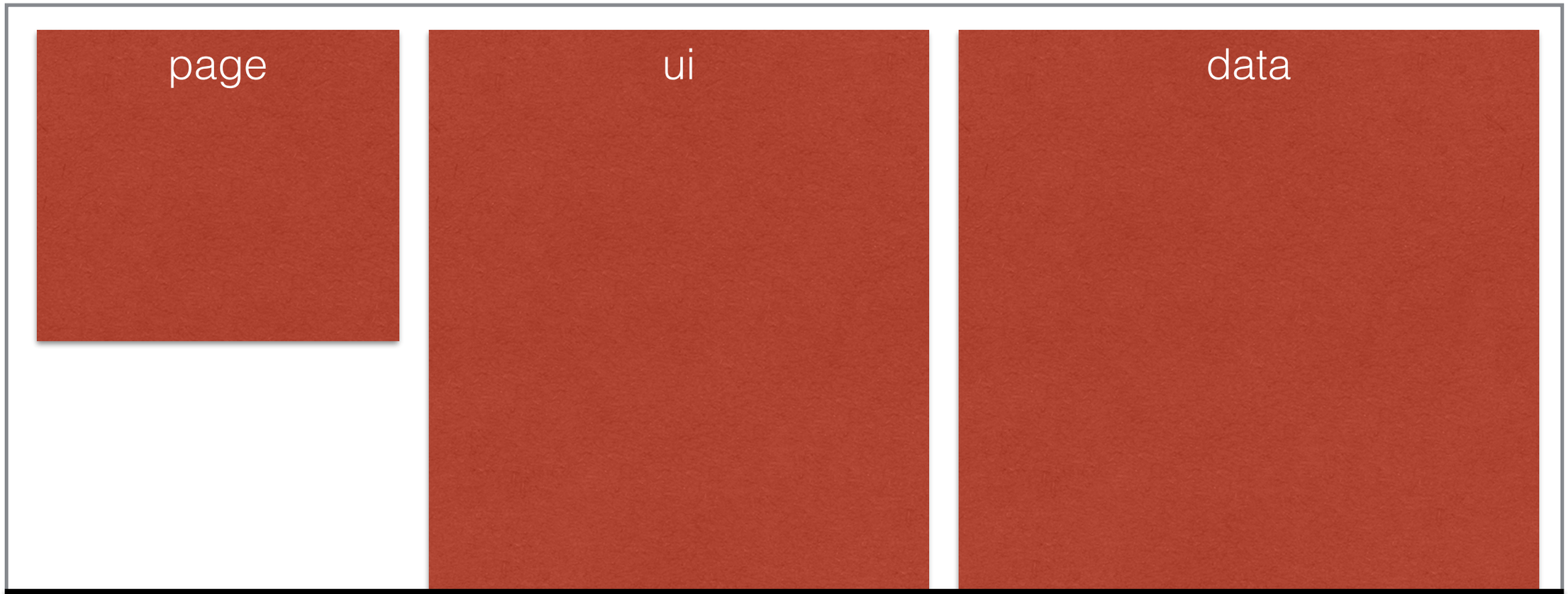
Contenido

- 2. Átomo: gestión de estado global
  - Un ejemplo
  - Inmutabilidad
  - Estructuras inmutables: Map, Set, Vector
  - Secuencias y operaciones con secuencias
  - Átomo: qué, para qué, cómo

Átomo: un ejemplo

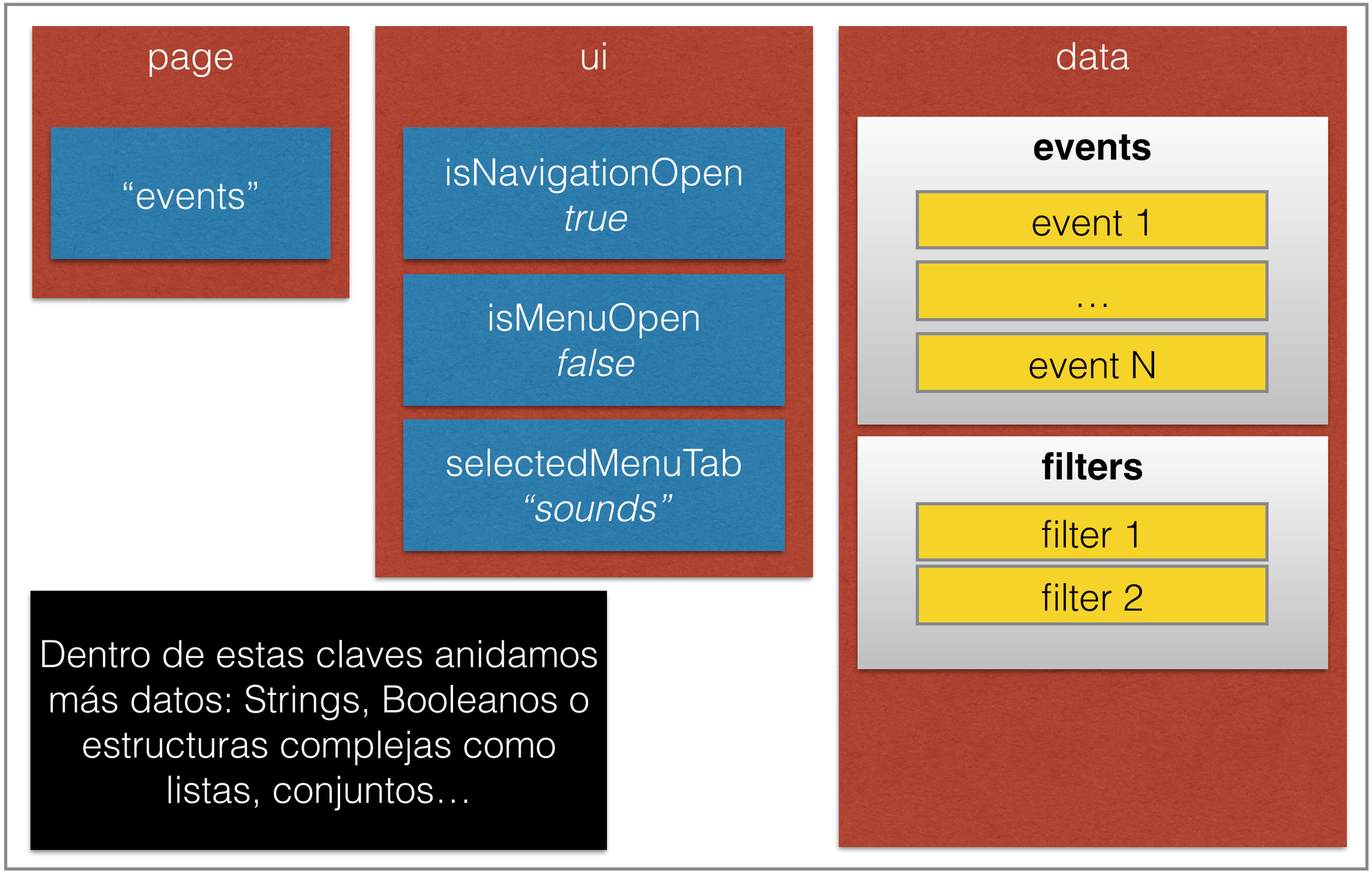


El átomo es un mapa (tabla hash) estructurado en forma de árbol con los nombres de clave que queramos.

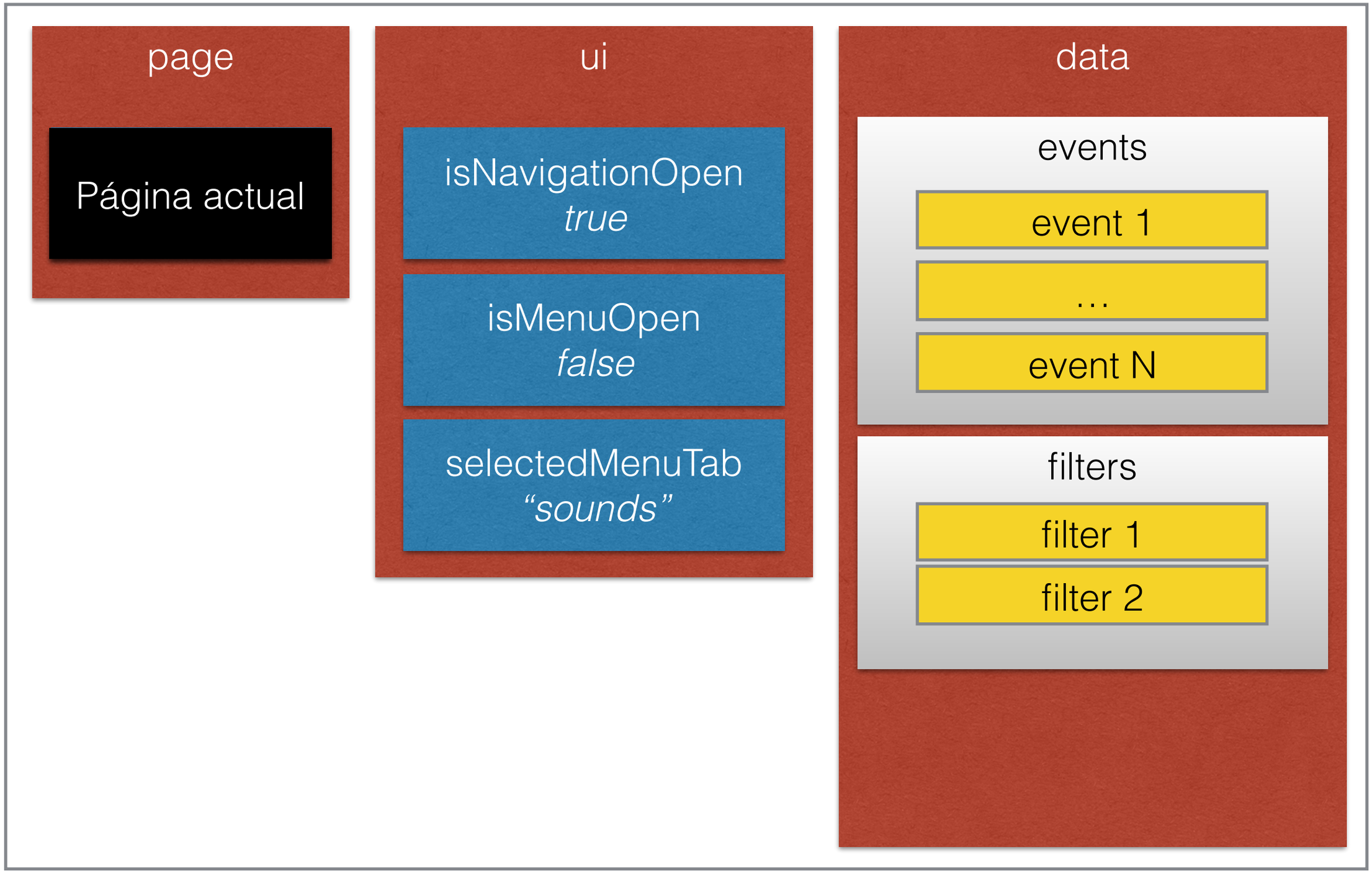
Habitualmente, tenemos *estado de la UI* y *datos de la aplicación.*

La clave *page* es especial: define el layout entero de la aplicación en función de la página (“pantalla”) actual

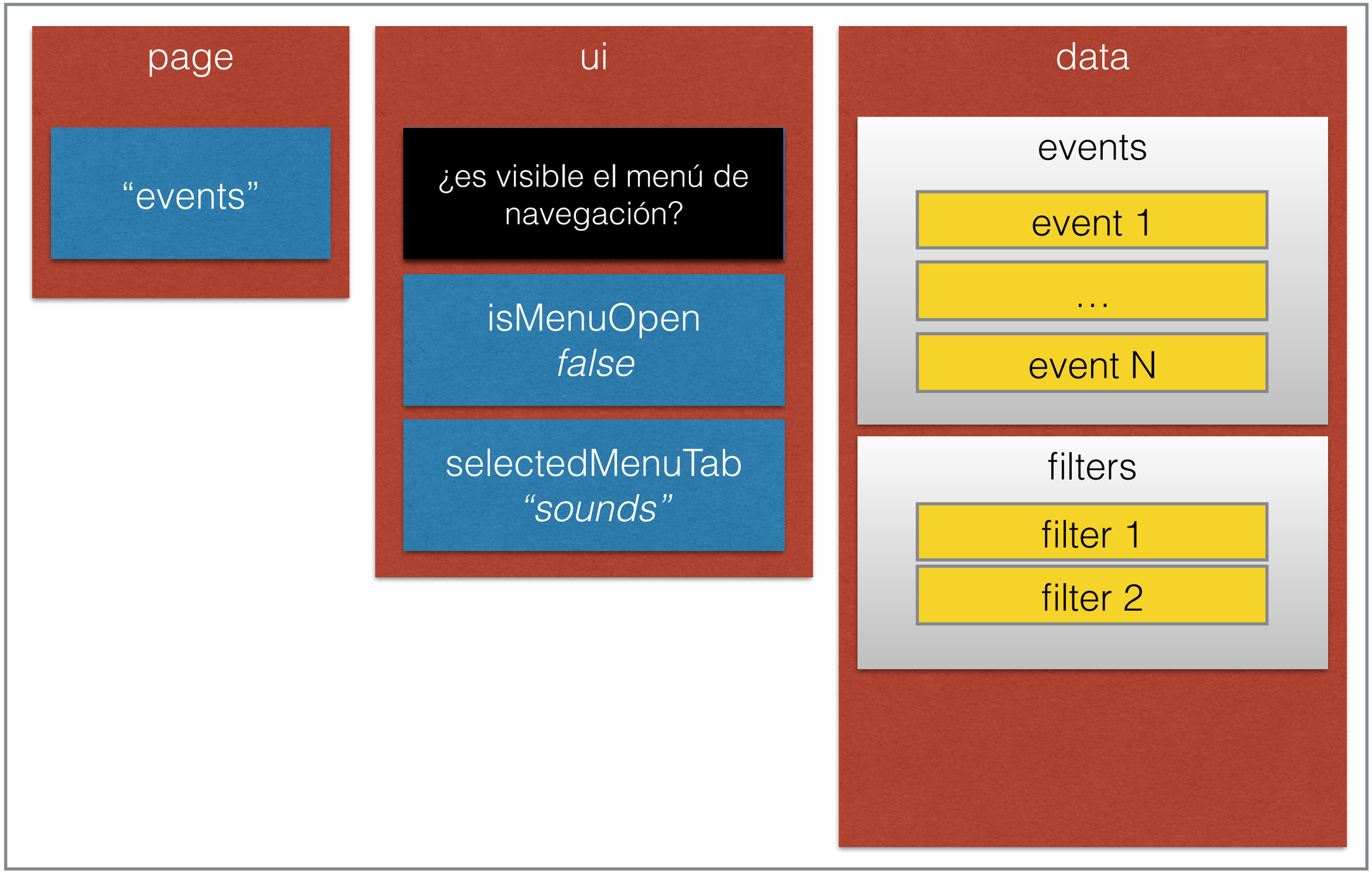
Átomo: un ejemplo



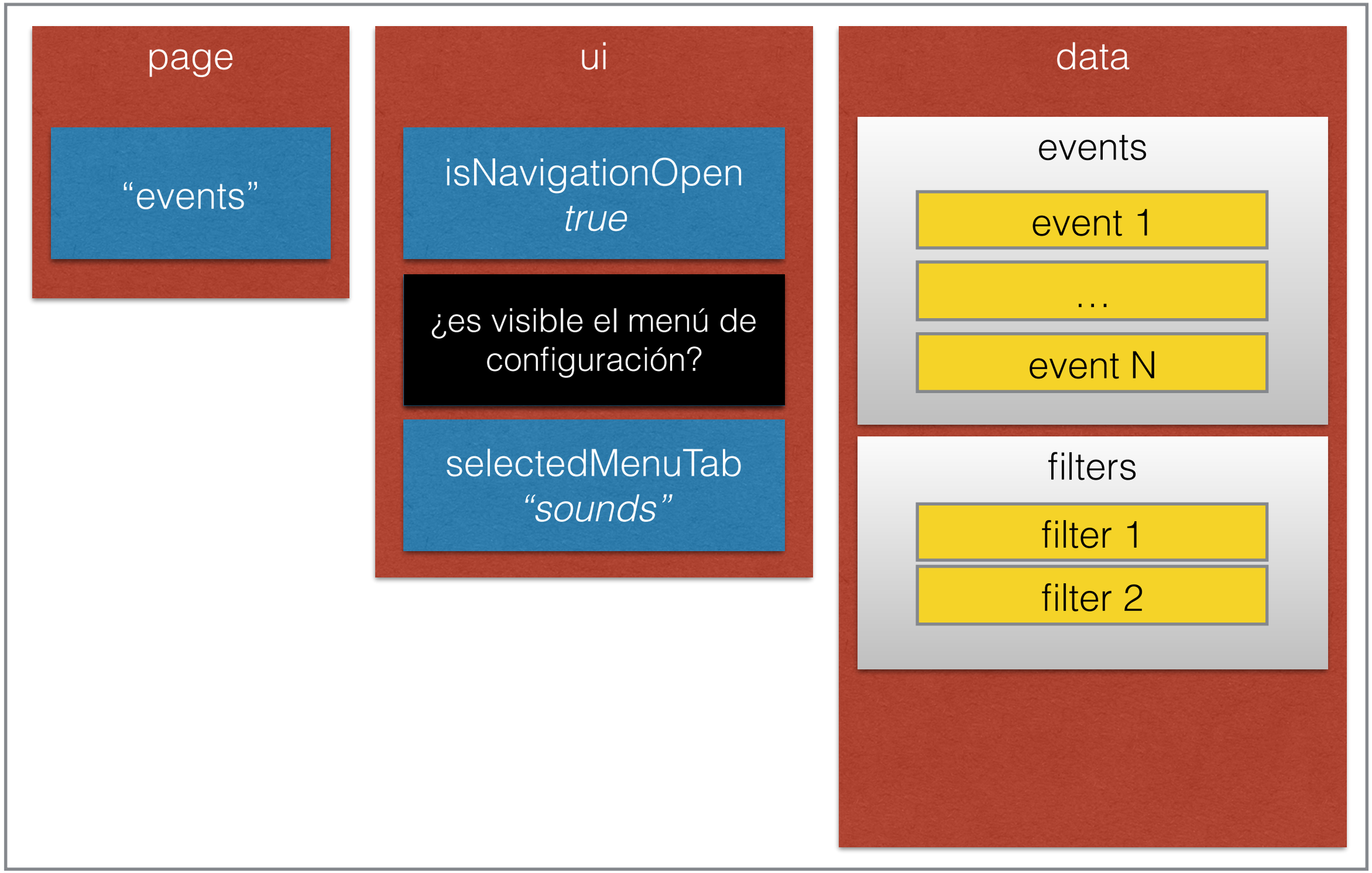
Átomo: un ejemplo



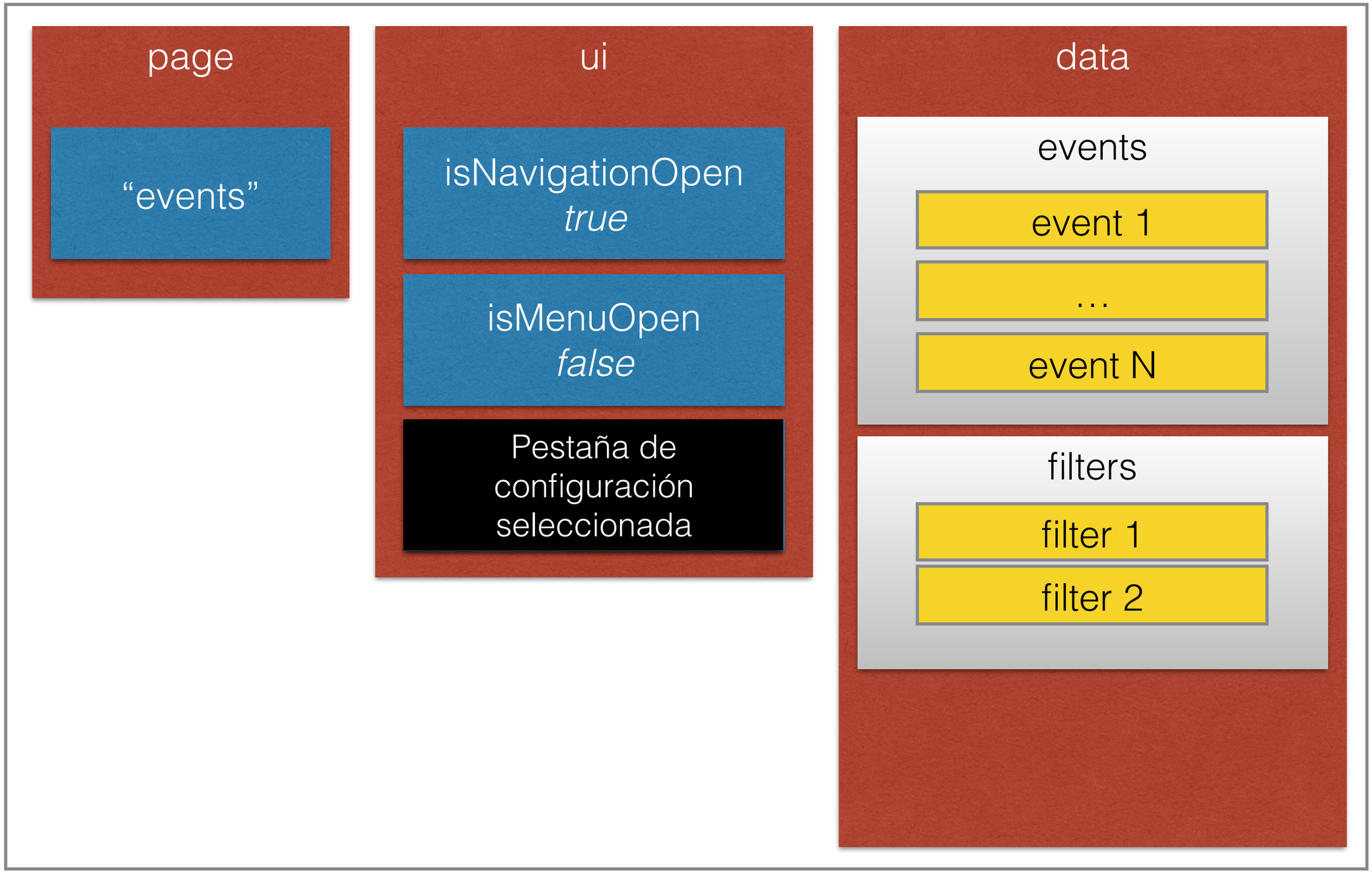
Átomo: un ejemplo



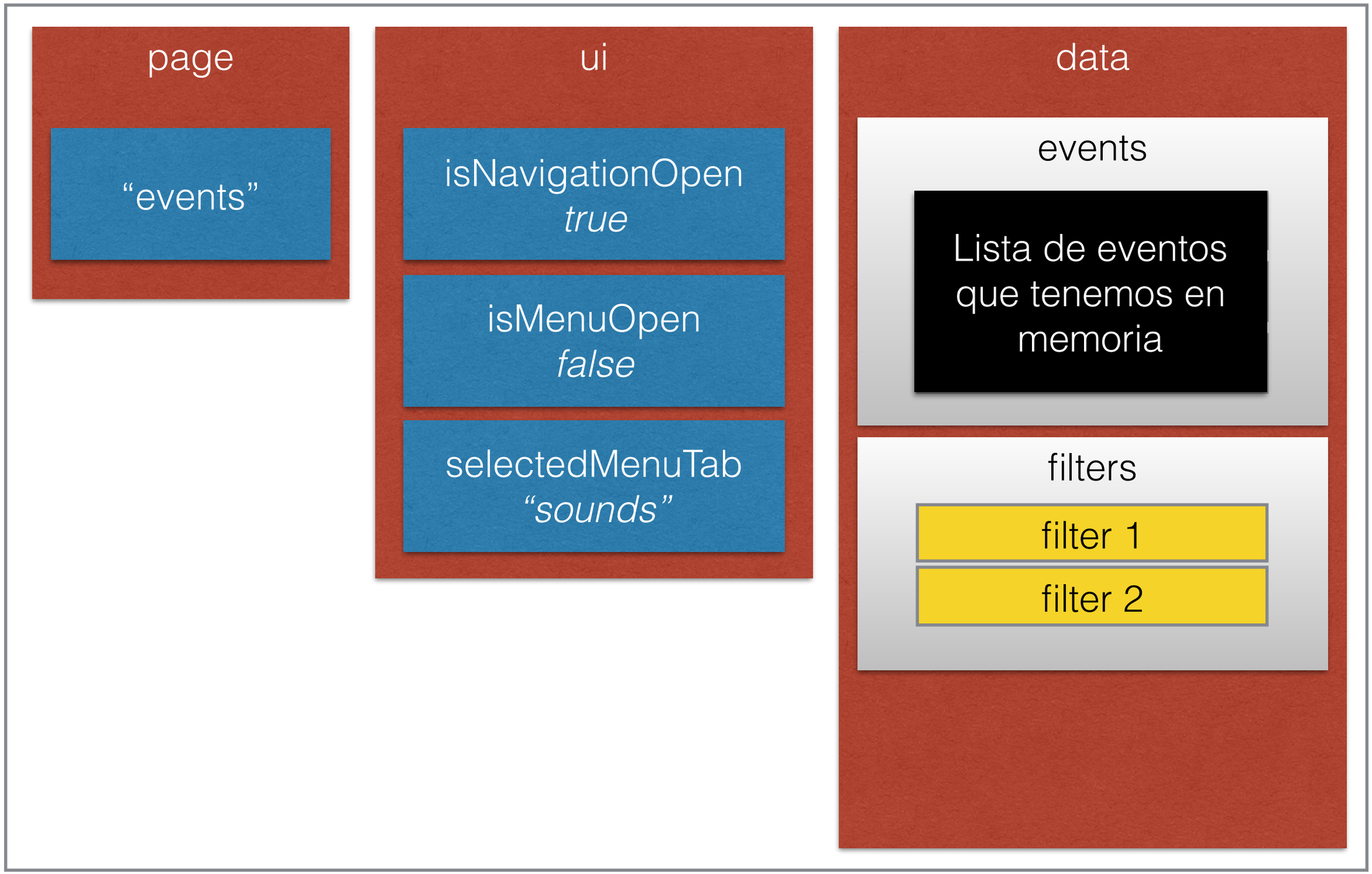
Átomo: un ejemplo



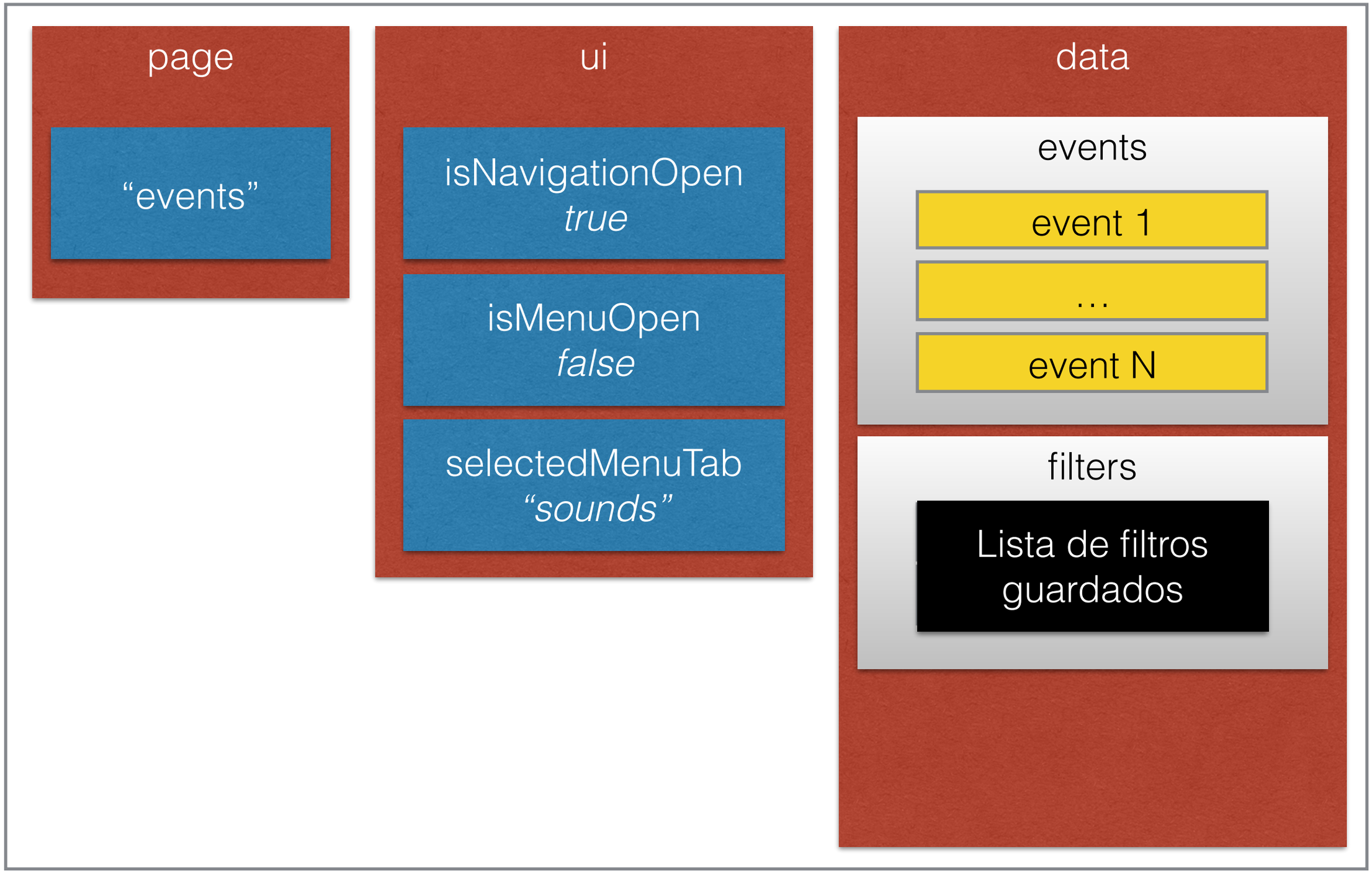
Átomo: un ejemplo



Átomo: un ejemplo



Átomo: un ejemplo



Inmutabilidad

Un objeto inmutable es aquel que no permite mutaciones (cambios) una vez creado

Inmutabilidad

Una operación sobre un objeto inmutable devuelve siempre **un nuevo objeto inmutable,** nunca actúa sobre el que pasamos como parámetro

operacion(obj1, arg1) -> obj2

Inmutabilidad

Determinar la **igualdad** de dos objetos inmutables es **muy eficiente**

(aunque sean vectores, listas, conjuntos…)

Inmutabilidad

- Se elimina la programación defensiva y la complejidad del cambio:
  - Modificar un objeto en un punto del código no tiene efectos secundarios…
  - …porque cada cambio genera **otro** objeto (no se puede modificar el original nunca)

Inmutabilidad

```
var numeros = [1,2,3];

var copia1 = numeros;
var copia2 = numeros;

copia1.push(4);
copia1.push(5);

console.log(copia1);
console.log(copia2);
```

¿Qué valor tendrán **copia1** y **copia2**?

Inmutabilidad

¡¡Los dos serán [1,2,3,4,5]!!

Inmutabilidad

- Probemos con objetos…

```
var usuario = {
  nombre: "carlos",
  email: "carlos@redradix.com"
};

var copia1 = usuario;
var copia2 = usuario;

copia1.nombre = "Bugs bunny";
console.log(copia2);
```

¿Qué valor tendrá **copia2**?

Inmutabilidad

{ nombre: 'Bugs bunny', email: 'carlos@redradix.com' }

Inmutabilidad

El único tipo de datos inmutable de forma nativa en Javascript son los strings

var nombre = “Pepe”;

var nombreMays = nombre.**toUppercase**();

Devuelve otra cadena sin modificar la original

Inmutabilidad

¿Y para objetos, Arrays, etc?

Javascript no lo soporta de forma nativa:
necesitamos una librería externa - **mori**

mori

- Es un port a Javascript de las colecciones inmutables de **Clojure:** hashMap, list, vector, set, sortedSet, queue, range, sequence
- Ofrece además herramientas de programación funcional (**map**, **reduce, first, rest…**)
- Eficiente y completa
- http://swannodette.github.io/mori/

mori

- Para utilizarlo, simplemente hacemos un **require** de esa dependencia (instalada con npm)

```
var mori = require('mori');

var mapa = mori.hashMap();
mapa = mori.assoc(mapa, "nombre", "Coyote");

console.log(mori.get(mapa, "nombre"));
```

Colecciones de mori

- Lista (list)
- Vector (vector) *
- Mapa (hashMap) *
- Conjunto (set) *
- Conjunto ordenador (sortedSet) *
- Rango (range)
- Cola (queue)

Colecciones de mori

- Veremos que en mori hay muchas operaciones sobrecargadas
- En función del tipo de colección tienen un comportamiento u otro, con la misma sintaxis.

Mapa - mori.hashMap()

- Una tabla hash de parejas clave-valor (la versión inmutable de un objeto simple en Javascript)

```
var mapa = mori.hashMap('key1', 1, 'key2', 2);
//mapa = { "key1" 1, "key2" 2 }
var mapaComplejo = mori.hashMap("data", mapa);
// mapaComplejo = { "data" { "key" 1, "key2", 2 } }
var mapa2 = mori.assoc(mapa, "key3", 3);
// mapa2 ={ "key" 1, "key2" 2, "key3" 3 }
var mapa3 = mori.dissoc(mapa2, "key1");
// mapa3 ={ “key2” 2, “key3” 3 }
```

Mapa - mori.hashMap()

Después del código anterior, ¿qué contiene la variable **mapa**? ¿Y **mapa2**?

Mapa - operaciones

- **assoc** - mori.assoc(map, key, value)
  Asocia una nueva tupla clave-valor en el mapa (también en vector)

- **dissoc** - mori.dissoc(map, key)
  Elimina una clave del mapa

- **get** - mori.get(map, key)
  Devuelve un valor del hashMap a partir de su clave

- **hasKey** - mori.hasKey(map, key)
  Devuelve verdadero si el mapa contiene la clave

Mapa - operaciones

- **assocIn** - mori.assocIn(map, array_claves, valor)
  Asocia una tupla clave-valor en un mapa anidado

```
var data = mori.assocIn(mori.hashMap(), ["user", "name"], "Carlos")
```

{ “user” {“name” “Carlos” } }

- **getIn** - mori.getIn(map, array_claves [,not-found])
  Devuelve un valor en un hashMap anidado, opcionalmente devuelve not-found si no se encuentra la clave anidada

```
mori.getIn(data, ["user", "name"]);
// "Carlos"
mori.getIn(data, ["user", "email"], "notfound")
// "notfound"
```

Mapa - operaciones

- **updateIn** - mori.updateIn(mapa, array_claves, *function*)
  Actualiza el valor de una tupla anidada con el **resultado** de aplicar la función al valor actual

```
mori.updateIn(date, ["user", "name"], function(name){
  return "new " + name;
});
```

{ “user”, { “name”, “new Carlos” } }

- **keys** - mori.keys(map)
  Devuelve una secuencia con las claves del mapa

- **vals** - mori.vals(map)
  Devuelve una secuencia con los valores del mapa

- **merge** - mori.merge(map1, map2, …)
  Devuelve el resultado de fundir los mapas 2..N con el primero. Si una clave ya existía, se reemplaza.

Ejercicio 1 - mapas inmutables

1. Crear un mapa inmutable con la siguiente estructura:

```
{
  user: {
    name: "tunombre"
  }
}
```

2. Añadir la clave “email” dentro de “user” con tu email

3. Añadir un mapa anidado dentro de **user**, bajo la clave “currentJob”, con la forma { “position”: *tucargoactual* }

4. Modificar tu cargo actual por otro

5. Eliminar la clave anidada “email” del usuario

6. Imprimir en consola el mapa final resultante

Vector - mori.vector()

- Un vector (“array”) inmutable. Eficientes para añadir elementos al **final** y acceso aleatorio.

```
var v = mori.vector(1,2,3);
//[1,2,3]
var v2 = mori.conj(v, 4);
//[1, 2, 3, 4]
var v3 = mori.conj(v2, 5);
//[1, 2, 3, 4, 5]
var v4 = mori.pop(v3);
//[1, 2, 3, 4]
```

Vector - operaciones

- **conj** - mori.conj(vector, elemento)
  Añade un elemento al **final** del vector

- **assoc** - mori.assoc(vector, index, elemento)
  Modifica un elemento en la posición N
  cuidado: N tiene que ser < longitud del vector

- **nth** - mori.nth(vector, index)
  Devuelve el elemento en la posición N del vector

- **last** - mori.last(vector)
  Devuelve el último elemento del vector (colección)

Vector - operaciones

- **subvec** - mori.subvec(vector, start [,end])
  Devuelve una sección del vector como otro vector

- **pop** - mori.pop(vector)
  Devuelve un vector sin su último elemento

- **peek** - mori.peek(vector)
  Devuelve el último elemento de un vector

- **count** - mori.count(vector)
  Devuelve la longitud del vector, **o de cualquier colección** (incluidos los mapas)

Ejercicio 2 - vectores inmutables

1. Crea un vector people con 1 mapa que representa los datos de una persona. Para cada persona queremos los campos “id”, “firstName”, “lastName”

2. Añade dos nuevas personas al vector

3. Imprime por consola la última persona en el vector

4. Imprime por consola la primera persona en el vector

5. Recupera la segunda y tercera personas del vector en uno nuevo

Conjunto - mori.set()

- Colección de elementos sin repeticiones, puede contener elementos complejos (como mapas o vectores)

- El constructor acepta otras colecciones inmutables e incluso un Array de Javascript directamente:

var set = **mori.set**(“react”, “node”, “node”);
#{ “react”, “node” }
var set = **mori.set**([1,2,3,4,1,2,3,4]);
#{1, 2, 3, 4}

Conjunto - operaciones

- **conj** - mori.conj(set, element)
  Añade un elemento al conjunto.

- **disj** - mori.disj(set, element)
  Elimina un elemento de un conjunto

- **hasKey** - mori.hasKey(set, element)
  Devuelve verdadero si el conjunto contiene el elemento

Conjunto - operaciones

- **union** - mori.union(set0, set1, …)
  Devuelve la unión de dos o más conjuntos como otro conjunto

- **intersection** - mori.intersection(set0, set1,…)
  Devuelve la intersección de dos o más conjuntos

- **difference** - mori.difference(set0, set1, …)
  Devuelve la diferencia de conjuntos

- **isSubset / isSuperset** - mori.isSubset(setA, setB)
  Devuelve verdadero si el conjunto A es subconjunto / superconjunto de B

Conjunto ordenado

- **mori.sortedSet(el0, el1, …)**
  Crea un conjunto con ordenación por defecto y añade los elementos (opcionales)

- **mori.sortedSetBy(*sortFunction*, el0, el1, …)**
  Crea un conjunto ordenado según una función especificada por nosotros, y añade los elementos (opcionales)

Secuencias

- Una secuencia (**seq**) es una vista sobre una colección inmutable. No **copia** la colección, es una proyección de la original, a la que va accediendo cuando es necesario.

- Esto nos permite utilizar muchas funciones comunes con vectores, conjuntos, mapas, etc.

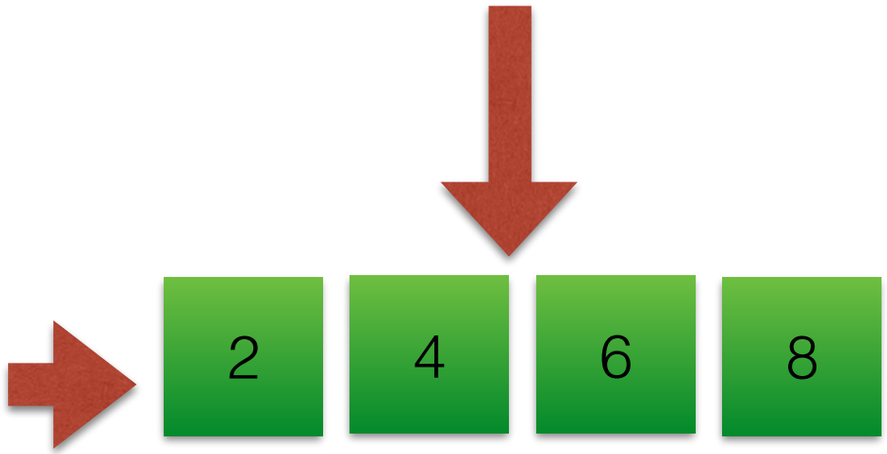
- Las operaciones de secuencias aceptan una colección y **devuelven una secuencia**

Secuencias - ejemplo:

vector

```
mori.filter(function(n){ n % 2 === 0}, vector)
```

¡secuencia, no vector!



Secuencias - Lazy

- Hay muchas operaciones que devuelven otra secuencia con una función aplicada sobre la original
- Estas operaciones **no evalúan las funciones** hasta que son necesarias para extraer uno o varios valores.
- A esto lo llamamos una operación **lazy** o perezosa, ya que delega la ejecución hasta que sea necesario
- Esto nos permite trabajar con secuencias potencialmente infinitas sin incurrir en un gran consumo de memoria

Secuencias Lazy

vector

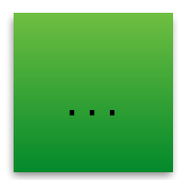
```
var pares = mori.filter(function(n){ n % 2 === 0}, vector)
```

```
mori.take(2, pares)
```

Secuencias Lazy

vector

```
var pares = mori.filter(function(n){ n % 2 === 0}, vector)
```




```
mori.take(2, pares)
```

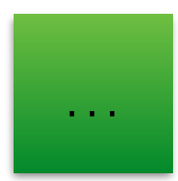
take toma N valores….

Secuencias Lazy

vector

```
var pares = mori.filter(function(n){ n % 2 === 0}, vector)
```

...de la secuencia resultante de filtrar los números pares del vector



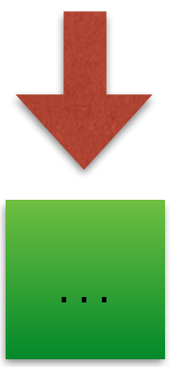

```
mori.take(2, pares)
```

Secuencias Lazy

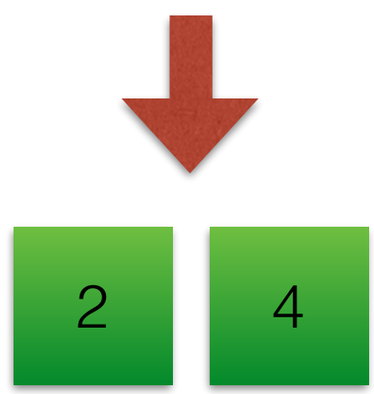
vector

```
var pares = mori.filter(function(n){ n % 2 === 0}, vector)
```

Aunque el vector tuviera 1 millón de elementos, **filter** sólo se ha ejecutado 4 veces (sobre 1, 2, 3, 4)



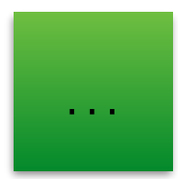

```
mori.take(2, pares)
```



Secuencias Lazy

vector

```
var pares = mori.filter(function(n){ n % 2 === 0}, vector)
```

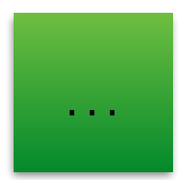


El resultado de una operación **lazy** es una vista sobre la colección o secuencia subyacente…

Secuencias Lazy

vector

```
var pares = mori.filter(function(n){ n % 2 === 0}, vector)
```



…que queda “a la espera” de que le pidan valores

Secuencias: operaciones

http://swannodette.github.io/mori/#sequences

- first
- rest
- seq
- cons
- concat
- flatten
- intoArray
- each
- map*
- mapcat
- filter*
- remove*
- reduce
- reduceKV
- take
- iterate*
- repeat*
- repeatedly*
- partition
- partitionBy
- groupBy

* = Lazy

Secuencias: operaciones habituales

- **first** - mori.first(col)
  Devuelve el primer elemento de una colección (vector, conjunto, conjunto ordenado…)

- **rest** - mori.rest(col)
  Devuelve el resto de elementos de una colección como una secuencia. No es lazy, la evalúa entera

- **seq** - mori.seq(col)
  Devuelve una secuencia a partir de la colección (la mayoría de operaciones lo hacen de forma implícita)

- **intoArray** - mori.intoArray(col)
  Convierte una secuencia/colección en un Array de Javascript, con elementos inmutables. La utilizamos **muy a menudo**.

Secuencias: operaciones habituales

```
var mori = require('mori');

var mapa = mori.hashMap("key1", 100, "key2", "una cadena", "key3", true);
var set = mori.set([1,2,3,4,5]);
var vector = mori.vector("foo", "bar", "foobar");

console.log("First de mapa", mori.first(mapa));
// ["key1" 100]
console.log("First de set", mori.first(set));
// 1
console.log("First de vector", mori.first(vector));
// "foo"

console.log("Rest de mapa", mori.rest(mapa).toString());
// (["key2" "una cadena"] ["key3" true])
console.log("Rest de set", mori.rest(set).toString());
// (2 3 4 5)
console.log("Rest de vector", mori.rest(vector).toString());
// ("bar" "foobar")

console.log("intoArray de mapa", mori.intoArray(mapa));
// [ ["key1" 100], ["key2" "una cadena"], ["key3" true] ]
console.log("intoArray de set", mori.intoArray(set));
// [ 1, 2, 3, 4, 5 ]
console.log("intoArray de vector", mori.intoArray(vector));
// [ 'foo', 'bar', 'foobar' ]
```

Secuencias: operaciones habituales

```
var mori = require('mori');

var mapa = mori.hashMap("key1", 100, "key2", "una cadena", "key3", true);
var set = mori.set([1,2,3,4,5]);
var vector = mori.vector("foo", "bar", "foobar");

console.log("First de mapa", mori.first(mapa));
// ["key1" 100]
console.log("First de set", mori.first(set));
```

La secuencia generada a partir de un mapa contiene **vectores** con dos elementos, la clave y el valor

[x x x] sin comas -> vector
[x, x, x] -> Array JS
(x x x x) -> secuencia

```
console.log("intoArray de mapa", mori.intoArray(mapa));
// [ ["key1" 100], ["key2" "una cadena"], ["key3" true] ]
console.log("intoArray de set", mori.intoArray(set));
// [ 1, 2, 3, 4, 5 ]
console.log("intoArray de vector", mori.intoArray(vector));
// [ 'foo', 'bar', 'foobar' ]
```

Secuencias: operaciones habituales

- **each** - mori.each(col, *f*)
  Itera sobre los elementos de la secuencia aplicando *f* a cada uno. Para efectos secundarios.

- **map** - mori.map(*f*, col0, col1…)
  Devuelve una secuencia **lazy** que representa la colección original con *f* aplicada a cada elemento.

- **reduce** - mori.reduce(*f*, [initialValue,] col)
  Acumula una colección completa en un único valor. *f* tiene la firma f(acc, value)

- **reduceKV** - mori.reduceKV(*f*, [initialValue], col)
  Una variante de **reduce** para Mapas y Vectores. La función *f* recibe la clave/índice y el valor.

Secuencias: operaciones habituales

```
var mori = require('mori');

var mapa = mori.hashMap("key1", 100, "key2", "una cadena", "key3", true);
var set = mori.set([1,2,3,4,5]);
var vector = mori.vector("foo", "bar", "foobar");

//map, reduce

var sequence = mori.map(function(item){
  return item * 100;
}, set);
console.log('Map sobre set', sequence.toString());

var sumatorio = mori.reduce(function(acc, value){
  return acc + value;
}, 0, sequence);
console.log('Reduce sobre seq', sumatorio);

// obtenemos el índice de un elemento en un vector con reduceKV
function getIndexOf(vector, value){
  return mori.reduceKV(function(acc, index, val){
   if(val === value){
      return index;
    }
    else {
      return acc;
    }
  }, -1, vector);
}
console.log('ReduceKV sobre vector', getIndexOf(vector, 'foobar'));
```

Secuencias: operaciones habituales

```
var mori = require('mori');

var mapa = mori.hashMap("key1", 100, "key2", "una cadena", "key3", true);
var set = mori.set([1,2,3,4,5]);
var vector = mori.vector("foo", "bar", "foobar");

//map, reduce

var sequence = mori.map(function(item){
  return item * 100;
}, set);
console.log('Map sobre set', sequence.toString());

var sumatorio = mori.reduce(function(acc, value){
  return acc + value;
}, 0, sequence);
console.log('Reduce sobre seq', sumatorio);

// obtenemos el índice de un elemento en un vector con reduceKV
function getIndexOf(vector, value){
  return mori.reduceKV(function(acc, index, val){
   if(val === value){
      return index;
    }
    else {
      return acc;
    }
  }, -1, vector);
}
console.log('ReduceKV sobre vector', getIndexOf(vector, 'foobar'));
```

(100 200 300 400 500)

1500

2

Secuencias: operaciones habituales

- **filter** - mori.filter(*f*, col)
  Devuelve una secuencia **lazy** que contiene los elementos de *col* que devuelven “verdadero” para el predicado *f*

- **remove** - mori.remove(*f*, col)
  El opuesto a filter: devuelve una secuencia **lazy** sin los elementos de *col* que devuelven “verdadero” para el predicado *f*

- **sort** - more.sort([*f*], col)
  Ordena la colección y devuelve una secuencia. La función de comparación *f* es opcional.

- **take** - mori.take(*n*, col)
  Devuelve una secuencia **lazy** con los *n* primeros elementos de la colección.

- **drop** - mori.drop(*n*, col)
  Devuelve una secuencia **lazy** con la colección original, descartando los *n* primeros

Secuencias: operaciones habituales

```
var vector2 = [1,2,3,4,5,6,7,8,9,10,11,12,14,15,16,17,18,19,20];
//filter
var impares = mori.filter(function(n){
  return n % 2 === 1;
}, vector2);
//remove
var imparesMayoresQue10 = mori.remove(function(n){
  return n < 10;
}, impares);

console.log('Impares > 10', imparesMayoresQue10.toString());
//sort
var deMenorAMayor = mori.sort(function(a,b){
  return b-a;
}, imparesMayoresQue10);

console.log(‘Impares de menor a mayor', deMenorAMayor.toString());
//take
console.log('Take 2', mori.take(2, deMenorAMayor));
//drop
console.log('Drop 7 de impares', mori.drop(7, impares));
```

Secuencias: operaciones habituales

```
var vector2 = [1,2,3,4,5,6,7,8,9,10,11,12,14,15,16,17,18,19,20];
//filter
var impares = mori.filter(function(n){
  return n % 2 === 1;
}, vector2);
//remove
var imparesMayoresQue10 = mori.remove(function(n){
  return n < 10;
}, impares);

console.log('Impares > 10', imparesMayoresQue10.toString());
//sort
var deMenorAMayor = mori.sort(function(a,b){
  return b-a;
}, imparesMayoresQue10);

console.log('Impares de menor a mayor', deMenorAMayor.toStri
//take
console.log('Take 2', mori.take(2, deMenorAMayor));
//drop
console.log('Drop 7 de impares', mori.drop(7, impares));
```

(11 15 17 19)

(19 17 15 11)

(19 17)

(17 19)

Secuencias: generadores

- **repeat** - mori.repeat(*n*, valor)
  Devuelve una secuencia **lazy** que contiene n veces el valor indicado. Sin *n* es una secuencia infinita.

- **repeatedly** - more.repeatedly(n, *func*)
  Devuelve una secuencia **lazy** que contiene n veces el resultado de ejecutar la función *f.* Sin *n* es una secuencia infinita.

Secuencias: generadores y lazyness

```
var mori = require('mori');

//una secuencia infinita de números aleatorios
var aleatorios = mori.repeatedly(Math.random);

//números aleatorios entre 0 y 25
var enteros = mori.map(function(n){
  console.log('Redondeando: ' + n);
  return Math.floor(n*25);
}, aleatorios);

// tomamos 10 enteros aleatorios
var muestra = mori.take(10, enteros);

// y ahora forzamos la evaluación de todo
// para imprimirlo por consola

console.log('Enteros aleatorios', mori.intoArray(muestra));
```

Secuencias: generadores y lazyness

Redondeando: 0.3983391302172094 6
Redondeando: 0.9514610418118536
Redondeando: 0.8935665111057 46
Redondeando: 0.1900905289221555
Redondeando: 0.5578903879504651
Redondeando: 0.1318754926323890 7
Redondeando: 0.7512924217153341
Redondeando: 0.2079645127523690 5
Redondeando: 0.2997703517321497
Redondeando: 0.5525723269674927
Enteros aleatorios [9, 23, 22, 4, 13, 3, 18, 5, 7, 13]

Secuencias <-> colecciones

- Las operaciones de **secuencias** que reciben una colección nos devuelven una **secuencia**.

- ¿Pero y si queremos el mismo tipo de colección original?

➡ **into** - mori.into(col, from)
Añade todos los elementos de la colección/ secuencia *from* a la primera y devuelve una colección del tipo *col*.

Secuencias <-> colecciones

```
var mori = require('mori');

var aleatorios = mori.repeatedly(Math.random);

var enteros = mori.map(function(n){
  console.log('Redondeando: ' + n);
  return Math.floor(n*25);
}, aleatorios);

var muestra = mori.take(10, enteros);

// y ahora forzamos la secuencia a un conjunto
// para eliminar los repetidos. Aquí se evalúa todo
var sinRepetidos = mori.into(mori.set(), muestra);

console.log('Conjunto de enteros aleatorios', sinRepetidos);
```

Funciones auxiliares de mori

- **partial** - mori.partial(*f*, x, y, …)
  Devuelve *una nueva función* equivalente a *f* con los argumentos proporcionados parcialmente aplicados.

Ejemplo de partial

```
var mori = require('mori');

function multiplicarPor(x, n){
  return n*x;
}

var por5 = mori.partial(multiplicarPor, 5);
var por6 = mori.partial(multiplicarPor, 6);

console.log(por5(10)); //50
console.log(por6(10)); //60
```

Útil para fijar uno o varios argumentos y dejar “libres” el resto

Conversión mori <-> Javascript

- **toClj** - mori.toClj(x)
  Transforma recursivamente estructuras Javascript a colecciones de mori: Arrays a vectores, objetos Javascript a mapas.

- **toJs** - mori.toJs(x)
  Transforma recursivamente estructuras inmutables a Javascript. Vectores, conjuntos y listas a Arrays, mapas a objetos Javascript.

Conversión mori <-> Javascript

- El resto del mundo no “habla” mori
- Podemos convertir estructuras Javascript a colecciones inmutables: **mori.toClj(arg)**
- Podemos convertir colecciones inmutables a estructuras Javascript: **mori.toJs(col)**
- Ambos funcionan recursivamente con estructuras anidadas
- **¡¡¡¡Operaciones costosas!!!!**
- Necesario **solo** para comunicarnos con sistemas externos como APIs, WebSockets…

Ejercicio 3- mori

1. Vamos a representar “eventos” de un sistema externo con mapas que contengan claves **timestamp**, **message**

2. Crea una función que genere eventos aleatorios con timestamp entre 0 y 1000 y campos message “únicos”. (pista: Math.random, Math.round)

3. Crea un conjunto ordenado, **events**, cuya función de ordenación acepte estos eventos, y los ordene por la clave **timestamp** de forma descendente o por la clave **message** si los tiempos coinciden

4. Introduce 25 eventos aleatorios en el conjunto, usando la función creada en el punto 2.

5. Crea un conjunto con la misma ordenación, **top10events,** que contiene los 10 primeros elementos en **events**

Ejercicio 3 - mori (2)

6. Crea un conjunto con la misma ordenación, **last10events**, que contenga todos los elementos excepto los 15 primeros.

7. Crea otro conjunto **recent** que contenga todos los eventos cuyo timestamp sea > 500

8. Genera un Mapa de mori **stats** con campos **longestmessage** y **sumtimestamp**, que contengan el mensaje más largo y la suma agregada de los tiempos de todos los eventos, respectivamente.

9. Imprime todos los resultados por consola, convertidos a estructuras Javascript

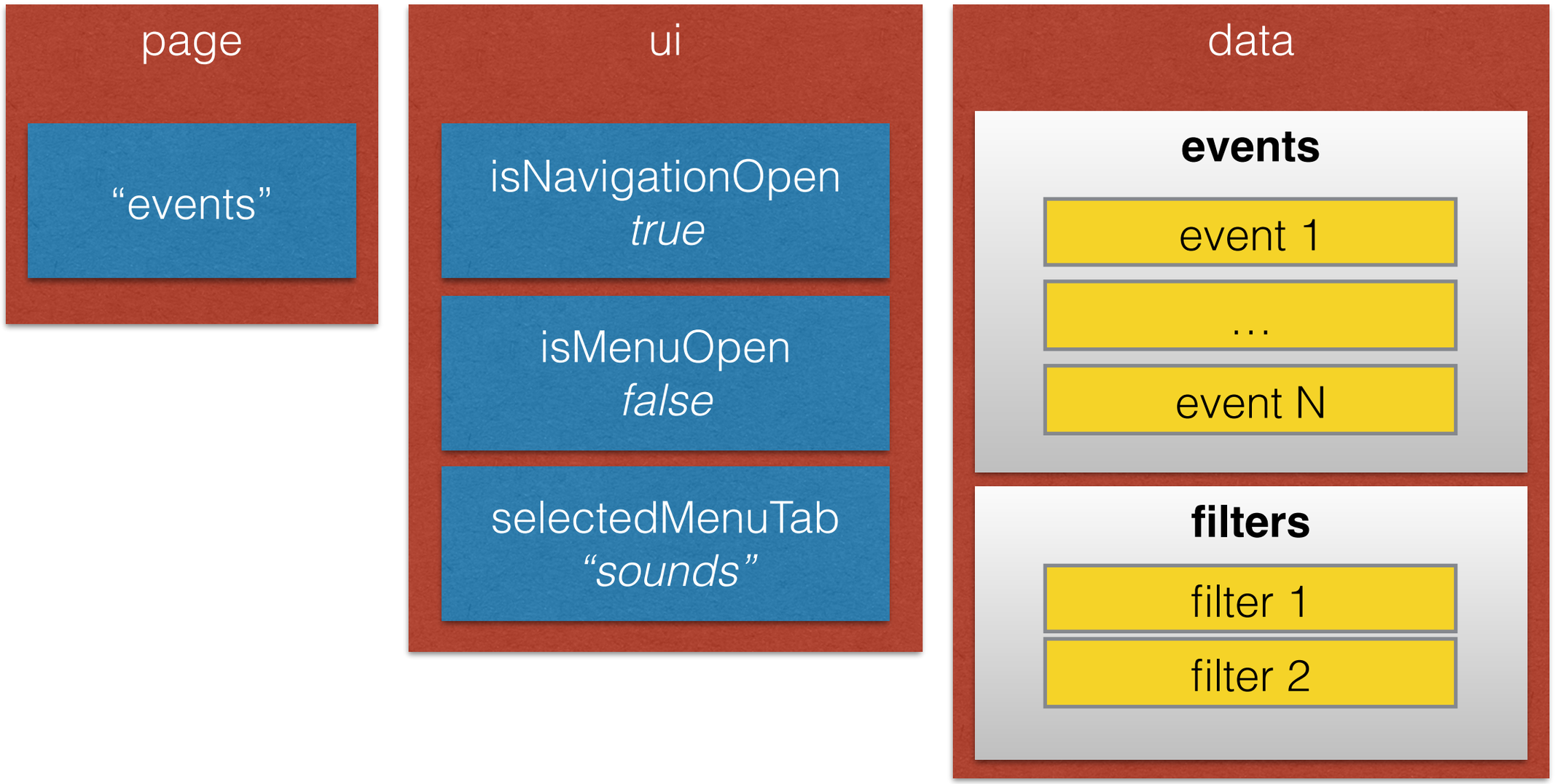
¿Qué es el átomo?

- Una referencia **mutable** a una estructura de datos **inmutable**
  - **state = mori.op(state, ….)**
- Puesto que cada operación sobre una colección inmutable devuelve otra…
- …lo que vamos cambiando es la referencia (reasignando la variable **state** para que apunte al nuevo objeto)

¿Qué es el átomo?

- El único almacén de **estado** en nuestra aplicación
- Un **árbol de estado** que contiene todos los datos de nuestra aplicación, incluyendo:
  - Datos externos (proporcionados por un servidor)
  - Datos internos (estado UI)

¿Qué es el átomo?




```
mori.get(state, "page");
mori.getIn(state, ['ui', 'isNavigationOpen']);
mori.count(mori.getIn(state, ['data', 'events']));
```

¿Para qué usamos el átomo?

- Para almacenar el estado completo de la aplicación en **un momento determinado** (un estado determinado)
- Para **actualizar** la UI de forma **automática** cuando el átomo **cambia**

UI = f(atom)

¿Cómo funciona el átomo?

- Se puede **apuntar** el átomo completo a otro mapa de una vez (**swap**), o actualizarlo parcialmente (**update**)
- Un cambio en cualquier punto del átomo disparará el evento **CHANGE**
- La UI escucha este evento para repintarse cuando haya un cambio
- El **tiempo** lo marca los cambios del átomo
- Sin cambios, la aplicación es estática

atom en nuestra arquitectura

- Un wrapper sobre un Mapa de mori
- Una única variable **state** que apunta a este hashMap
- Operación de consulta del estado actual del átomo(**getState**) - “Dame el valor actual al que apuntas”
- Operación de sustitución del átomo completo (**swap**) - “Apunta a este nuevo valor”

atom en nuestra arquitectura

- Operaciones de consulta de claves anidadas (**getIn**)
- Operaciones de modificación de claves anidadas (**assocIn**, **updateIn**)
- Operaciones de modificación “silenciosas” (**silentAssocIn**, **silentUpdateIn**)

Ejercicio - implementar un átomo

1. Crea un módulo Javascript que encapsule nuestro mapa inmutable. Archivo: **atom_state.js**

2. Implementa un método **getIn** que devuelva el contenido de una clave anidada en el átomo

3. Implementa un método **assocIn** que asocie una clave anidada en el átomo

4. Implemente un método **updateIn** que modifique una clave asociada en el átomo mediante una función

Ejercicio - implementar un átomo

5. Implementa un método **getState** que devuelva el estado completo

6. Implementa un método **swap** que reciba un objeto Javascript y reemplace el estado en nuestro átomo por su correspondencia inmutable

7. Implementa un método **addListener** que acepta una función, y la añade a una lista de funciones como *handlers* del evento “change” de nuestro átomo.

8. Haz que cada operación sobre el átomo llame a los *handlers* registrados pasándoles el nuevo estado.

Ejercicio - implementar un átomo

Puedes validar tu átomo ejecutando:

> cd ejercicios/tema2/ && node atom_pruebas.js

(Espera el átomo implementado en la misma carpeta, con el nombre *atom_state*)