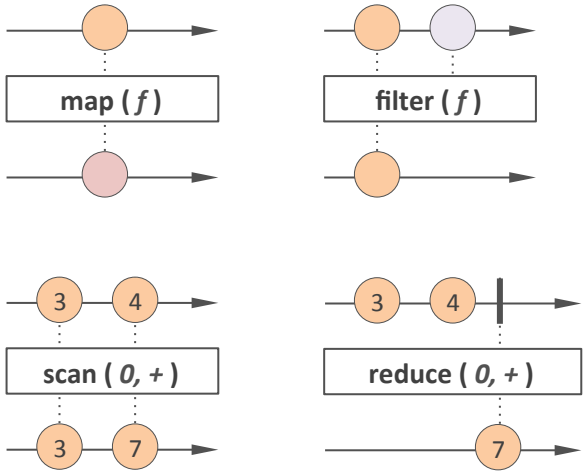
III. Visión Dinámica

C. Operaciones Reactivas

Del capítulo anterior ya ha quedado patente que las arquitecturas reactivas presentan una clara formulación funcional. Se trata en esencia de definir cadenas de transformación que serán atravesadas por los datos según van generándose. Sin embargo en términos más prácticos existen una serie de carácterísticas diferenciales que resumimos a continuación.

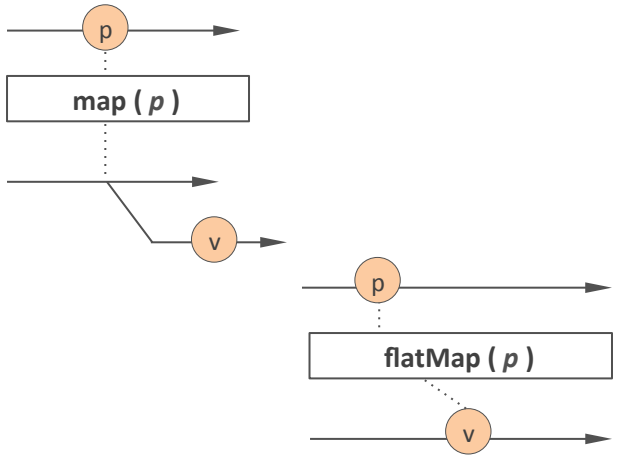
Operaciones de Secuenciamiento

Las operaciones clásicas de secuenciamiento (map, reduce, filter) se adaptan al modelo unitario de procesamiento reactivo



Operaciones de Asincronía

Las operaciones asíncronas permiten gestionar distintos aspectos relacionados con el carácter asíncrono de los eventos



Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

3

Un Framework Sencillo de Programación Reactiva

- Modelos de Programación Reactiva
- Framework Sencillo de Programación Reactiva
- Evaluación

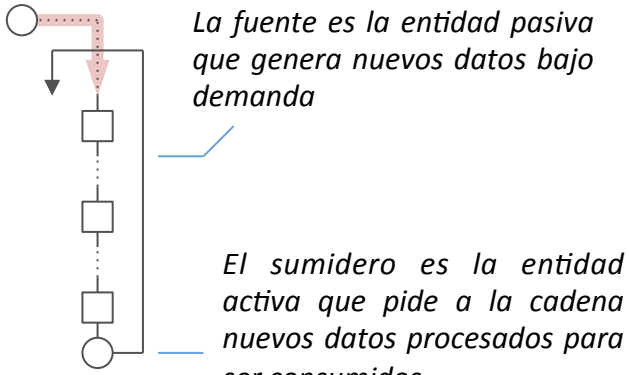
I. Modelos de Programación Reactiva

A. Modelos de Operación Pull & Push

Es posible caracterizar las arquitecturas de programación funcional reactiva en virtud de la naturaleza síncrona o asíncrona de los datos que procesan. Ello da lugar a sendos modelos de operación conocidos con el nombre de Pull y Push respectivamente. En el primer caso, es el sumidero el que pide nuevos datos a la cadena mientas que en el segundo es la fuente la encargada de empujar dichos datos proactivamente en cuanto éstos son generados.

I. Modelo Pull

En el modelo pull, el sumidero solicita nuevos datos y la solicitud recorre ascendentemente la cadena hasta llegar a la fuente

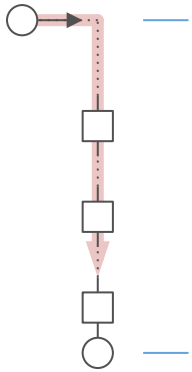


La fuente es la entidad pasiva que genera nuevos datos bajo demanda

El sumidero es la entidad activa que pide a la cadena nuevos datos procesados para ser consumidos

I. Modelo Push

En el modelo push, a medida que se disponen de nuevos datos, la fuente los empuja por la cadena de transformación hacia abajo



La fuente es la entidad activa que produce datos y los entrega a la cadena

El sumidero es una entidad pasiva que se mantiene a la escucha para consumir datos cuando llegan desde la cadena

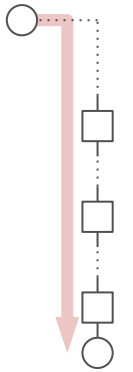
I. Modelos de Programación Reactiva

B. Modelos de Computación Continua & Discreta

Ortogonalmente, las fuentes también pueden clasificarse en virtud de la naturaleza continua o discreta de los datos que procesan. En el primer caso, hablamos de modelos de streamimg donde los datos son señales continuas y perdurables mientras que en el segundo caso, cada dato representa una ocurrencia puntual en el tiempo y ello conforma arquitecturas dirigidas por eventos.

I. Modelo Continuo

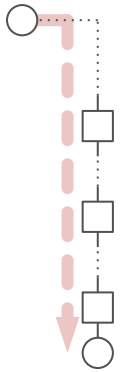
En el modelo de computación continua los datos se corresponden con streams llamados señales o comportamientos



Los sistemas reaccionan a los cambios en el flujo de datos. A este cambio se le suele llamar switching. En este tipo de sistemas la frecuencia de muestreo es uno de los parámetros característicos

I. Modelo Discreto

En el modelo discreto los datos se corresponden con eventos que se producen en instantes puntuales del tiempo

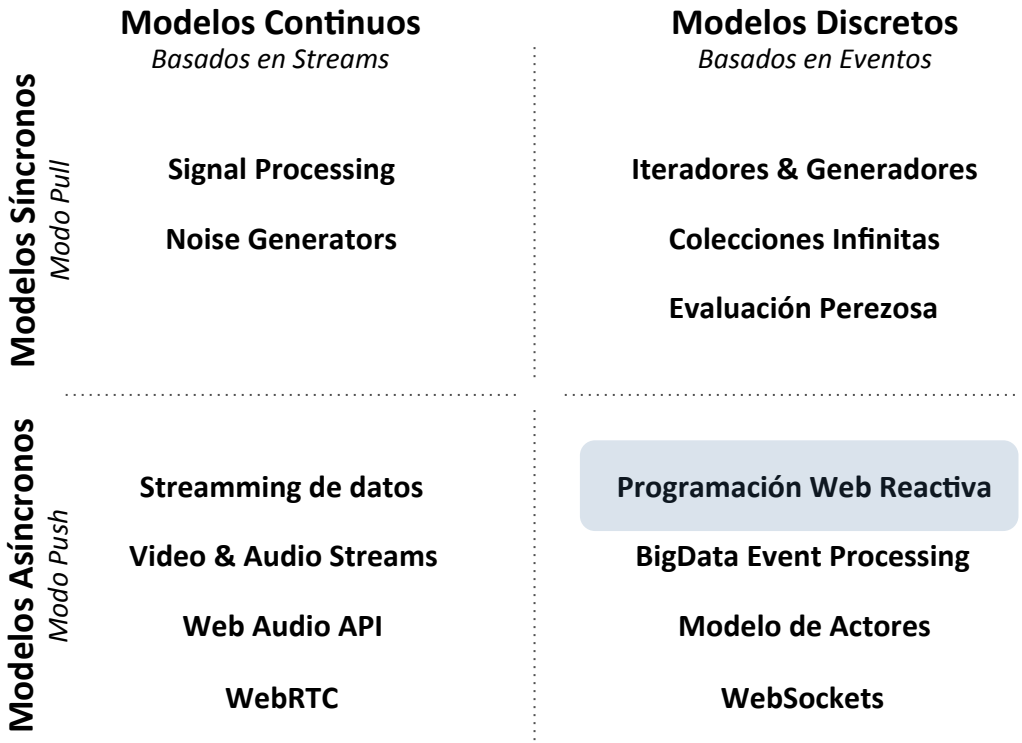


Los eventos son elementos vehiculares que acarrean valores de contexto transformados a través de la cadena reactiva

I. Modelos de Programación Reactiva

C. Clasificación

Es posible establecer las dos caracterizaciones anteriores como dos ejes dimensionales dispuestos ortogonalmente. Ello da lugar a un espacio de clasificación que divide las arquitecturas reactivas en cuatro familias. Nosotros centraremos nuestra atención en arquitecturas Web reactivas.



II. Framework Sencillo de Programación Reactiva

A. Generación de Fuentes

Como hemos visto en el capítulo anterior existe una gran variedad de constructores que permiten definir fuentes asíncronas que funcionan en modo Push. Aquí nos centraremos en presentar el código interno de las más comúnmente utilizadas.

Adaptador de Evento Web

Los adaptadores de eventos son funciones que recogen eventos Web y los redirigen a una función manejadora pasada como argumento en una segunda fase de evaluación

En este ejemplo se ve cómo la función fromTarget construye una fuente s a partir de un selector css y un tipo de evento

```
<button id="btn">click</button>
 ...
<script>
   var s = fromTarget('#btn', 'click');
   s(console.log);
</script>
```

```
function fromTarget(css, type) {
  var target = document.querySelector (css);
  return function (fn) {
    target.addEventListener(type, function (e) {
       fn(e);
    });
  };
}
function fromTargetAll(css, type) {
  var targets = document.querySelectorAll (css);
  return function (fn) {
    [].forEach.call(targets, function (target) {
       target.addEventListener(type, function (e) {
         fn(e);
       });
    });
  };
}
```

II. Framework Sencillo de Programación Reactiva

A. Generación de Fuentes

Como hemos visto en el capítulo anterior existe una gran variedad de constructores que permiten definir fuentes asíncronas que funcionan en modo Push. Aquí nos centraremos en presentar el código interno de las más comúnmente utilizadas.

Adaptador de Colección

Los adaptadores de colección persiguen generar fuentes push que iteran circularmente sobre los elementos de una colección a intervalos regulares de tiempo

La función fromValues recibe un array de valores y un intervalo de tiempo y construye una función que emitirá un evento con el valor siguiente en la colección cada vez que es invocado

```
function fromValues (values, ms) {
  return function (fn) {
    var idx = 0;
    var hn  = fn || function () {};
    setInterval (function () {
      fn(values[idx]);
      idx = (idx + 1) % values.length;
    }, ms || 1000);
  };
}
```

```
var s = fromValues([1,2,3]);
s(console.log);
```

En este sencillo ejemplo, la construcción de la fuente asíncrona de eventos emite circularmente, a intervalos regulares de 1 segundo, los elementos del array pasado como parámetro de entrada. Los eventos se vinculan al escuchador de salida estándar

II. Framework Sencillo de Programación Reactiva

A. Generación de Fuentes

Como hemos visto en el capítulo anterior existe una gran variedad de constructores que permiten definir fuentes asíncronas que funcionan en modo Push. Aquí nos centraremos en presentar el código interno de las más comúnmente utilizadas.

Adaptador de Iteración

La función serie toma una función genérica fn y devuelve otra función iteradora que genera valores consecutivos aplicando dicha función

fromSerie es una fuente push que invoca la serie pasada como parámetro a intervalos regulares de tiempo. Esta es una manera de hacer un adaptador Push sobre una fuente Pull

El código de ejemplo define la función inc y sobre ella genera la serie de números naturales num. Despues construye una fuente y la vincula a la salida estándar

```
var inc = function (x) { return x + 1; }
var num = serie(inc, 0);
var s = fromSerie(num, 3000);
s(console.log);
```

```
function serie (fn, base) {
  var value = base;
  return function () {
    value = fn(value);
    return value;
  };
}
function fromSerie (serie, ms) {
  return function (fn) {
    var hn = fn || function () {};
    setInterval (function () {
      var value = serie();
      fn(value);
    }, ms || 1000);
  };
}
```

II. Framework Sencillo de Programación Reactiva

B. Operadores de Transformación

Hasta ahora damos soporte a la definición de fuentes de diferentes tipos que se conectan en modo Push a sumideros directamente. Definamos a continuación, la colección de operadores de transformación que podrán ser utilizados para definir cadenas en nuestro framework.

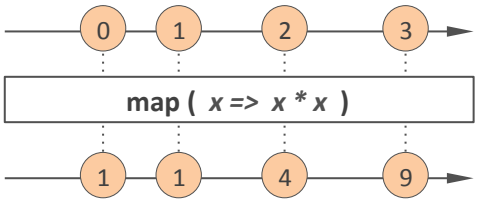
Operador map

El operador map es una función de orden superior que recibe como parámetro una función fn genérica y devuelve otra función como resultado. El comportamiento de dicha función es invocar a fn con los argumentos pasados como parámetro en esta función de retorno

```
function map (fn) {
  return function () {
    return fn.apply(this, arguments);
  };
}
```

El siguiente código de ejemplo muestra un uso prototípico de la función map que transforma cada dato de entrada en su cuadrado. Para ello se aplica map con una función anónima que define el cuadrado y luego se aplica la función resultante

```
var sqr = map(function (x) {
  return x * x;
});
sqr(3); // 9
```



II. Framework Sencillo de Programación Reactiva

B. Operadores de Transformación

Hasta ahora damos soporte a la definición de fuentes de diferentes tipos que se conectan en modo Push a sumideros directamente. Definamos a continuación, la colección de operadores de transformación que podrán ser utilizados para definir cadenas en nuestro framework.

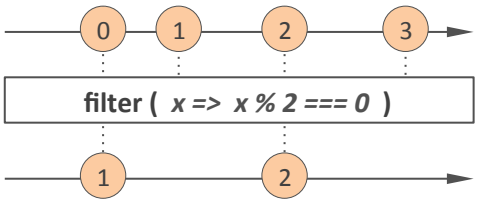
Operador filter

El operador filter recibe un predicado lógico como parámetro y genera otra función como retorno cuyo propósito es filtrar sólo aquellos argumentos que satisfagan dicho predicado. Si el filtro no es pasado entonces la función devuelve undefined

En el ejemplo siguiente se aplica el operador filter sobre un predicado lógico que determina si un valor es par. El resultado es una función que sólo deja filtrar los eventos con valor par.

```
var even = filter(function (x) {
  return x % 2 === 0;
});
even(2); // 2
even(3); // undefined
```

```
function filter (fn) {
  return function () {
    var out = fn.apply (this, arguments);
    if (out) return arguments[0];
  };
}
```



II. Framework Sencillo de Programación Reactiva

B. Operadores de Transformación

Hasta ahora damos soporte a la definición de fuentes de diferentes tipos que se conectan en modo Push a sumideros directamente. Definamos a continuación, la colección de operadores de transformación que podrán ser utilizados para definir cadenas en nuestro framework.

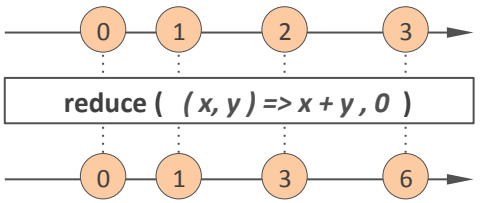
Operador scan

El operador scan es un operador con memoria. Cada nuevo dato lo combina con un acumulador retenido en ac a través de una función binaria de reducción pasada como parámetro. El valor inicial del acumulador se pasa como segundo parámetro b de la función

En el ejemplo adjunto se aplica el operador scan sobre la función reductora de la suma y con base en 0. Como resultado, cada evento devuelve la suma acumulada de los valores en los eventos procesados hasta el momento

```
var add = scan(function (x, y) {
  return x + y;
}, 0);
add(2); // 2 = 2 + 0
add(3); // 5 = 3 + 2
add(4); // 9 = 4 + 5
```

```
function scan (fn, b) {
  var ac = b;
  return function () {
    var args = [].slice.call (arguments);
    ac = fn.apply (this, [ac].concat(args));
    return ac;
  };
}
```



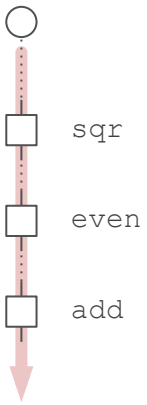
II. Framework Sencillo de Programación Reactiva

C. Encadenamiento de Operadores

Nuestros operadores aplican funciones de transformación y devuelven el resultado de dicha aplicación. Como tal no pueden encadenarse de acuerdo a una sintaxis de API fluida. Es necesario definir una función que adapte cada función adecuadamente en este sentido.

Se pretende transformar de forma transparente las funciones map, filter y scan para que puedan ser encadenadas en notación de punto a modo de API fluida

```
var numbers = fromValues([2,3,4])
    .map(sqr)
    .filter(even)
    .scan(add, 0)
    .end();
numbers(console.log);
```



El orden en que se incluyen las transformaciones sqr, even y add en la expresión de API fluida corresponde al orden en que éstas son añadidas en la cadena reactiva

```
function fluent (hn) {
   var cb = hn || function () {};
   return function (fn) {
     return function () {
        cb(fn.apply(this, arguments));
        return this;
     };
  };
}
```

Necesitamos transformar los operadores a través de la función fluid que genera una función para ejecutar el operador, entregar el resultado a un manejador y devolver una referencia al objeto de contexto sobre el que se aplica el operador punto para poder seguir encadenando operadores

II. Framework Sencillo de Programación Reactiva

D. Composición de Transformaciones

Una vez logrado el objetivo de encadenar operadores y antes de cerrar nuestro framework es necesario definir una función de secuenciamiento que aplique compositivamente cada función de transformación dentro de la cadena. Veamos como hacerlo.

La función sequence recorre el array de funciones pasado como parámetro y lo aplica compositivamente a partir del parámetro x. Como se ve la implementación hace uso de una estrategia de reducción con base en x y garantiza que en todo momento cada función aplicada devuelve un resultado no nulo

```
function sequence (fns) {
  return function (x) {
    return fns.reduce (function (ac, fn) {
      return (ac !== void 0) ?
         fn(ac) :
         void 0;
    }, x);
  };
}
```

```
var s = sequence([
  map(sqr),
  filter(even),
  scan(add, 0)
]);
s(3) // undefined (even(9) === false)
S(2) // 4
```

Para entender mejor como opera esta función pensemos en un ejemplo con el array de funciones [sqr, even, add]. Pretendemos construir una función que cuando se invoque sobre x devuelva el resultado compositivo equivalente a evaluar la expresión add (even (sqr (x)))

II. Framework Sencillo de Programación Reactiva

E. Constructor Stream

Después de todo este trabajo sólo resta definir el constructor Stream que proporciona el objeto de contexto donde se definirá la cadena de composición, incluir las funciones anteriores y añadir alguna lógica adicional para soportar el proceso de registro de escuchadores.

```
function Stream (source) {
    var fns = [];
    var lns = [];
    var define = fluent (function (fn) {
         fns.push (fn);
    });
    return {
        map   : define (map),
        filter: define (filter),
        scan  : define (scan),
        end: function () {
          var seq = sequence(fns);
          source(function (data) {
             var result = seq(data);
             lns.forEach(function (ln) {
                   if (result) ln (result);
             });
          });
          return {
             listen: function (ln) {
                lns.push(ln);
             }
          };
       }
    };
}
```

Patrón Stream Push

La colección lns acumula los escuchadores que se registran al stream para ser notificados de cada nuevo resultado procesado

La función end aplica sequence una vez para obtener la cadena de composición y después arranca la fuente con un manejador que obtiene el siguiente resultado y si es definido lo publica a cada escuchador

Como retorno se devuelve un objeto con un método listen que permite registrar nuevos manejadores asociados a escuchadores interesados

III. Evaluación

Antes de terminar repasaremos un par de ejemplos sencillos que pueden ser desarrollados con nuestro framework de forma declarativa. Para cada ejemplo incluiremos a efectos comparativos la versión tradicional basada en eventos Web para apreciar sus diferencias.

Suma de Cuadrados Pares

La serie de datos genera un iterador de naturales que se emplea como fuente push para una cadena de trasformadores [sqr, event, add]. Tras finalizar la declaración de la cadena, se invoca el método listen para registrar a la consola como único escuchador

Reactivo

```
var data = serie (function (x) { return x + 1; }, 0);
var source = fromSerie(data);
var stream = Stream (source)
    .map    (function (e)    { return e * e;     })
    .filter (function (e)    { return e % 2 === 0})
    .scan   (function (a, e) { return a + e;     }, 0)
    .end ();
stream.listen (console.log);
```

Clásico

```
var idx = 0;
var sqr = function (n) {
    var sqrn = n * n;
    if (sqrn % 2 === 0) console.log (sqrn);
    idx++;
    setTimeout(function () { sqr(n+1); }, 1000);
};
sqr(0);
```

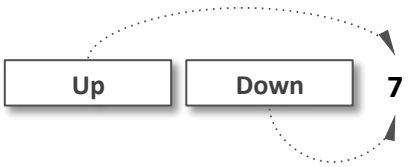
La versión imperativa de este problema es considerablemente menos declarativa que la forma reactiva. Además se trata de una solución con estado donde el estado no está encapsulado sino mantenido por una variable externa idx.

III. Evaluación

Antes de terminar repasaremos un par de ejemplos sencillos que pueden ser desarrollados con nuestro framework de forma declarativa. Para cada ejemplo incluiremos a efectos comparativos la versión tradicional basada en eventos Web para apreciar sus diferencias.

Contador de Clicks

En el enfoque clásico se utilizan variables para referir los elementos (display & btns) y para mantener el estado (count) así como estructuras de control de flujo para articular el proceso de registro de escuchadores



Clásico

```
<button id="up" class="btn">Up</button>
<button id="down" class="btn">Down</button>
<span id="display"></span>

<script>
var count = 0;
var display = document.querySelector('#display');
var btns = document.querySelectorAll('.btn');
var register = function (idx) {
  btns[idx].addEventListener ('click', function (e) {
    if (e.currentTarget.id === 'up') count++;
    if (e.currentTarget.id === 'down') count--;
    display.innerHTML = count;
  });
};
for (var idx=0; idx < btns.length; idx++)
  register(idx);
</script>
```

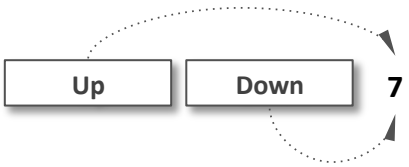
III. Evaluación

Antes de terminar repasaremos un par de ejemplos sencillos que pueden ser desarrollados con nuestro framework de forma declarativa. Para cada ejemplo incluiremos a efectos comparativos la versión tradicional basada en eventos Web para apreciar sus diferencias.

Contador de Clicks

Reactivo

```
var source = fromTargetAll('.btn', 'click');
var stream = Stream (source)
  .map  (function (e)    { return e.currentTarget.id === 'up' ? 1 : -1 })
  .scan (function (a, e) { return a + e; }, 0)
  .end ();
stream.listen (function (n) {
  document.querySelector('#display').innerHTML = n;
});
```



La versión reactiva resulta mucho más declarativa y limpia. Primero se crea una fuente adaptadora de eventos click sobre el target .btn y después se define una cadena formada por dos operaciones, la primera identifica el paso de incremento que se debe aplicar a la suma y la segunda realiza la suma. Aquí no existe contador explícito

Arquitecturas Reactivas de Streams

Preguntas

Reactividad

Composición

Stream

Eventos

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Arquitecturas de Frontend

Arquitecturas Reactivas de Streams

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Octubre 2014