Programación Funcional

Conceptos de JavaScript

Pepe Fabra Valverde

Disclaimer

- Al final de la sesión se compartirán las slides
- Pregunta sin miedo
- Me gustaría que fuese más dinámico, está a rebosar de contenido

Los conceptos son generales, pero uso JavaScript para aterrizar conceptos

Scope

- Elementos y conceptos principales
  - No van a ser todos los conceptos

Scope

- Elementos y conceptos principales
  - No van a ser todos los conceptos
- Cómo enfocarlos

Programación Funcional

$$y = f(x)$$

input

$$y = f(x)$$

function

$$y = f(x)$$

output

$$y = f(x)$$

SABER MÁS
Hindley–Milner

Hindley–Milner

Roger Hindley y Robin Milner,

Un sistema de tipados usado en cálculo lambda, usado en algunos lenguajes funcionales, impacto a nivel de industria.

Saber más: https://stackoverflow.com/questions/399312/what-is-hindley-milner

Paradigma de Programación

Paradigma de programación

- Approach concreto para resolver problemas
- Conjunto de convenciones y reglas

“No se resuelve el problema, se traduce a un universo de patrones reproducibles que sí que sé resolver”

Filosofía y objetivos

Filosofía y objetivos

- Enfoque en legibilidad
- Modelo mental más natural (una vez entendemos las bases)
- Ser declarativo, enunciamos acciones, no ordenamos instrucciones

Objetivos

- Mantener un flujo contextual
- Representar funcionalidades de una manera que garantice su propósito
- Legibilidad por diseño (es declarativo, debe serlo)
- Evitar mutaciones de estado

Programación Funcional busca escribir código como ecuaciones

Declarativo vs imperativo

Declarativo vs imperativo

Son maneras de escribir código, no son mutuamente excluyentes por lo que en buenas soluciones encontraremos ambos estilos (no demasiado mezclados)

- **Declarativo** describe qué va a ocurrir, lo enuncia en voz alta
- **Imperativo** ordena las instrucciones que se cumplirán para que algo ocurra

Declarativo vs imperativo

Eficiencia, cuál es *mejor** y por qué

*Mejor desde la eficiencia, siempre es un trade-off

Imperativo será más eficiente

- explícitamente indicas los pasos
- no llamas métodos en bucles
- tienes el control absoluto de las decisiones (doble filo)

Esto abruma...

Es mucho de golpe

…pero no tienes que entenderlo todo a la primera (ni a la decimoquinta)

- **Monads**, ni los entiendo ni los entenderé
- **Máquina de Turing**, me lo han explicado ya demasiadas veces
  - No es funcional, pero tampoco lo entiendo
- **Subredes**, … :)

Concurrencia

Concurrencia

Programación Funcional ayuda mucho a evitar problemas de concurrencia y paralelización gracias a su modelo mental

…pero eso sería desviarnos de la charla…

Aunque saldrás con buena base para explorar

Pureza

Pureness

Qué es la pureza en programación

"es una función **idempotente** que no tiene **efectos secundarios**"

…determinismo garantizado

Cuándo decimos que es puro

- Localidad, solo depende de sus argumentos (las props)
- No tiene efectos secundarios (desvíos del propósito)
- Es predecible/determinista, los mismos argumentos garantizan el mismo resultado

Es puro

- Sumar, multiplicar, restar, etc.
- formatName(name)
- function(…props) === function(…props)

Es impuro

- Date.now()
- Math.random()

pero también…

- isOnLine()
- writeToFile(filename, content)

```
let name = "…"

function formatName() {
  return name.toLowerCase()
}
```

```
function formatName(name) {
  return name.toLowerCase()
}
```

```
const MAX_RETRIES = 3

function withRetries() {
  // …

  for (… i < MAX_RETRIES)
}
```

```
function withRetries(…, retries) {
  // …

  for (… i < retries)
}
```

Efectos secundarios

Side effects

Efectos secundarios

Todo lo que no es el efecto primario de la función, es decir, no es la intención explícita.

- Single Responsability Principle
- Toda mutación o influencia no local
  - que no venga por argumentos o no se devuelva como resultado

Mutación y cambio son tratados como sinónimos

“El efecto secundario es como una distracción en el código”

¿Los podemos evitar?

No, tampoco es el objetivo.

Queremos ser conscientes y ***mitigarlos***, pero no podemos eliminarlos

- Usaremos máquinas de estados
- Seguiremos operando con I/O
- Dependemos de conexiones externas (BDD, HTTP, topics, etc.)

```
function getUserName() {
  const user = getCurrentUser()
  return user?.name
}
```

```
function getUserName(user) {
  return user?.name
}

getUserName(getCurrentUser())
```

Idempotencia

Idempotency

Idempotencia

Es un concepto con origen en las matemáticas.

Refiere a una función que siempre que reciba los mismos parámetros, devolverá el mismo resultado. En otras palabras, **determinismo**.

Mismo *input*, mismo *output*.

Todo junto...

“Una función pura es una función idempotente con el menor número posible de efectos secundarios”

“Un efecto secundario es una dependencia o mutación con un alcance (scope) no local”

“Una función idempotente es una función determinista”

Las funciones puras

Son más fáciles de testear.

- Los efectos secundarios son elementos a mockear
- La idempotencia es algo que nos garantiza pruebas
- Serán más concretas para evitar desvíos del propósito

Memoización

No le falta una “r”

Memoización

Recordar el resultado de una función idempotente para ahorrar computaciones

> Es una técnica de optimización para procesos **pesados***

*Qué es pesado dependerá de cada sistema/proceso

Memoización

Para recordar los argumentos de una función se suele implementar un key-value

- La key hará de acceso al
- Valor, sea el tipo de resultado que sea

Por lo que:

```
cache.set(props, function(props))
```

Computación y eficiencia

Evitar computaciones pesadas

- La recursividad, necesaria pero nunca eficiente
  - Se prefiere recursividad para hacer iterar (declarativo > imperativo)
  - En lo personal no lo aplico tanto
- Memoizar puede ser más pesado que lo que intentamos optimizar

¿Cómo pasamos el resultado a una key?

Para poder memoizar

Necesitamos comparar valores y para comprar los valores tenemos que serializarlos

Antes veíamos que las props serán una key, pero si me pasan varias props ¿cómo lo trato?

Serializando… pero antes veamos por qué nos hace falta

```
{ foo: "bar" } === { foo: "bar" }
```

```
{ }  ===  { }
```

```
new Object() === new Object()
```

Comparación de valores

Primitivos, no primitivos y métodos de comparación

Primitivos

Tipos nativos de un lenguaje

- Undefined
- Boolean
- String
- Number

null -> es un valor que toman los ***objetos***

IEEE 754-2008

De esto JavaScript no tiene la culpa

- Infinity vs -Infinity
- 0 vs -0
- NaN === NaN

No primitivos

Definidos en desarrollo, más o menos

- Objetos
- Funciones
- Derivados (array, clases)

Aunque en JavaScript **todo son objetos**

SABER MÁS

Primitivos en JavaScript y otros lenguajes

Lo que no sabías de primitivos

- Los primitivos tienen una value pool (caché de valores)
  - Son Number igual, pero un número sin decimales es más eficiente
  - En Java no todos los números pueden comprarse con (==)
- Las Strings no son primitivos reales a bajo nivel
  - Pero los lenguajes de alto nivel cubren ese comportamiento
  - Por eso usamos métodos específicos
    - `.localeCompare, .equals(), str_equals()`

Profundiza en: https://v8.dev/blog/react-cliff

Pasar valores en JavaScript

Pasar valores

¿De qué maneras pasábamos valores en programación?

Paso el valor por…

Pasar valores

¿De qué maneras pasábamos valores en programación?

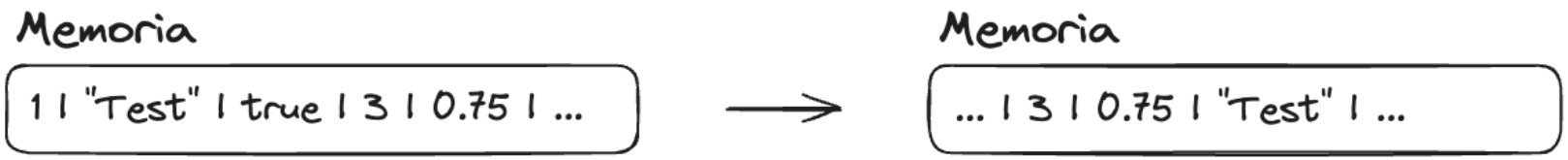
Paso el valor por…

- **Valor**, tipo primitivo, guardado en memoria tal cual
  - Más o menos, las strings siguen siendo arrays internamente
- **Referencia**, tipo *no* primitivo, en memoria guardo la dirección real
  - Para mover este argumento más fácilmente
  - Todo objeto se pasa por referencia

Si lo visualizamos, más o menos sería

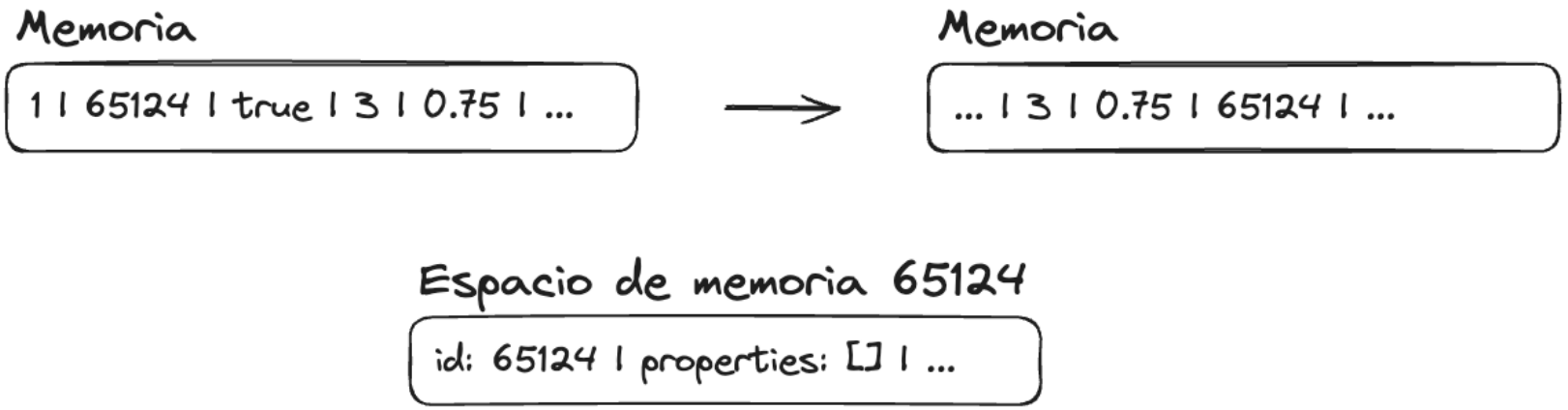
Valor

```
const variable = "Test"
example(variable)
```



Referencia

```
const variable = {}
example(variable)
```



En JavaScript se gestiona automáticamente si pasamos por referencia o valor

Comparar valores

Ahora sí

Tipos de comparación

JavaScript no podía ser de otra manera y tenemos tres maneras de comparar valores…

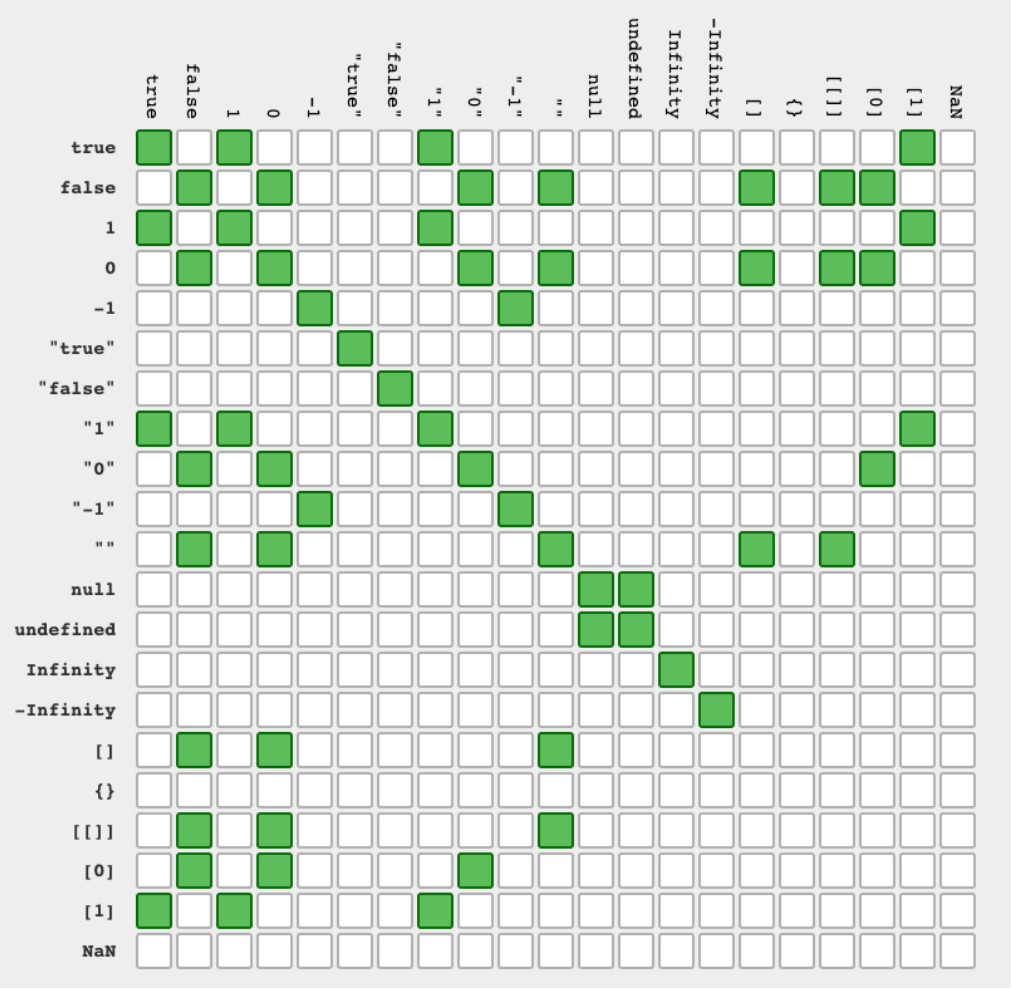
- Loose equality (==)
- Strict equality (===)
- Same value (`Object.is`)

Loose equality

No lo usamos directamente.

Es lo que permite *truthy* y *falsy*

Parece más *El juego de la vida* de Matt Conway



Strict equality

Compara el valor y el tipo, no entiende de *truthy* y *falsy*.

Es la comparación más parecida al resto de lenguajes

Same value

Strict equality, pero contempla el IEEE 754 (números)

Serialización

Por qué serializamos

En general, porque queremos transmitir contenido en red, memoria o fichero

Para memoización, queremos guardar una key en memoria que podamos consultar consistentemente

Serialización en JavaScript

De nuevo, JavaScript siempre va a su bola… y eso es bueno.

Dato: Java es all-in por diseño en serialización (chorprecha, no todo es serializable)

¿Qué tenemos a nuestra disposición?

- JSON.parse*
- devalue, del creador de Svelte

Ya sabemos de pureza de funciones, cómo memoizar y que tenemos que serializar...

Ahora vamos a explicar conceptos más funcionales ;^)

Alto Orden

Alto Orden

Son funciones que no se usan directamente, sino que son pasos intermedios para lo que se usa. Nos permiten potenciar la reusabilidad.

Una escuela no *saca faena*, educa a las personas para que sean estas las que saquen la faena… las escuelas son entidades de **alto orden**.

“Un elemento de alto orden puede ser el input y/o el output de una función”

High-Order Function

Una función que crea otras a partir de algo (función, valor, contexto, etc.)

- benchmark(function)
- logger(function)
- transactional(function)

No utilizo benchmark directamente, sino que utilizo la función que me devuelva

High-Order Component

Un componente que potencia otro, se usa para lo que se va a mostrar en verdad

- navigation(UsersPage)
- myContextHOC(PureComponent)
- withErrorBoundary(UnsafeComponent)

Ciudadanía de primera clase

Primera clase

Hace referencia a que no se le veta nada, no tiene *más* restricciones de las que añade el lenguaje.

Es decir, es un elemento que puede comportarse como valor, en **runtime** puede crearse, modificarse, desecharse, se puede pasar como argumento de una función, o ser el resultado de otra

Ejemplo de primera clase

Clases vs String

Ejemplo de primera clase

Clases vs String

Bajemos a C, ¿se mantiene?

Lo que es primera clase en un lenguaje, no está garantizado que lo sea en otro

“Se comporta como un primitivo del lenguaje”

Como un primitivo

- Una clase no es un primitivo
  - Lo son sus instancias
- No creo/modifico/elimino clases, creo/modifico/elimino instancias
  - ReflectClass -> modifica una instancia

Contexto de uso

Se usa para hablar de lenguajes de programación, sirviendo de comodín para expresar comportamientos y propiedades de entidades de este.

No suele tener mucho uso en el día a día del lenguaje

Closure

Pero explicados de verdad

Closure

Podríamos tratarlos como una implementación de High-Order Functions.

- Cuando una función recibe otra por parámetro
- Cuando se crea una función dentro del cuerpo de otra
- Cuando devolvemos una función como resultado

Contexto

Los closures nos permiten nutrir de contexto o encapsularlo

- useState
- Promises

SABER MÁS

Horizontalidad de closures

Horizontalidad

- IIFE -> Inmediately Invoked Function Expression
- Se ejecuta cuando el DOM ha cargado
  - `(() => {})()`
- jQuery, debug en consola (redeclaración de constantes)
- Web scraping desde la consola

Closures, first-class y high-order

Closures, first-class y high-order

- Funciones creadas dentro de otras funciones
- Encapsulamos o nutrimos de contexto con closures
- Pueden recibir funciones y devolverlas (o crearlas)
- Comportamiento sin restricciones *extra*

¿Dudas?

Localidad

Localidad

Concepto de la física (cuántica) *aplicado*

- **Física local**, un objeto se ve afectado por su entorno directo
- **Física no local**, un objeto puede verse afectado por elementos más allá de su entorno directo

Referencia: https://es.wikipedia.org/wiki/Principio_de_localidad

“Un objeto es influenciado directamente sólo por su entorno más inmediato”

Localidad en programación

Dependencia de algo que no son argumentos, o que los está mutando*

*El scope escapa a su entorno más inmediato

Transparencia referencial

$$y = f(x)$$

transparencia referencial

$$const\ y = f(x)$$

Transparencia referencial

Una referencia devuelve siempre el mismo valor. *No es idempotencia*.

Siempre que acceda a una variable, tendré el mismo valor porque **no muta**.

Es un término que hace referencia al hecho de que no mutamos variables, creamos nuevas.

```
let balance = getUserBalance(user)
let cost = getTotalCost(basket)
cost *= 1 - discount
balance -= cost
return balance
```

```
const initialBalance = getUserBalance(user)
const initialCost = getTotalCost(basket)
const discountedCost = initialCost * (1 - discount)
const balanceAfterCost = initialBalance - discountedCost
return balanceAfterCost
```

Inmutabilidad

Inmutabilidad

La inmutabilidad nos da consistencia y mantiene la transparencia referencial.

Los datos son trazables en runtime y no tengo sorpresas. Útil para concurrencia

Si tengo que modificar algo, creo una copia (no tiene por qué ser entera)

Implementaciones

Cada sistema y estructura traduce la inmutabilidad diferentemente

- Algunas veces serán nodos y mantendrán referencias (ej. commits)
- Otras no computarán nada hasta una acción (ej. Spark)

Casos de éxito

Para JavaScript -> mori

De lenguajes -> clojure

Pipelines

Pipelines

Contribuye a la transparencia referencial.

Visualizo el problema como una serie de acciones con entrada y salida.

El ejemplo más conocido son los comandos de Linux, aunque Jenkins se queda cerca.

Se usa mucho en sistemas concurrentes

```
let text = "…"
text = formatName(text)
text = sanitize(text)
text = beautify(text)
```

```
const text = beautify(
  sanitize(
    formatName(
      "…"
    )
  )
)
```

```
const text = |> "..."
|> formatName(%)
|> sanitize(%)
|> beautify(%)
```

TC39 Proposal

Technical Comittee 39 llevan el Spec y la evolución de ECMAScript (JavaScript estandarizado)

Hay una propuesta en progreso para el pipe operator (| >) actualmente en estado 2.

- *Avanzada*, pero sigue siendo una **propuesta**

Seguimiento: https://github.com/tc39/proposal-pipeline-operator

Extra: spec proposals

Si te interesa saber más acerca de las stages de una proposal

https://tc39.es/process-document/

SABER MÁS

Casos de uso de pipelines

En el mundo real...


MARIO BROS.

Pipelines en el mundo real

La CLI sigue el patrón de pipelines, pueden encadenarse (e incluso vivir desacopladas)

Los comandos suelen diseñarse para ser altamente reutilizables:

- Leen del std-in (standard input)
- Y escriben al std-out (standard output)

Algunos proyectos usan pipelines como patrón de arquitectura

Loops y recursividad

Loops funcionales

¿Por qué decía que se usaba recursividad?

Programación funcional busca expresar los problemas como ecuaciones matemáticas, las ecuaciones no tienen loops, tienen recursividad

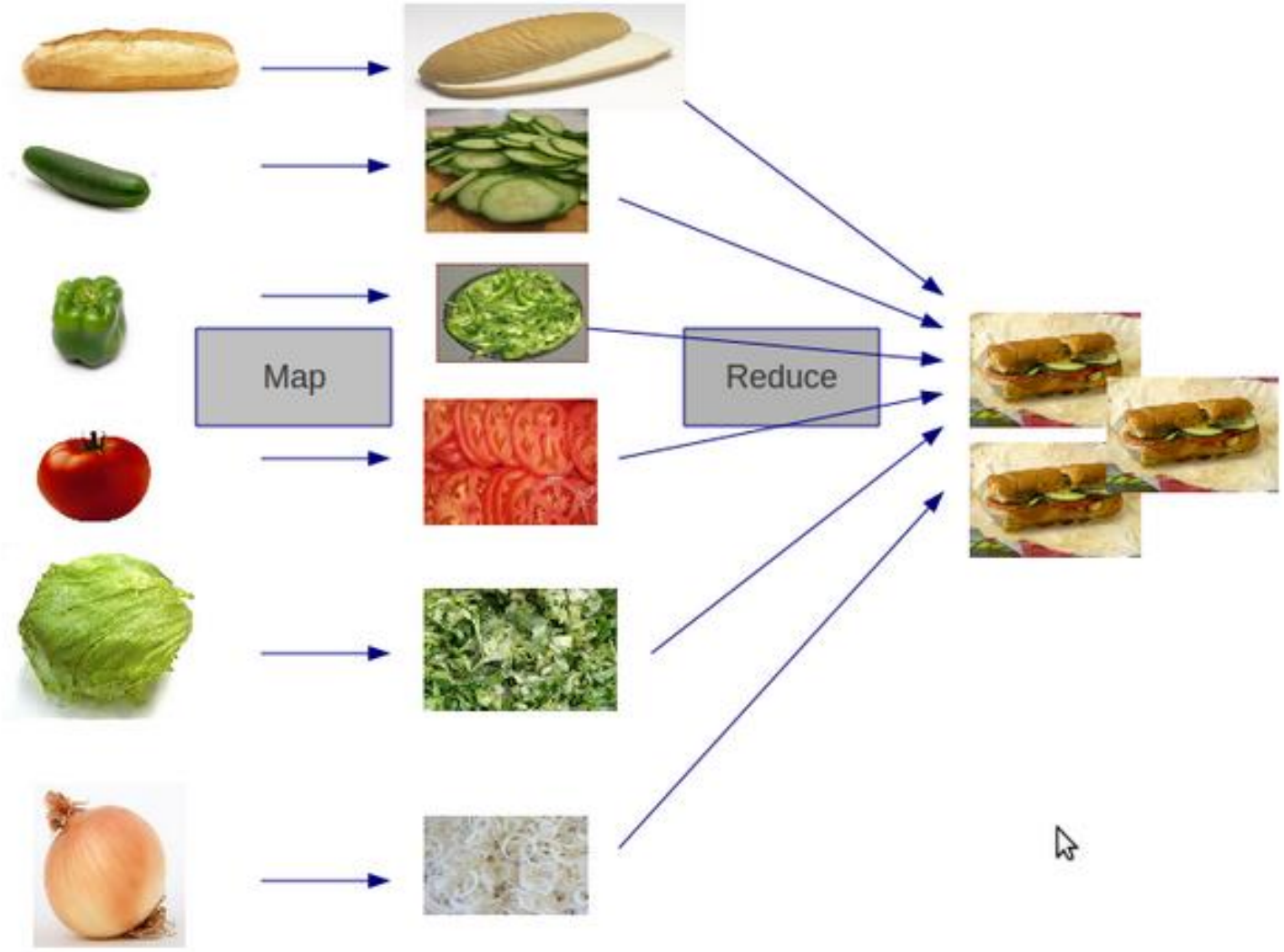
- Ejecuta esta fórmula con el 1, ahora con el 2… y así hasta N
- Sumatorios, factoriales, Fibonacci, etc.

```
function summatory(n) {
  let total = 0;
  for (let i = 1; i <= n; i++) {
    total += i;
  }
  return total;
}
```

```
function summatory(n) {
  if (n <= 0) return 0;
  return n + summatory(n - 1);
}
```

MapReduce

De Programación Funcional a Big Data


Map
Reduce

MapReduce

- Adopción general más allá de lo funcional
- Uso en Big Data (Hadoop, Spark, sistemas concurrentes)

- **Map** -> modifica elemento a elemento de la colección
  - Mantendrá la longitud inicial de la colección
- **Reduce** -> modifica la colección recorriendo elemento a elemento
  - De la colección se produce un nuevo valor (otra colección o no)

Casos de uso

Conteo de palabras

- texto.split(porPalabra).map(palabraNumero).reduce(contarPalabras)

Sumatorio

- numeros.reduce(suma)

Número más alto o más bajo

- numeros.reduce(comparar)

mapAndFilter

- coleccion.reduce(mapAndFilter)

Currificación

El sutil arte de complejizar lo que es sencillo

Currying

- **Qué es**
  - Desglosar la solución en pasos que se ejecutan como funciones independientes
  - Cada paso recibe un único argumento, son closures
- **Ejemplos y casos de uso**
  - const onClick = () => export({ … }).csv().withHeader().download()
  - alCuadrado(20).power().power().power()
- **Disclaimer**
  - No siempre tienen un caso de uso… o merecen la pena
  - A veces es un poco *calentada de cabeza* (overengineering)

Currying

- **Qué es**
  - Desglosar la solución en pasos que se ejecutan como funciones independientes
  - Cada paso recibe un único argumento, son closures
- **Ejemplos y casos de uso**
  - const onClick = () => export({ … }).csv().withHeader().download()
  - alCuadrado(20).power().power().power()
- **Disclaimer**
  - No siempre tienen un caso de uso… o merecen la pena
  - A veces es un poco *calentada de cabeza* (overengineering)

Currying

- **Qué es**
  - Desglosar la solución en pasos que se ejecutan como funciones independientes
  - Cada paso recibe un único argumento, son closures
- **Ejemplos y casos de uso**
  - const onClick = () => export({ … }).csv().withHeader().download()
  - alCuadrado(20).power().power().power()
- **Disclaimer**
  - No siempre tienen un caso de uso… o merecen la pena
  - A veces es un poco *calentada de cabeza* (overengineering)

Currying en el mundo real

- Promises
- Builder pattern
- Construir una query

Sobre todo, el uso lo tiene en cálculo

Eficiencia

Eficiencia

¿Cómo de eficiente es la programación funcional?

¿Es el objetivo que busca?

Consistencia

Lo que la programación funcional busca es consistencia, es tener un buen modelo mental para resolver problemas, eso viene a coste de la eficiencia.

Ten en cuenta que:

- Sigue siendo un paradigma bastante eficiente (ignoremos recursividad)
- Te ayuda a resolver el problema sin demasiadas vueltas
    - No añades pasos innecesarios para intentar resolverlo (ej. concurrencia)

Declarativo vs imperativo

Ligado con eficiencia

BigO notation

- Declarativo -> O(n * m + n)
  - Es legible, lees lo que ocurre, pero tiene coste extra
- Imperativo -> O(n * m)
  - Es eficiente, dices qué va a ocurrir

Declarativo

- Programación Funcional añade pasos extra en pro
  - De la legibilidad
  - De un buen modelo mental

Un buen modelo mental te ayuda a ser consistente, porque lo importante es hacer buenas soluciones.

- Que no sea lo *más eficiente*™ no lo hace lento

State

Programación

State... ¿qué es un estado?

La representación de nuestra aplicación, sistema o entidad en un momento dado.

- Salario en mi cuenta bancaria
- Si una persona tiene hambre o no
- ¿Has aceptado las cookies?

Mutaciones de estado

Las mutaciones de estado no están muy bien vistas en un contexto funcional.

Se prefieren estructuras de datos inmutables

- Acordémonos de la **transparencia referencial**

Peeeeeero, son el punto justo de corrupción para que el paradigma nos siga siendo cómodo

State Machines

State machines

También llamadas Máquinas de Estado Finitas (porque tienen un número concreto y definido de posibles acciones) y abreviadas como FSM (en inglés)

En vez de resolver problemas, resuelves comportamientos.

Cuando las entidades se encuentran en estados, podemos realizar acciones concretas

XState

Creado por David Khourshid, tipado por Matt Pocock

Máquinas de estado finitas en JavaScript

- con integraciones para React, Vue y Svelte

DAVID KHOURSHID

Goodbye, useEffect

REACTATHON 2022 IN THE PARK

Statecharts

Statecharts

¡¡Vivan los diagramas!!

Un statechart nos permite visualizar **flujos de estado** de una *máquina de estado*

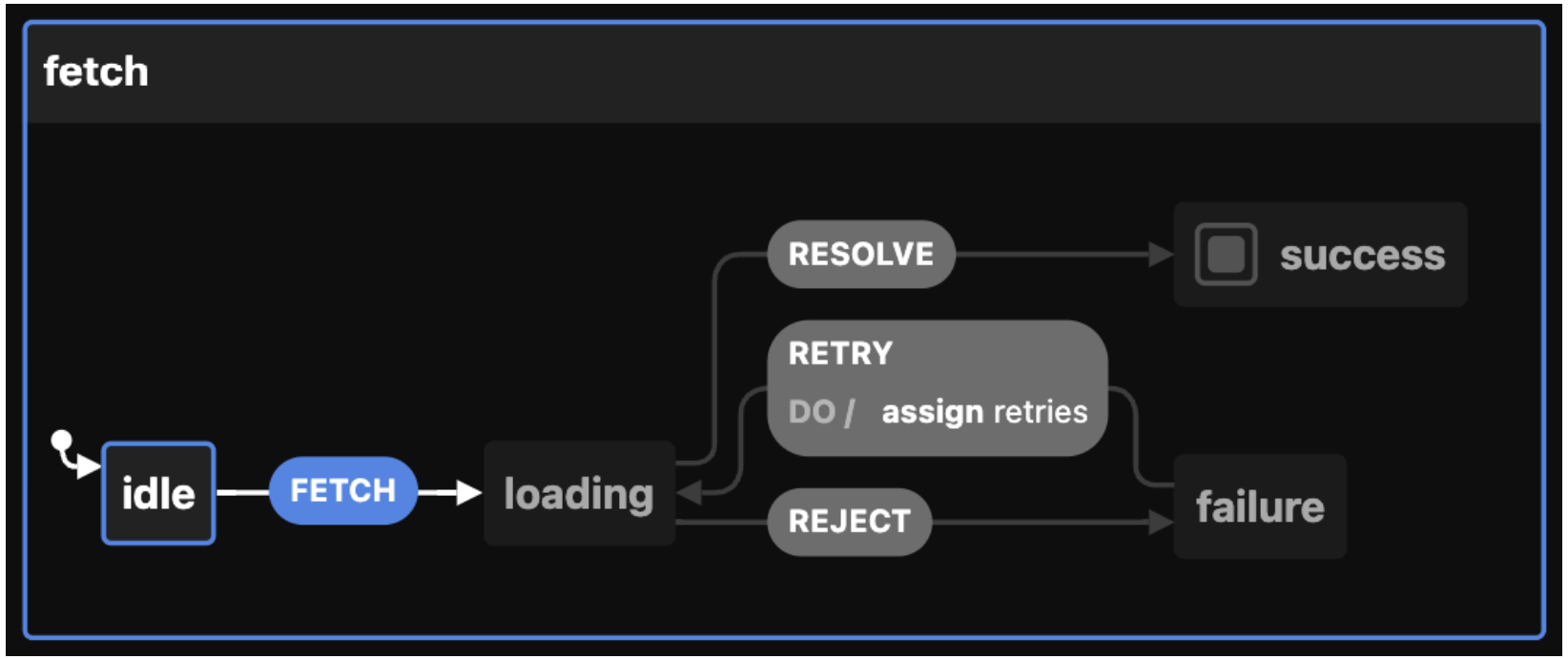
“A visual formalism for complex systems”

Harel, 1987 – statecharts.dev

XState Visualizer

StateCharts usando Diagram as Code (JavaScript con XState)

https://stately.ai/viz

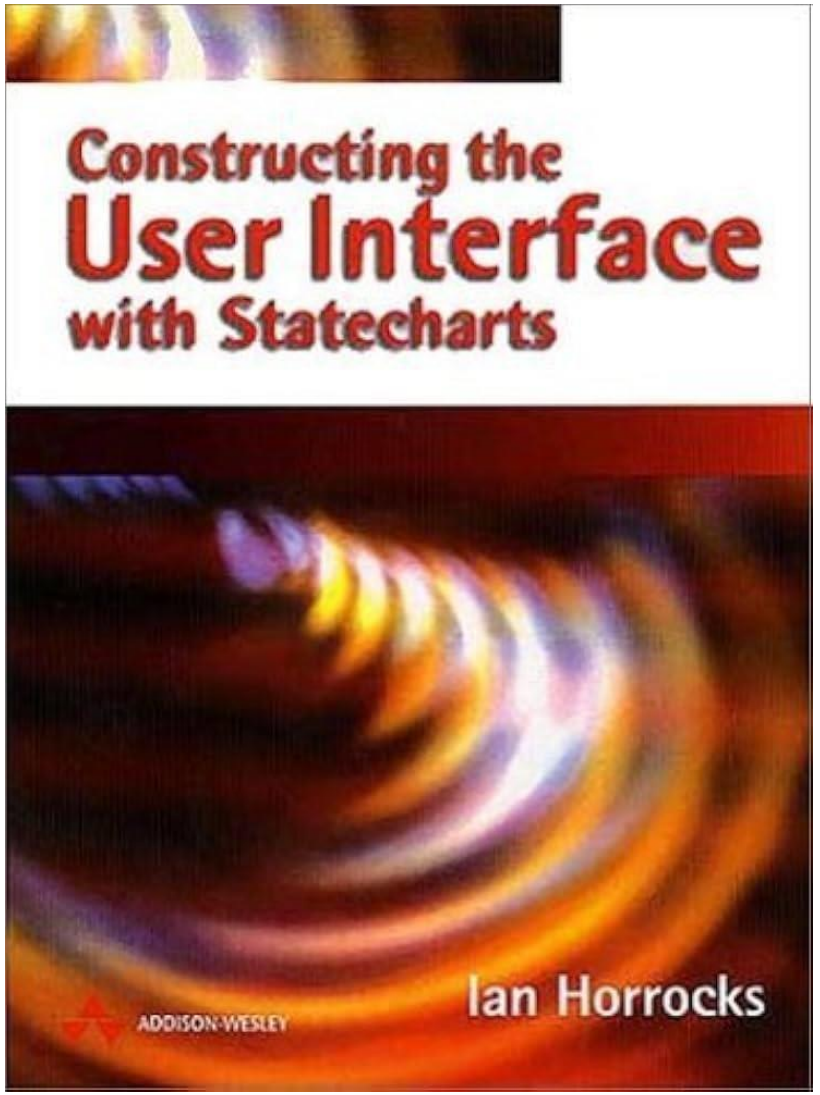

fetch
idle
FETCH
loading
RESOLVE
success
RETRY
DO / assign retries
REJECT
failure

UI <3 Statecharts

El libro está *descatalogado*…

Pero está disponible online (dicen claro, yo no lo sé)

Es la propuesta de idea de statecharts como definición de UIs



Lo que no te he contado

Eran demasiadas cosas

- Actor model -> modelo mental para concurrencia
  - https://stately.ai/docs/actor-model
- Functors, Monads, MonadTransformers, etc.
- Estructuras de Datos inmutables (casos de uso, cómo funcionan, beneficios)

Conclusión

Conclusión

- Programación Funcional busca que todo se represente en ecuaciones
  - Pero no siempre va a tener sentido

Conclusión

- Programación Funcional busca que todo se represente en ecuaciones
  - Pero no siempre va a tener sentido
- Son conceptos y modelos mentales, no librerías concretas

Conclusión

- Programación Funcional busca que todo se represente en ecuaciones
  - Pero no siempre va a tener sentido
- Son conceptos y modelos mentales, no librerías concretas
- Es cierto que algunos lenguajes son más funcionales que otros

Lenguajes funcionales

- Haskell
- Erlang
- OCaml

Flavor de JavaScript funcional

- **PureScript**

La buena solución

La que parece fácil y obvia en retrospectiva

“Si una solución parece innecesariamente difícil seguramente es que lo sea”

SABER MÁS

Anécdota del software

More shell, less eggs

A veces perdemos el foco del problema para centrarnos en divagaciones…

Artículo: https://leancrew.com/all-this/2011/12/more-shell-less-egg/

Bibliografía

JJS

Explore the JavaScript Universe

Rebuild your mental model from the inside out.

WITH
Dan Abramov & Maggie Appleton

[JSConf] Programación Funcional
Anjana Vakil

QR de las slides

https://github.com/jofaval/talks-about/tree/master/tech-talks/valencia-js/programacion-funcional

Encuéntrame en

- Linkedin - **linkedin.com/in/jofaval/**
- Github - **github.com/jofaval**

Programación Funcional

Conceptos de JavaScript

Gracias por la atención

Preguntas