

MASTERING SHINY

capítulos 13-14-15

Encuentro 9



CAPÍTULO 13 :

- ¿POR QUÉ ES NECESARIO EL MODELO DE PROGRAMACIÓN REACTIVA?
- HISTORIA DE LA PROGRAMACIÓN REACTIVA FUERA DE R.

CAPÍTULO 14:

DETALLES DEL GRAFO REACTIVO:

- DETERMINA EXACTAMENTE CUÁNDO SE ACTUALIZAN LOS COMPONENTES REACTIVOS.

CAPÍTULO 15:

LOS COMPONENTES BÁSICOS SUBYACENTES:

- LOS OBSERVADORES
- LA INVALIDACIÓN CRONOMETRADA.

"Hacemos mucha magia, pero es magia buena: eso significa que se puede descomponer en componentes simples y coherentes." — Tom Dale



¿Por qué necesitamos programación reactiva?

La programación reactiva es un estilo de programación que se centra en valores que cambian:

- con el tiempo
- en los cálculos
- en acciones que dependen de ellos

Es importante para las aplicaciones Shiny porque son interactivas:

- expresiones y salidas reactivas deben actualizarse solo si sus entradas cambian
- las salidas deben mantenerse sincronizadas con las entradas
- no deben trabajar más de lo necesario

¿Por qué no se pueden utilizar variables?

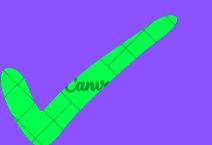
- Las variables en R representan valores y pueden cambiar con el tiempo
- Pero nunca cambian automáticamente.

```
13 - ````{r}
14  temp_c<-10
15  temp_f<- (temp_c*9/5) +32
16  temp_f
17
18
19 #Cambiamos el valor de temp_c pero no volvemos a calcular temp_f
20 temp_c <- 12
21 temp_f
22
```

¿Qué pasa con las funciones?

Resuelve el primer problema de la reactividad:

- cada vez que se accede a `temp_f()` **se obtiene el último cálculo**
- Pero **no minimiza el cálculo** (cada vez que se llama a `temp_f()` se vuelve a calcular, incluso si `temperatura_c` no ha cambiado)



```
27 - ``{r}
28   temp_c <- 10
29 - temp_f <- function() {
30   message("Converting")
31   (temp_c * 9 / 5) + 32
32 - }
33   temp_f()
34 - ````
```

```
Converting
[1] 50
```

```
35 - ``{r}
36   temp_c <- -3
37   temp_f()
38 - ````
```

```
Converting
[1] 26.6
```

13

Event-driven programming

```
42+ ``{r}
43 # Creamos una clase llamada DynamicValue usando R6.
44 # Esta clase sirve para guardar un valor y una función que se ejecuta cuando ese valor cambia.
45 DynamicValue <- R6::R6Class("DynamicValue", list(
46   value = NULL,           # donde se guarda el valor actual
47   on_update = NULL,       # función que se dispara cuando se actualiza el valor
48
49 # Método para obtener el valor
50 get = function() self$value,
51
52 # Método para cambiar el valor. Si hay una función registrada en on_update,
53 # se ejecuta cuando se cambia el valor.
54 set = function(value) {
55   self$value <- value
56   if (!is.null(self$on_update))
57     self$on_update(value)
58   invisible(self)
59 },
60
61 # Método para registrar una función que se ejecuta cuando cambia el valor
62 onUpdate = function(on_update) {
63   self$on_update <- on_update
64   invisible(self)
65 }
66 ) )
67
```

- Es un **paradigma** simple: se registran funciones de *callback* que se ejecutarán en respuesta a eventos.

- Usando R6 definimos un *DynamicValue* que tiene tres métodos:

1. `get()`
2. `set()`
3. `onUpdate()`

acceder y cambiar el valor subyacente

registrar el código que se ejecutará al modificar el valor

```
71+ ``{r}
72 # Creamos un objeto temp_c que va a ser un valor "dinámico"
73 temp_c <- DynamicValue$new()
74
75 # Registraremos una función que se va a ejecutar automáticamente cada vez que cambiemos temp_c
76 # Esa función va a actualizar temp_f haciendo la conversión a Fahrenheit
77 temp_c$onUpdate(function(value) {
78   message("Converting")
79   temp_f <- (value * 9 / 5) + 32
80 })
81 ...
82
83
84+ ``{r}
85 # Cambiamos el valor de temp_c. Automáticamente se actualiza temp_f gracias a la función anterior
86 temp_c$set(10)
87 temp_f
88
```

Converting
[1] 50

- Soluciona el problema de los cálculos innecesarios

- Pero introduce uno nuevo: hay que controlar cuidadosamente qué entradas afectan a qué cálculos.

Programación reactiva

- Resuelve ambos problemas
- Una expresión reactiva rastrea **automáticamente** todas sus **dependencias**.
- Pero si no hay cambios, **no hay recálculos**.

Las expresiones reactivas en Shiny tienen dos superpoderes:

- **Perezosas (lazy)**: solo hacen el trabajo cuando se las necesita.
- **En caché (cached)**: si el input no cambió, devuelven el resultado anterior sin recalcular.



```
97  ````{r}
98  library(shiny)
99  reactiveConsole(TRUE) # hace posible experimentar en la consola
100 ````

Aviso: package 'shiny' was built under R version 4.4.3

102 ````{r}
103 #inicializa el valor reactivo con temp_c = 10
104 temp_c <- reactiveVal(10) # create
105 temp_c() # get
106 ````

[1] 10

109 ````{r}
110 #reactualiza el valor
111 temp_c(20) # set
112 temp_c() # get
113 ````

[1] 20
```

```
117 ````{r}
118 #genera una expresión reactiva para calcular temp_f
119
120 temp_f <- reactive({
121 message("Converting")
122 (temp_c() * 9 / 5) + 32
123 })
124 temp_f()

Converting
[1] 68

126 ````{r}
127 #Actualiza el valor de temp_c y de manera automática se actualiza temp_f
128 temp_c(-3)
129 temp_c(-10)
130 temp_f()
131 ````

Converting
[1] 14

135 ````{r}
136 #Al no haber nuevas actualizaciones de temp_c, temp_f no cambia pero tampoco se recalcula.
137 temp_f()
138 ````

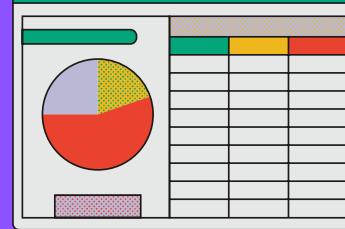
[1] 14
```

1
2
3

Breve historia de la programación reactiva

“Imaginé una pizarra mágica en la que si borrabas un número y escribías algo nuevo, todos los demás números cambiarían automáticamente, como un procesador de textos con números.”

Dan Bricklin



1979

VisiCalc
(Dan Bricklin)

“Word processor with numbers”

Primera intuición de reactividad con hojas de cálculo.



1990s

FRAN
(Functional Reactive Animation)

Primer intento académico de formalizar la reactividad en programación funcional.



2010s

Frameworks web JS
(Knockout, Ember, Meteor, etc.)

Shiny como parte de esta revolución: reactividad como solución concreta para la UI

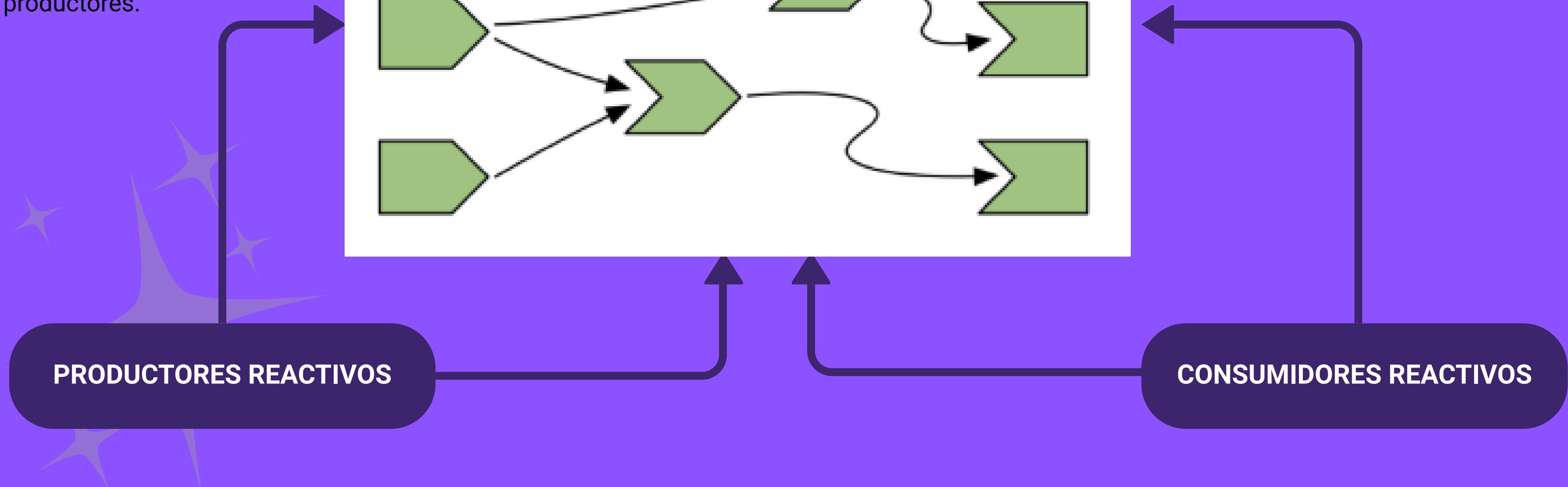
Cuando este libro habla de programación reactiva, lo hace desde la perspectiva específica de Shiny: un enfoque práctico, diferente a otras corrientes como ReactiveX o FRP.

El grafo reactivo

Un recorrido paso a paso por la ejecución reactiva

14

- Los componentes son direccionales.
- Algunos consumidores dependen de uno o más productores.

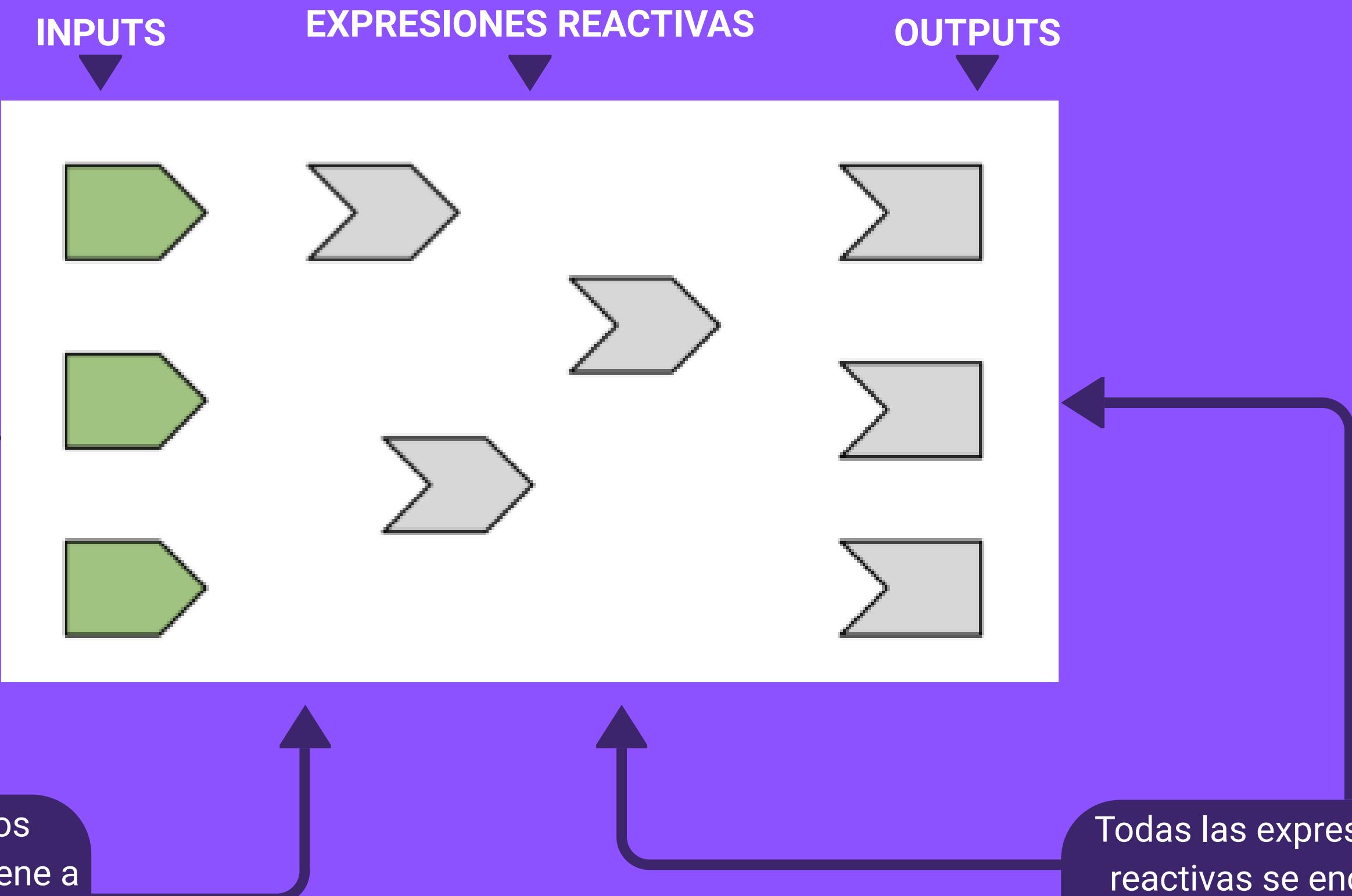


Comienza una sesión

14

- Estado inicial tras la carga de la aplicación

Las entradas reactivas están listas (verde), lo que indica que sus valores están disponibles para el cálculo.

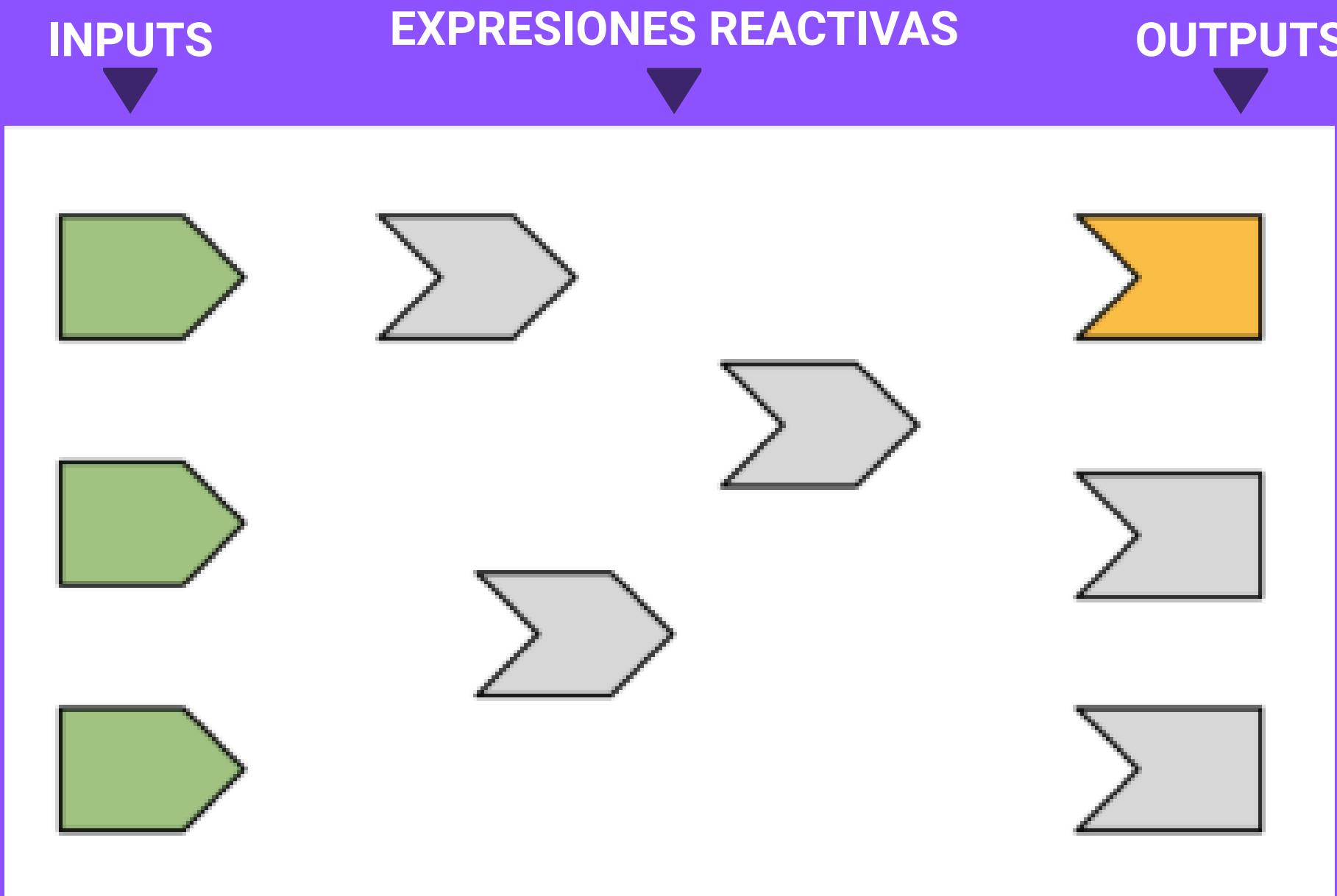


No hay conexiones entre los elementos porque Shiny no tiene a priori conocimiento de las relaciones entre reactivos.

Todas las expresiones y salidas reactivas se encuentran en su estado inicial **invalidado** (gris), lo que significa que aún no se han ejecutado.

Comienza la ejecución

- Shiny comienza a ejecutar un observador/salida invalidado de forma arbitraria (naranja)



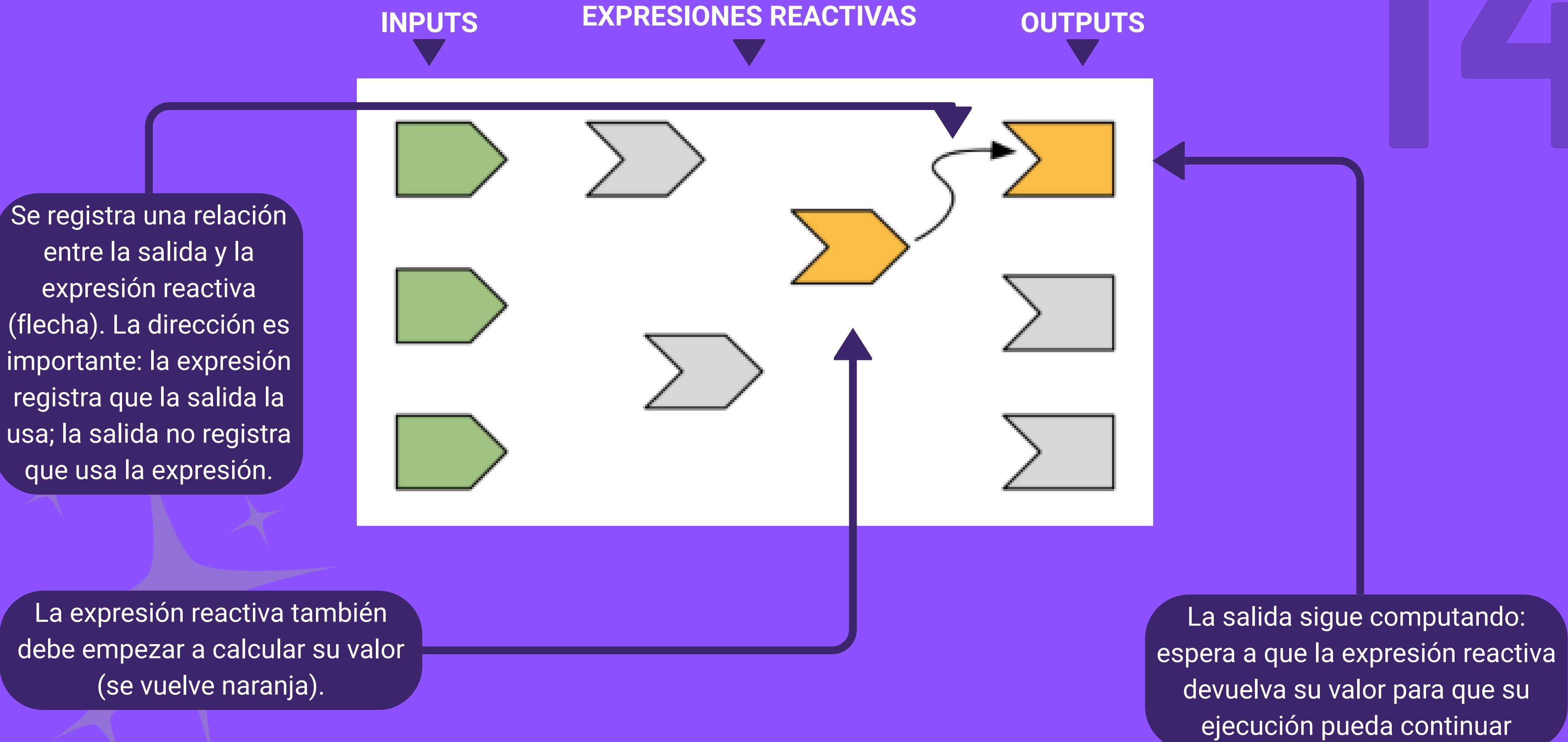
A los observadores y salidas no debería importarles el orden de ejecución, porque han sido diseñados para funcionar de forma independiente.

14

Leyendo una expresión reactiva

- La salida necesita el valor de una expresión reactiva, por lo que comienza a ejecutar la expresión.

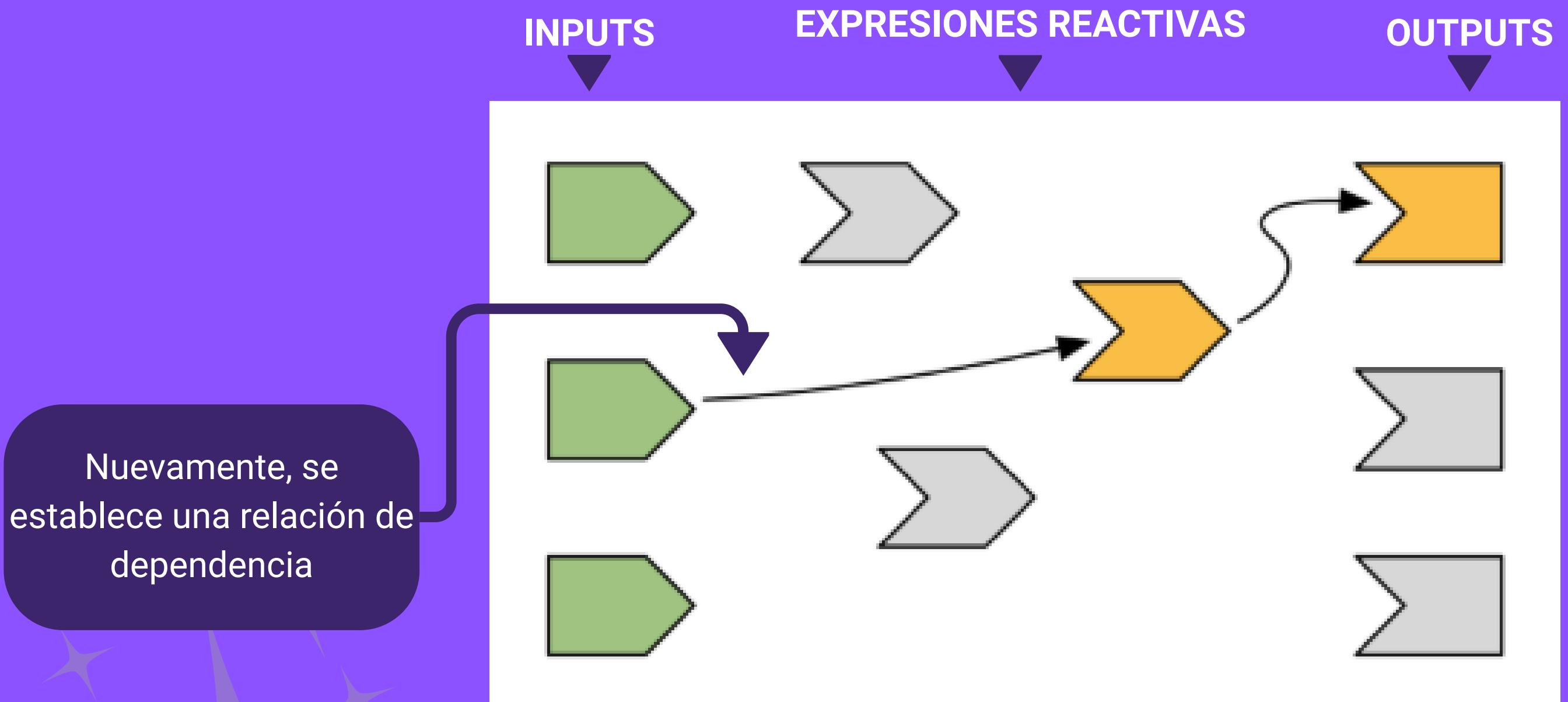
14



Leyendo una entrada

- Esta expresión reactiva en particular lee una entrada reactiva

14

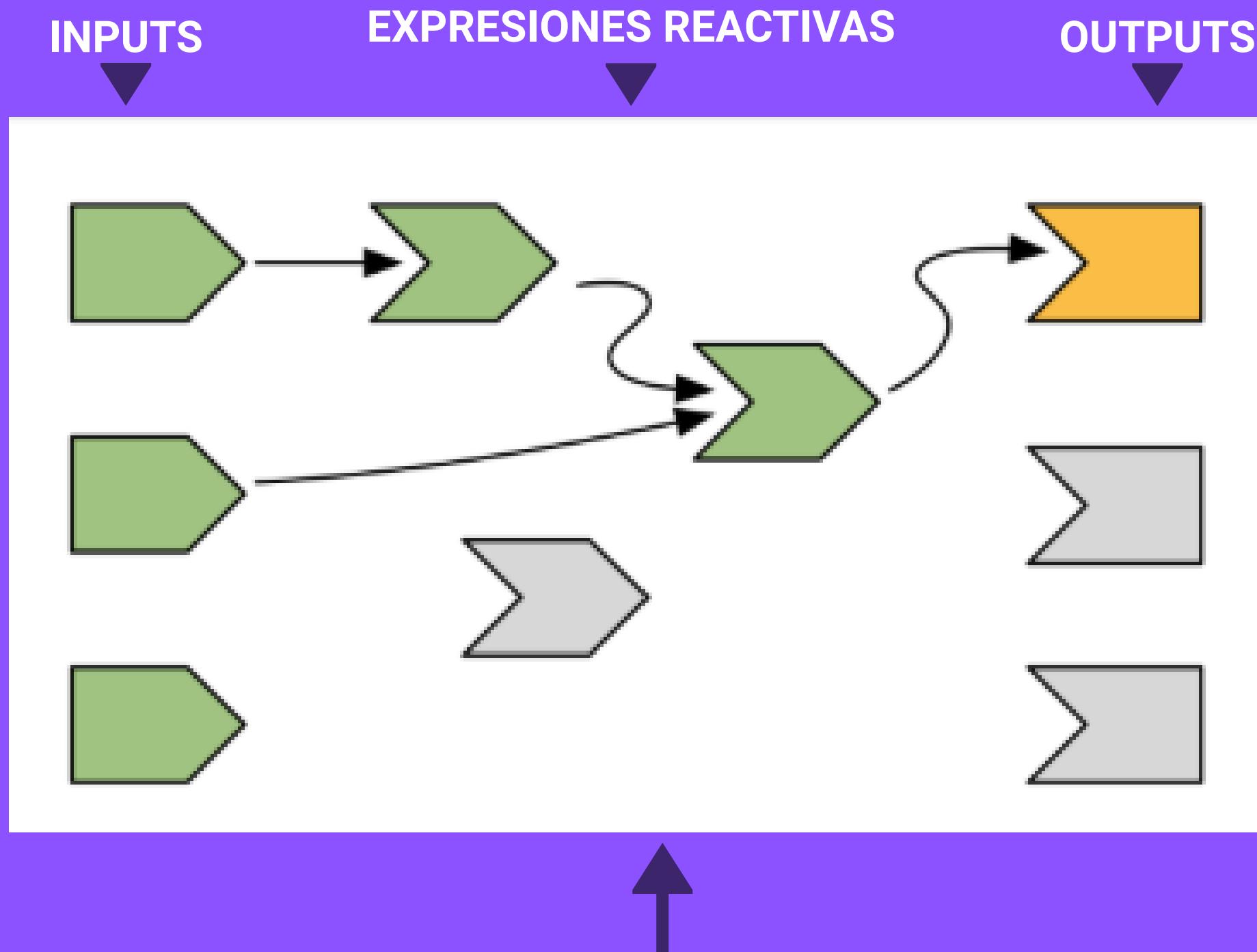


Las entradas reactivas no tienen
nada que ejecutar, por lo que
pueden regresar inmediatamente.

La expresión reactiva se completa

- La expresión reactiva ha terminado de calcularse (se vuelve verde).

14

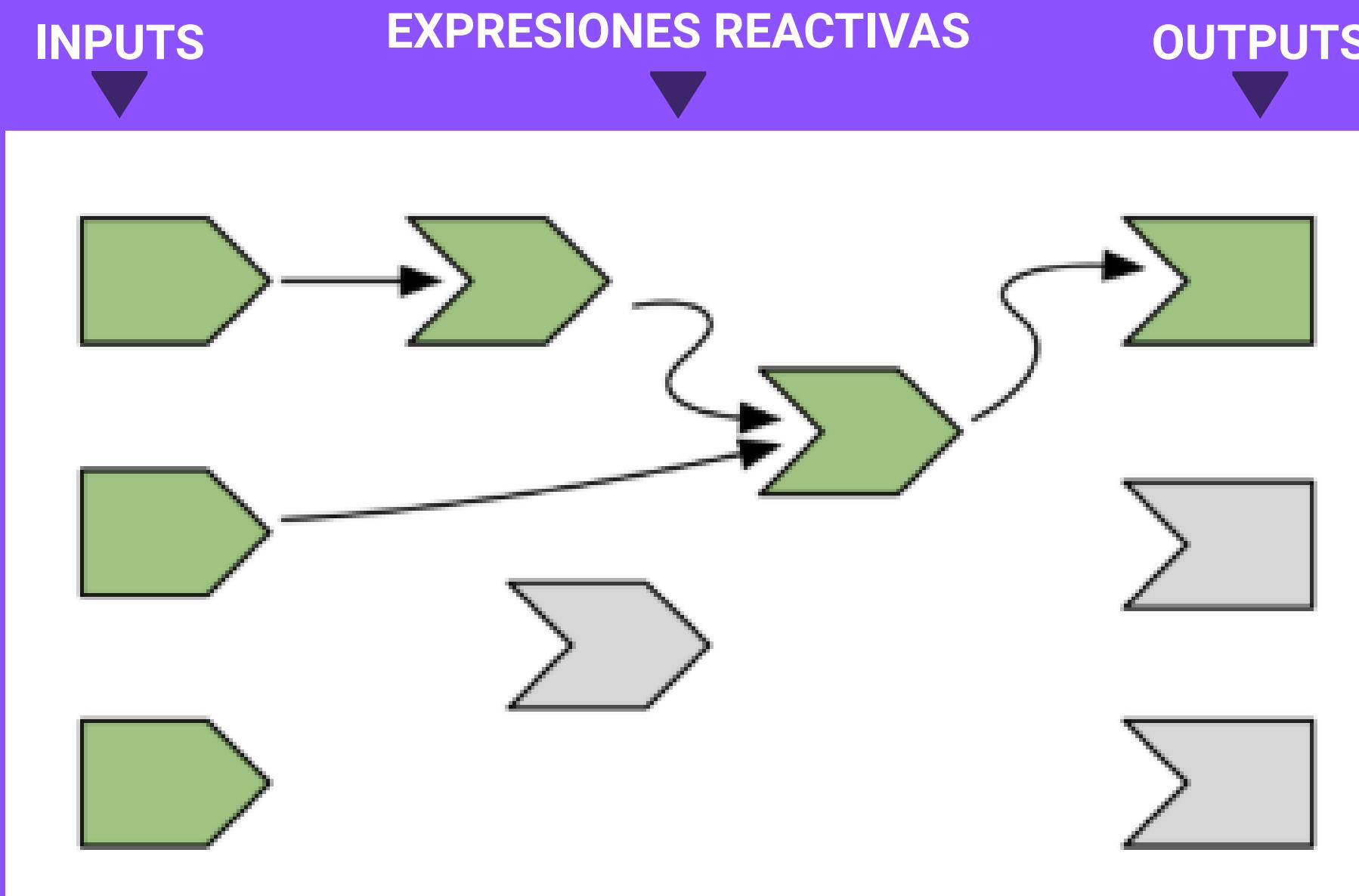


Una vez que la expresión reactiva ha terminado de ejecutarse, se vuelve verde para indicar que está lista. Almacena el resultado en caché, por lo que no necesita recalcularlo a menos que cambien sus entradas.

La salida se completa

- La salida ha finalizado el cálculo y se vuelve verde.

14

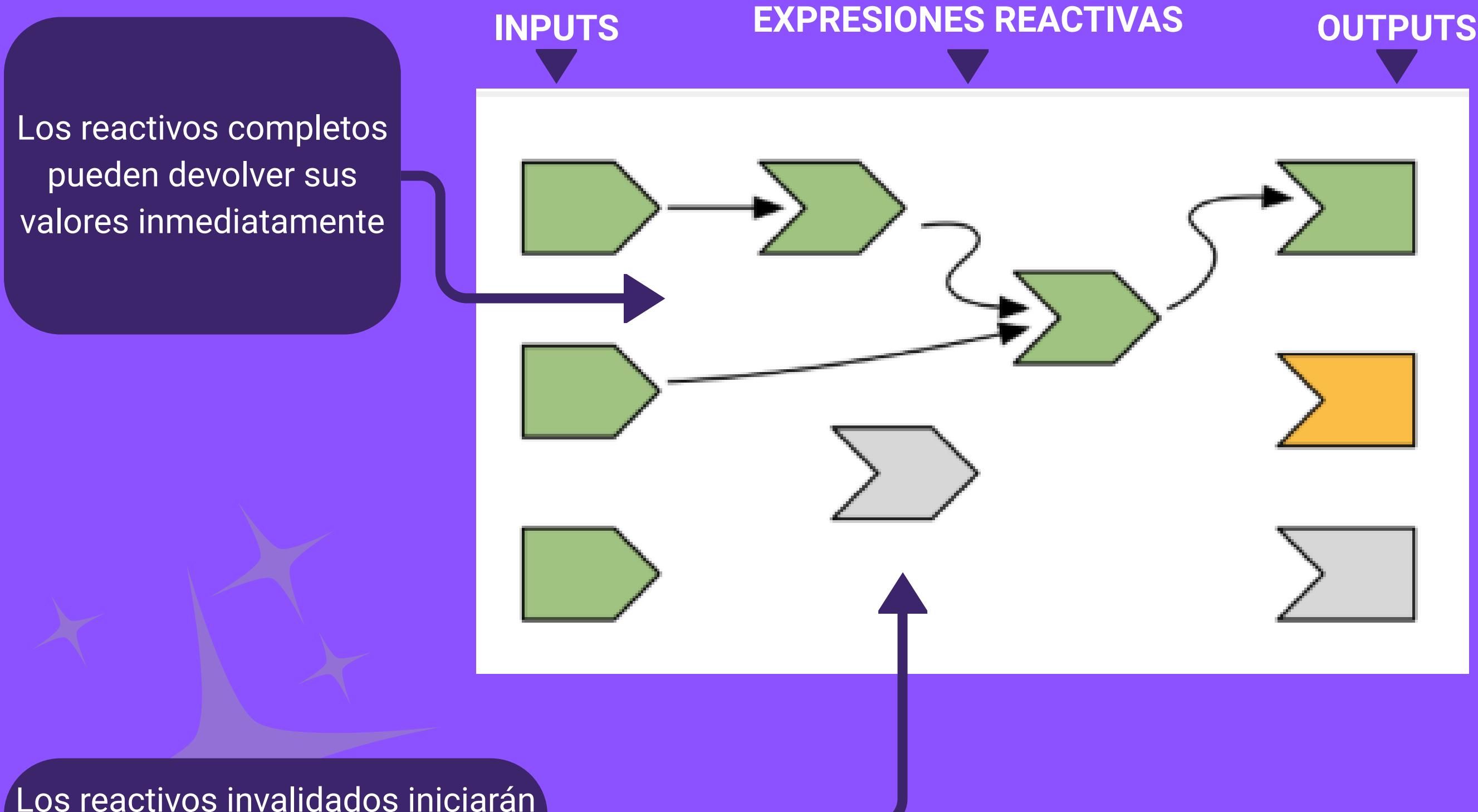


Se completa la salida

La siguiente salida se ejecuta

- La siguiente salida comienza a calcularse y se vuelve naranja.

14



Este ciclo se repetirá hasta que todas las salidas invalidadas alcancen el estado completo (verde).

La ejecución se completa, las salidas se vacían

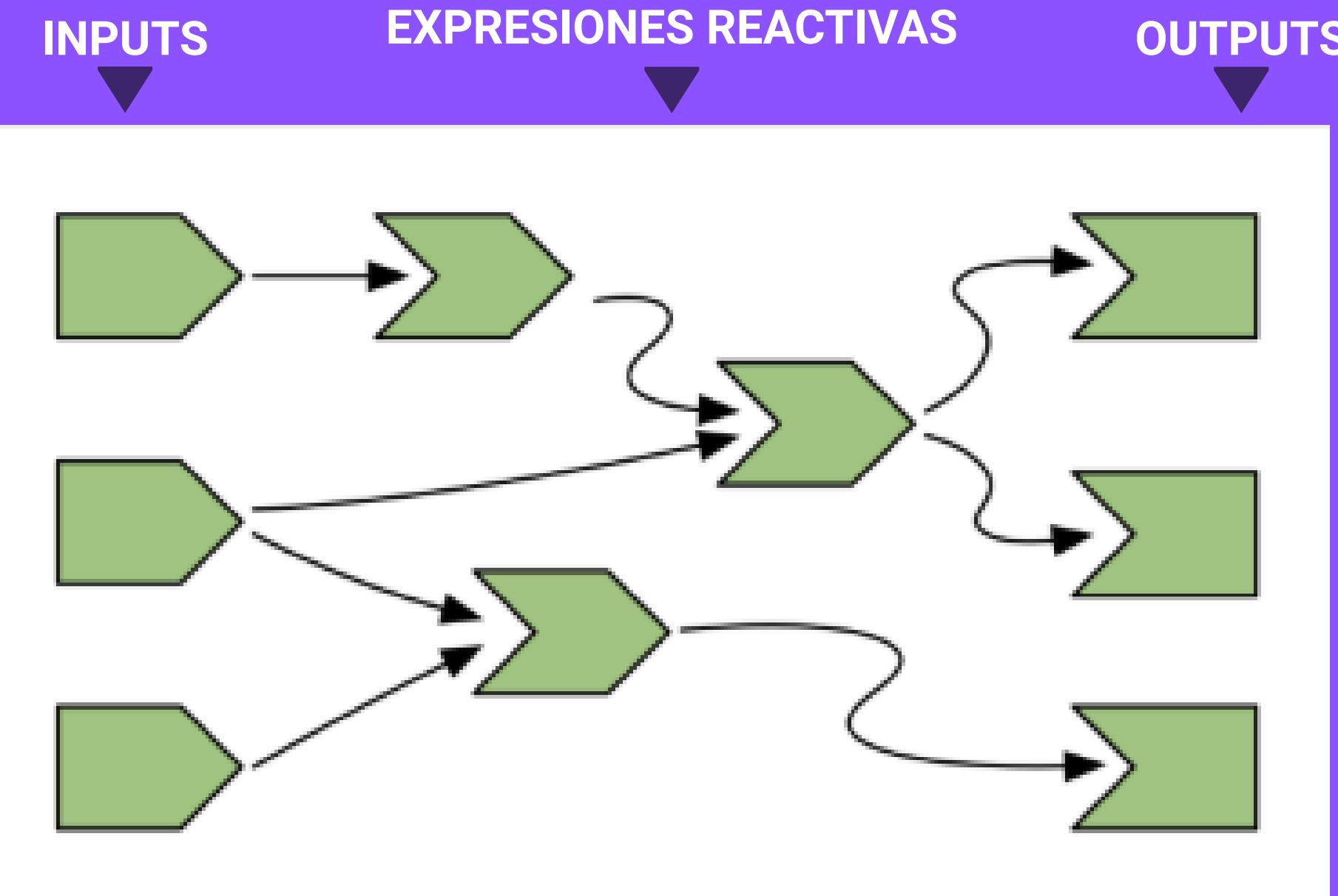
- Todas las expresiones de salida y reactivas han finalizado y se han vuelto verdes.

14

El ciclo de ejecución reactiva ha finalizado.

No se realizará ningún trabajo hasta que alguna fuerza externa actúe sobre el sistema

En términos reactivos, esta sesión ahora está en **REPOSO**.



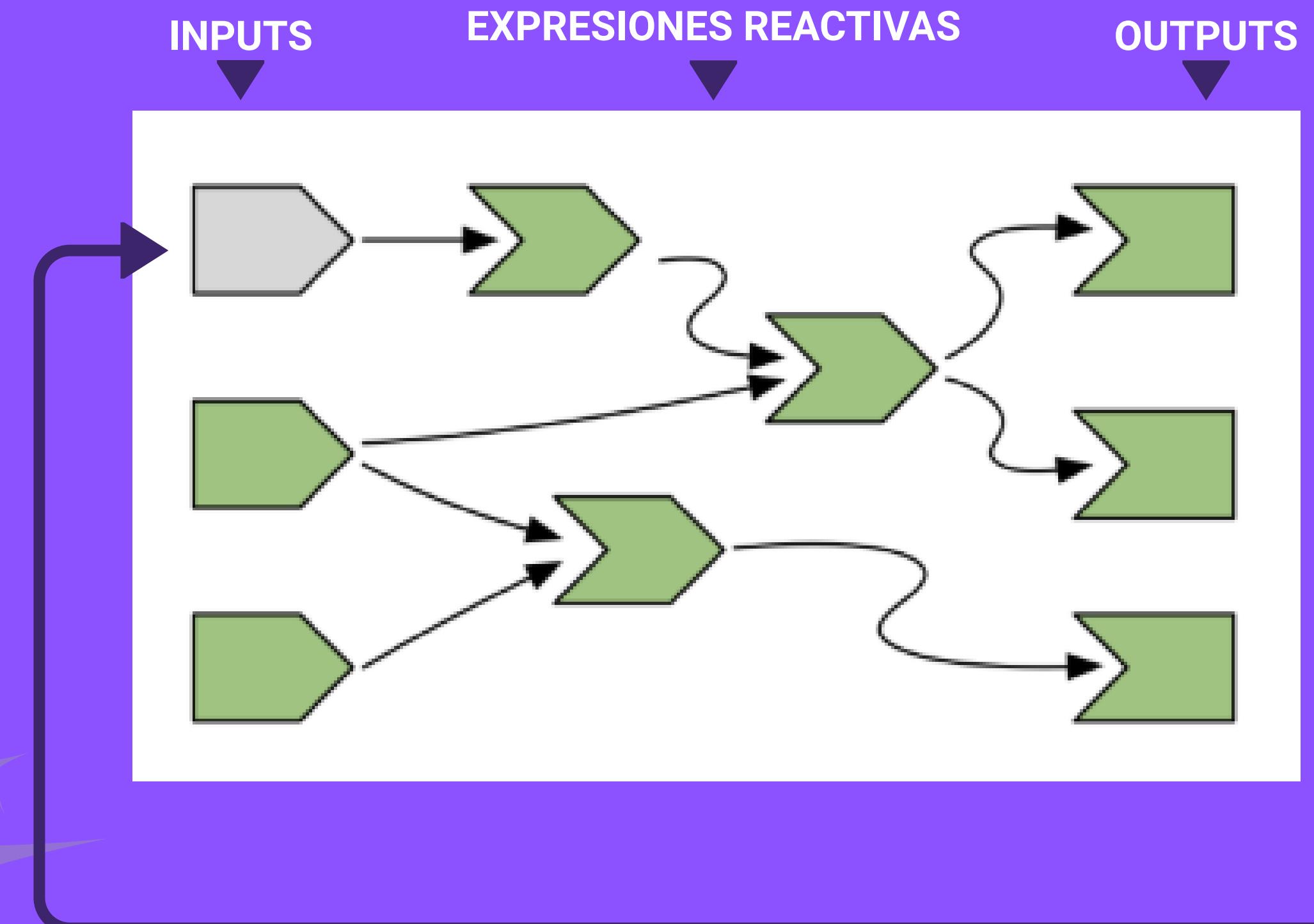
Descubrimos las **RELACIONES** entre los objetos reactivos: cuando una entrada reactiva cambia, sabemos exactamente qué reactivos debemos actualizar.

Una entrada cambia

- Al cambiar una entrada, el navegador envía un mensaje a la función del servidor, indicando a Shiny que actualice la entrada reactiva correspondiente.

- Esto inicia una **FASE DE INVALIDACIÓN:**

1. **invalidar la entrada**
2. **notificar las dependencias**
3. **eliminar las conexiones existentes**



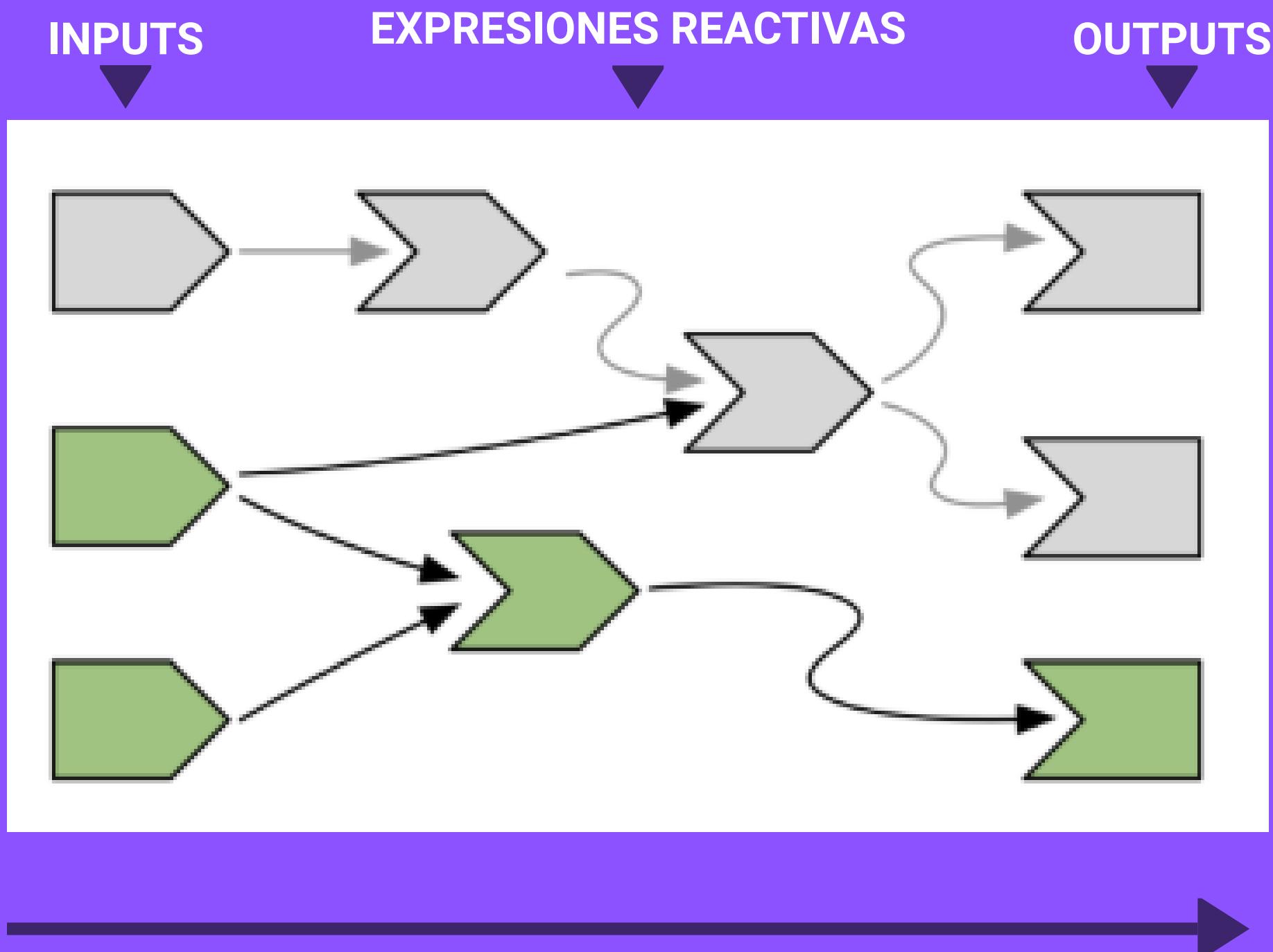
Invalidando las entradas

El usuario interactúa con la aplicación, invalidando una entrada.

Notificar dependencias

- La invalidación se emite desde la entrada, siguiendo cada flecha de izquierda a derecha.

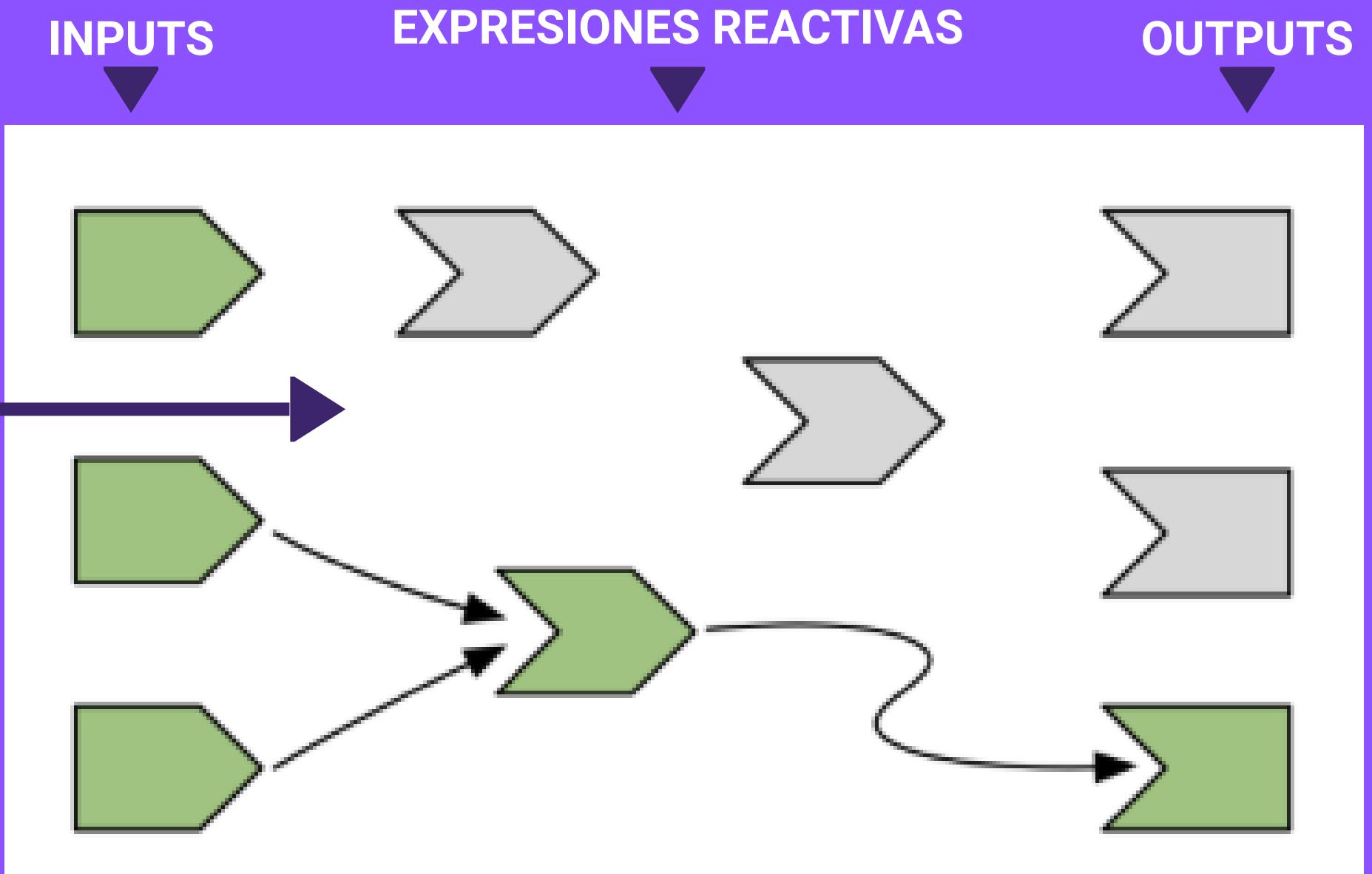
14



Invalidación de izquierda a derecha

Eliminando relaciones

- Los nodos invalidados olvidan todas sus relaciones anteriores para que puedan ser descubiertos de nuevo.



Cuando un nodo se invalida, Shiny borra todas sus flechas – las que salen y las que entran.

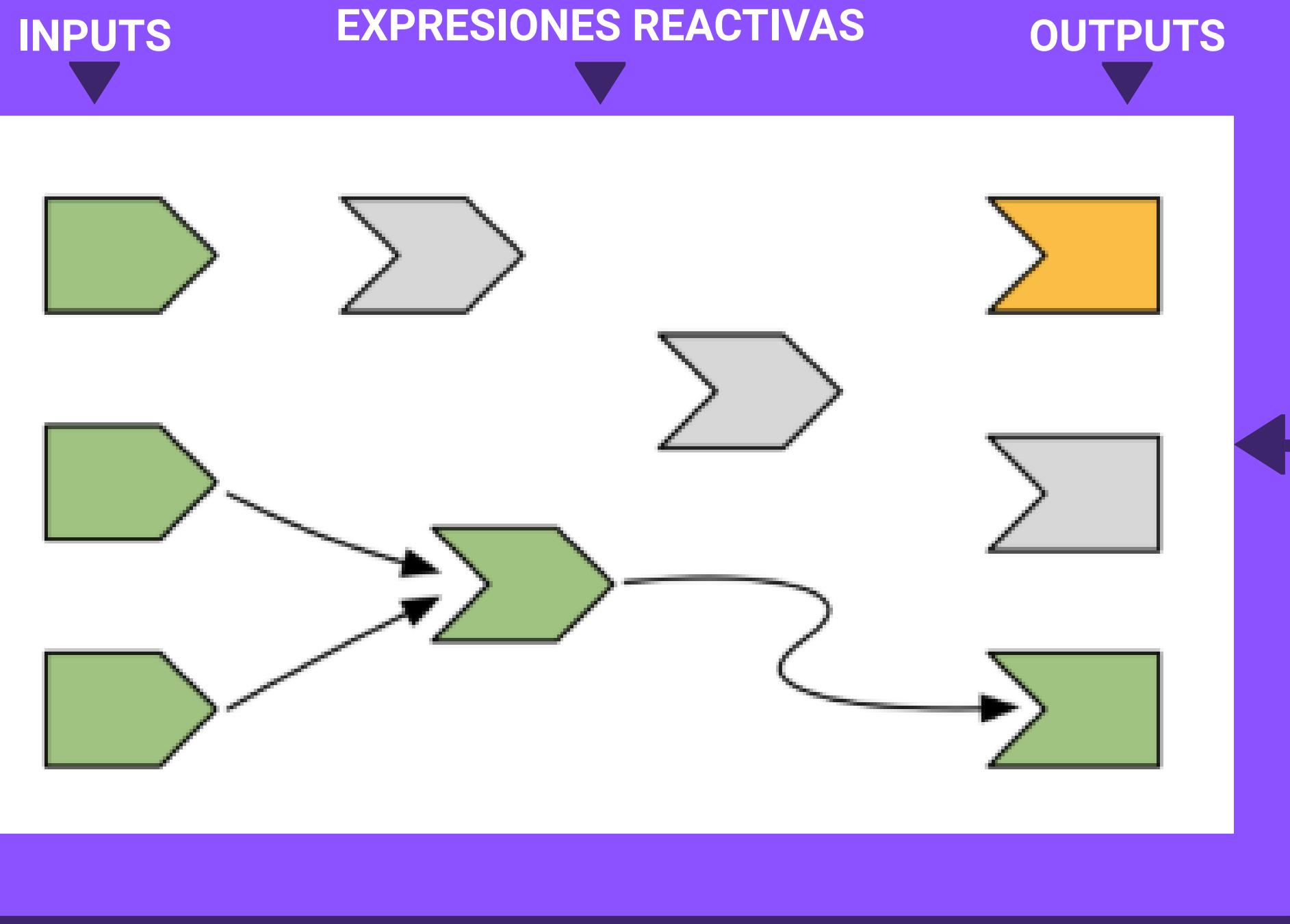


Incluso si otras expresiones no cambiaron, el nodo invalidado ya **no necesita recordar quién lo activó**: esas relaciones serán **redescubiertas** cuando el nodo se reejecute. Así se evita que el grafo conserve dependencias viejas o innecesarias.

14

Re-ejecución

- Ahora la re-ejecución procede de la misma manera que la ejecución, pero hay menos trabajo por hacer ya que no estamos empezando desde cero.



Nuevamente hay una combinación de reactivos válidos e inválidos. Se procede igual: ejecutar las salidas invalidadas una a una



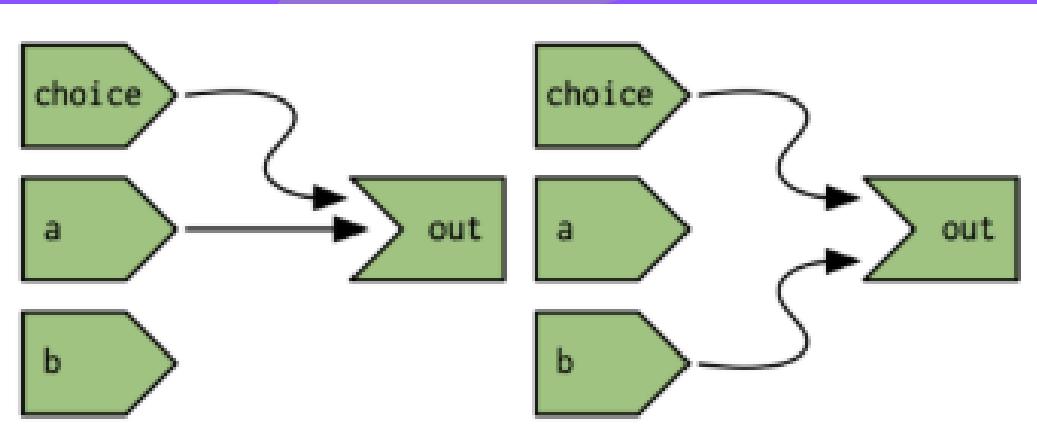
El resultado final será un grafo reactivo en reposo.

Shiny ha realizado el mínimo trabajo posible: solo hizo lo necesario para actualizar las salidas afectadas por las entradas modificadas.

Dinamismo

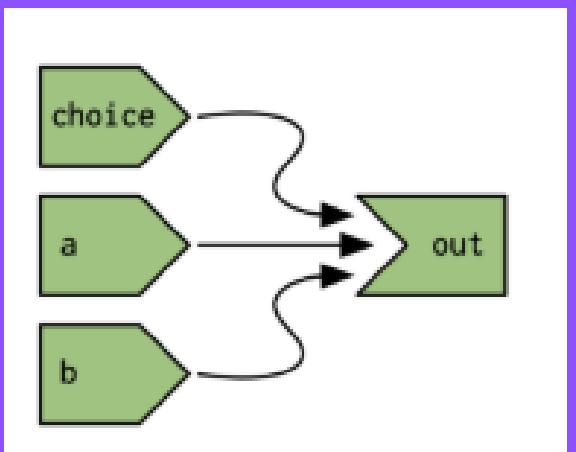
Shiny reconstruye dinámicamente el grafo reactivo cada vez que una salida es invalidada. Esto permite que el sistema descubra nuevamente las dependencias activas en tiempo real y garantiza que se realice sólo el trabajo necesario cuando cambian los inputs.

```
231
232 - ``{r}
233  ui <- fluidPage(
234  selectInput("choice", "A or B?", c("a", "b")),
235  numericInput("a", "a", 0),
236  numericInput("b", "b", 10),
237  textOutput("out")
238 )
239 - server <- function(input, output, session) {
240  output$out <- renderText({
241 - if (input$choice == "a") {
242  input$a
243 - } else {
244  input$b
245 - }
#el condicional lo hace más e
246 - })
247 - }
248
249 shinyApp(ui, server)
250 -
```



```
253 - ``{r}
254  ui <- fluidPage(
255  selectInput("choice", "A or B?", c("a", "b")),
256  numericInput("a", "a", 0),
257  numericInput("b", "b", 10),
258  textOutput("out")
259 )
260 - server <- function(input, output, session) {
261  output$out <- renderText({
262  a <- input$a
263  b <- input$b
264 - if (input$choice == "a") {
265  a
266 - } else {
267  b
268 - }
#el output depende de las tres entrad
269 - })
270 - }
271 shinyApp(ui, server)
272 -
```

Un mínimo cambio en el código puede hacer que una salida dependa de más inputs de los realmente utilizados. En Shiny, la dependencia reactiva se establece cuando se lee el valor, no cuando se lo usa. Por eso, leer todos los inputs antes de decidir cuál mostrar conecta innecesariamente al grafo completo.

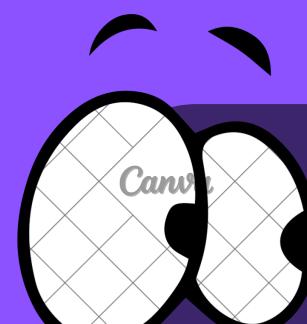


El paquete reactlog

14

```
273 ~`{r}
274 install.packages("reactlog")
275
276 ~`{r}
277 reactlog::reactlog_enable()
278
279 ~`{r}
280 ui <- fluidPage(
281   selectInput("choice", "A or B?", c("a", "b")),
282   numericInput("a", "a", 0),
283   numericInput("b", "b", 10),
284   textOutput("out")
285 )
286
287 server <- function(input, output, session) {
288   output$out <- renderText({
289     if (input$choice == "a") {
290       input$a
291     } else {
292       input$b
293     }
294   })
295 }
296
297 shinyApp(ui, server)
298 ~`{r}
299 shiny::reactlogShow()
300
301 ~`{r}
302 shiny::reactlogShow()
303 ~`{r}
304 shiny::reactlogShow()
```

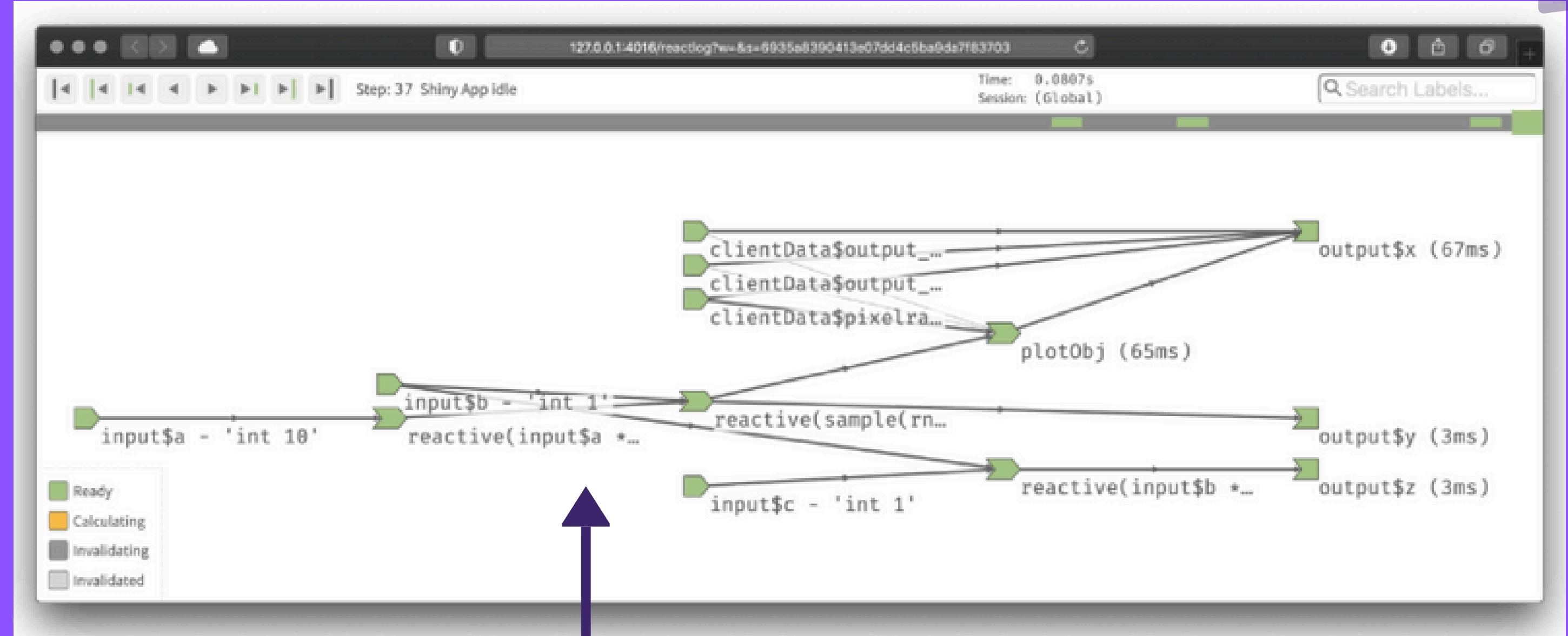
1. Instalar reactlog
2. Activarlo
3. Código de tu shiny app
4. Visualizar el gráfico:
 - a. Mientras se ejecuta
Ctrl + F3 (Windows)
Cmd + F3 (Mac)
 - b. Al finalizar la app



Con reactlog podés observar en tiempo real:
🔗 Cómo se crean las dependencias entre inputs y outputs.

⌚ En qué orden se invalidan y reactivan los nodos.
🧠 Cómo Shiny evita recálculos innecesarios, eligiendo solo los caminos relevantes.

El paquete reactlog



En el gráfico aparecen tres inputs extra (`clientData$output_x_height`, `clientData$output_x_width`, `clientData$pixelratio`) que son dependencias implícitas de los outputs gráficos, como `renderPlot()`, por ejemplo el cambio de tamaño de la ventana. **Shiny los considera como inputs reactivos invisibles que afectan al gráfico.**

Por defecto, los nodos como `reactive({...})` o `observe({...})` no tienen nombre explícito. Se pueden etiquetar para identificarlos:

```
reactive({  
  # código acá  
}, label = "nombre")
```

Bloques de construcción reactivos

La programación reactiva tienen tres componentes fundamentales:

- **valores reactivos**
- **expresiones reactivas**
- **observadores (las salidas como un tipo especial de observador)**

Dos herramientas para controlar el grafo reactivo:

- **el aislamiento**
- **la invalidación temporizada.**

Valores reactivos

```
310  
317 <--{r}  
318 #reactiveval()  
319  
320 x <- reactiveval(10)  
321 x() # get  
322  
[1] 10  
  
323  
324 <--{r}  
325 x(20) # set  
326 x() # get  
327  
[1] 20  
  
328  
329 <--{r}  
330 #reactivevalues()  
331  
332 r <- reactivevalues(x = 10)  
333 r$x # get  
334  
[1] 10  
  
335  
336 <--{r}  
337 r$x <- 20 # set  
338 r$x # get  
339  
[1] 20
```

Hay dos tipos:
Un único valor reactivo

↓
reactiveVal()

Una lista de valores
reactivos

↓
reactiveValues()

Interfaces diferentes
Sintaxis distintas
Comportamientos iguales

Ambos tipos de valores reactivos tienen
semántica por referencia

Valores reactivos

Semántica de copia en R

vs

Semántica de referencia en Shiny

Expresiones reactivas

15

Errores en Shiny: comportamiento reactivo

Los errores dentro de expresiones reactivas se cachean como si fueran valores normales.

Se propagan por el grafo respetando la lógica de dependencias.

Si no cambia ningún input, se vuelve a usar el error cacheado, sin re-evaluar.



¿Qué pasa cuando el error llega a...?

NODO	COMPORTAMIENTO
<code>output\$...</code>	Se muestra el mensaje de error en la app
<code>observe({...})</code>	Cierra la sección si no se controla

Para evitar que un observe tumbe la app, se recomienda envolverlo con `try()` o `tryCatch()`



Expresiones reactivas

15

Funciones para cortes suaves

`on.exit() dentro de reactive({...})`

CONDICIÓN	MÉTODO	RESULTADO EN LA APP
Corte brusco	<code>stop("mensaje")</code>	Aparece un error visible
Corte suave	<code>req(condición)</code>	El output se borra
Corte suave con retención	<code>req(condición, cancelOutput = TRUE)</code>	El output se conserva

`reactive({...})` es una función “especial” → tiene las mismas reglas que cualquier función en R, pero con caché y pereza.

Se puede usar `on.exit({...})` dentro de una expresión reactiva para ejecutar algo cuando se termina, incluso si hubo error.

Observers y outputs

15



Observers en acción:

Son **ansiosos**: ejecutan apenas pueden.

Son **olvidadizos**: no recuerdan lo que hicieron antes.

Actúan por sus **efectos secundarios**: su resultado no se almacena ni se propaga. Solo **ejecutan acciones**.

Si usan valores reactivos, **los activan inmediatamente**.

Observers y outputs

```
500 - ````{r}
501 x <- reactiveVal(1)
502 y <- observe({
503 x()
504 observe(print(x())))
505 })
506 ````

[1] 1

507
508 - ````{r}
509 x(2)
510 ````

[1] 2
[1] 2

511
512 - ````{r}
513 x(3)
514 ````

[1] 3
[1] 3
[1] 3
```

Cada vez que x() cambia:

1. El primer observe() se activa, porque detecta que x() cambió.
2. Dentro de ese bloque, se ejecuta otro observe(print(x())) → o sea, se crea un nuevo observer.

Entonces, cada vez que cambia x(), se agrega un nuevo observer más que imprime el valor

¿Y por qué eso pasa?

Porque el primer observe() no recuerda que ya creó otro observer antes – es **olvidadizo**. Y al volver a ejecutarse, repite la creación.

¿Por qué esto no se recomienda?

Porque crear observadores dentro de observadores es una receta para:

- ❌ Comportamiento inesperado
- ⚡ Repeticiones que escalan
- 🐛 Bugs difíciles de rastrear



Observers y outputs



observe() y observeEvent()

- Crean observadores que se activan cuando cambian los valores reactivos que usan.
- Se ejecutan por sus efectos secundarios (no devuelven valores que se usen en otro lado).
- Pueden activar lógica, guardar datos, mostrar modales, escribir archivos... sin producir outputs visibles.

¿En qué se diferencian?

observe()

- Es más **flexible pero más crudo**.
- Descubre **automáticamente** sus dependencias al correr.
- Ideal si se quiere controlar manualmente qué se observa y cómo se reacciona.

observeEvent(evento, handler)

- Es más **elegante y legible**.
- Divide la lógica en dos partes:
- Qué **evento** escuchar (input\$botón, por ejemplo).
- Qué **acción** realizar (guardar, mostrar, etc).
- Internamente usa isolate() para que lo observado no invalide lo que no debe.

15

Códigos de aislamiento

isolate(): silencio selectivo

- Evita que se cree una dependencia.
- El valor se usa sin activar el grafo.
- Ideal para leer algo sin que invalide (ej: contador de cambios sin bucle infinito).

```
r<- reactiveValues(count = 0, x = 1)
class(r) #muestra que r es un objeto reactivo
observe({
  r$x
  r$count <- isolate(r$count) + 1
})
```

observeEvent(): escucha activa con acción puntual

- Escucha un evento específico.
- Ejecuta código cuando ese evento se activa.
- Usa isolate() internamente para proteger el handler.

```
observeEvent(r$x, {
  r$count <- r$count + 1
})
```

eventReactive(): escucha activa con valor devuelto

- Escucha un evento y **devuelve un valor reactivo**.
- Útil para cálculos que dependen de eventos puntuales.
- También usa isolate() internamente

Códigos de aislamiento



observeEvent(x, y) ≈ observe({ x; isolate(y) })
eventReactive(x, y) ≈ reactive({ x; isolate(y) })

Estas funciones ya incorporan el aislamiento como parte de su diseño, entonces no es necesario repetir isolate() dentro del handler. Eso hace que el código sea:

- Más claro
- Más conciso
- Menos propenso a errores de dependencia no deseada

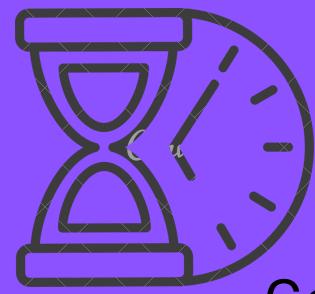
ARGUMENTOS EXTRA

ARGUMENTO	¿QUÉ HACE?
ignoreNULL= FALSE	Maneja eventos con NULL (útil para inputs que comienzan vacíos)
ignoreInit= TRUE	Evita que el handler se dispare al iniciar
once = TRUE	Solo para observeEvent: ejecuta el handler una sola vez

Son parámetros que no siempre se necesitan, pero saber que existen da opciones para cuando las condiciones del input o el tiempo de activación importan.

invalidateLater() y el tiempo reactivo

15



USOS FRECUENTES DE invalidateLater()

- Generar actualizaciones automáticas, incluso sin interacción.
- Animaciones
- Relojes
- Simulaciones
- Relectura periódica de datos externos

FUNCIÓN	¿QUÉ HACE?
isolate()	Evita que algo se vuelva reactivo. Silencia conexiones
invalidateLater(ms)	Fuerza una invalidación futura, sin cambios visibles.

```
x <- reactive({  
  invalidateLater(500)  
  rnorm(10) #se actualiza cada 500 ms  
})
```

invalidateLater() y el tiempo reactivo

15

Polling: leer datos que cambian

Problema con **invalidateLater()**: invalidar todo incluso si los datos no cambiaron.

Solución: **reactivePoll()**

- Usa un chequeo barato (file.mtime(), por ejemplo).
- Solo recalcula si hubo cambios.

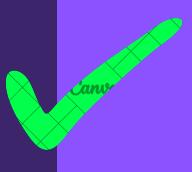
Extra tip: **reactiveFileReader()** es una versión más simple para archivos CSV.



Cálculos largos y precisión

Si el cálculo tarda más que el tiempo de espera... ¡ciclo infinito!

Solución: **on.exit(invalidateLater(...), add = TRUE)** que invalida después de terminar.



invalidateLater() es una sugerencia, no una exigencia.

Para precisión real, usar **proc.time()** y ajustar manualmente

```
server <- function(input, output, session) {  
  data <- reactivePoll(1000, session,  
    function() file.mtime("data.csv"), #chequea  
    cada 1000 ms si hubo cambios  
    function() read.csv("data.csv") #actualiza si  
    hubo cambios  
  )  
}  
  
#reactivePoll combina las dos funciones y  
#conecta con el grafo
```

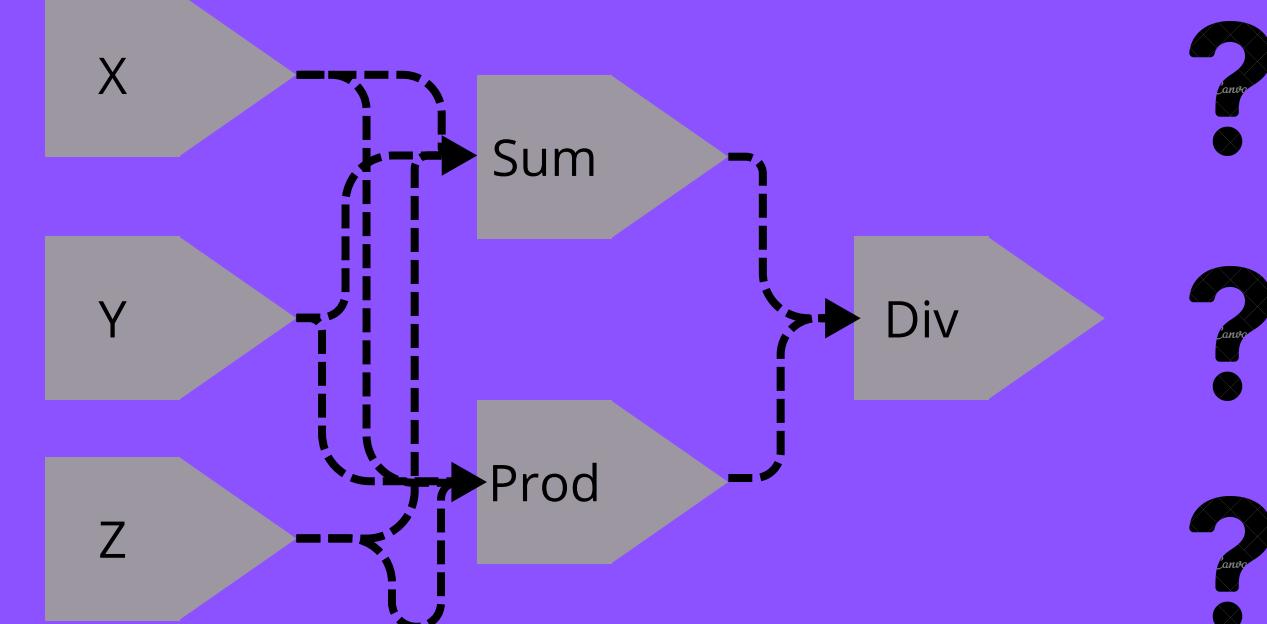
```
velocity <- 3  
r <- reactiveValues(distance = 1)  
last <- proc.time()[[3]] #guarda el momento actual (como si fuera una foto del reloj)  
observe({  
  cur <- proc.time()[[3]] # se saca una nueva "foto del reloj"  
  time <- last - cur # se calcula cuánto tiempo pasó desde la última vez  
  last <-> cur # se actualiza la variable global last  
  r$distance <- isolate(r$distance) + velocity * time #cálculo de distancia  
  invalidateLater(100)  
})
```



ALGUNOS EJERCICIOS

14.4 # 1. Dibuje el gráfico reactivo para la siguiente función del servidor y luego explique por qué no se ejecutan los reactivos.

```
server <- function(input, output, session) {  
  sum <- reactive(input$x + input$y + input$z)  
  prod <- reactive(input$x * input$y * input$z)  
  division <- reactive(prod() / sum())  
}  
  
## No hay salidas que shiny pueda activar,  
por eso no se ejecutan los reactivos
```



ALGUNOS EJERCICIOS

14.4 #2. El siguiente gráfico reactivo simula un cálculo de larga duración utilizando Sys.sleep().

#¿Cuánto tiempo tardará el gráfico en recalcularse si x1 cambia? ¿Qué pasa con x2 o x3?

```
x1 <- reactiveVal(1)
x2 <- reactiveVal(2)
x3 <- reactiveVal(3)
y1 <- reactive({
  Sys.sleep(1)
  x1()
})
#y1 depende solo de x1
y2 <- reactive({
  Sys.sleep(1)
  x2()
})
#y2 depende solo de x2
y3 <- reactive({
  Sys.sleep(1)
  x2() + x3() + y2() + y2()
})
#y3 depende de x2, x3 y de y2
observe({
  print(y1())
  print(y2())
  print(y3())
})
#Si x1 cambia, entonces y1 tarda 1 seg
#Si x2 cambia, entonces y2 e y3 se invalidan y tienen que recalcularse.
Tarda 2 seg
#Si x3 cambia, entonces y3 tarda 1 seg
```

ALGUNOS EJERCICIOS

15.1 #2. Diseñe y realice un pequeño experimento para verificar que reactiveVal() también tiene semántica de referencia.

```
a <- reactiveVal(10)
b <- a  #Si cambio b, debería cambiar a
b(99)
a()  #a debería ser 99
```



15.2 #2. Modifique la aplicación anterior para usar req() en lugar de stop(). Verifique que los eventos se sigan propagando de la misma manera. ¿Qué sucede cuando se usa el argumento cancelOutput?

```
reactlog::reactlog_enable()
```

```
ui <- fluidPage(
checkboxInput("error", "error?"),
textOutput("result")
)
server <- function(input, output, session) {
a <- reactive({
  req(!input$error, cancelOutput = TRUE)
  1
})
b <- reactive(a() + 1)
c <- reactive(b() + 1)
output$result <- renderText(c())
}

shinyApp(ui, server)
```

```
shiny::reactlogShow()
```

#req() también genera una especie de error, pero no invalida la sesión
#req(..., cancelOutput = TRUE), el output conserva su última visualización anterior
#el grafo se comporta igual que en el primer caso: la interrupción se propaga, pero el resultado es distinto en pantalla.

ALGUNOS EJERCICIOS

154 #1. Complete la aplicación a continuación con una función de server que actualice el out con el valor de x sólo cuando se presiona el botón.

```
ui <- fluidPage(  
  numericInput("x", "x", value = 50, min = 0, max = 100),  
  actionButton("capture", "capture"),  
  textOutput("out")  
)  
server <- function(input, output, session) {  
  valor <- eventReactive(input$capture, {  
    input$x  
  })  
  
  output$out <- renderText(valor())  
}  
  
shinyApp(ui, server)  
  
#eventReactive() ya incorpora isolate() internamente lo que evita reactividad no deseada.
```

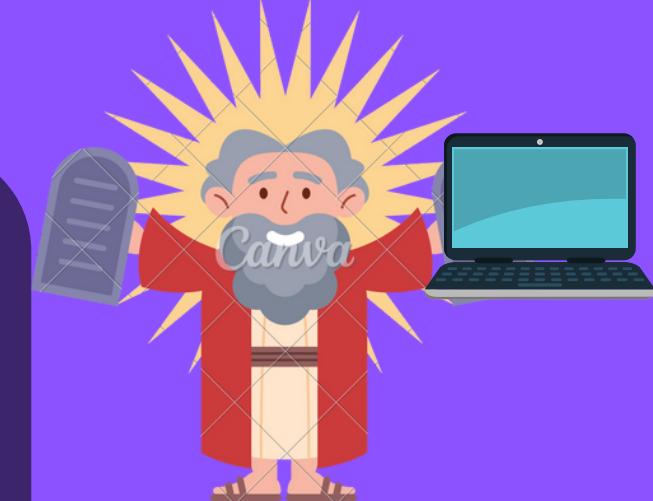


Mandamientos para domar el flujo reactivo



Sobre observe(), observeEvent(), y eventReactive()

- No observarás todo como quien espía por la cerradura. Mejor usa observeEvent() y escucha lo justo.
- Crearás tus observers en la cima del servidor, nunca en las cavernas profundas de un output.
- No multiplicarás tus observers como Gremlins reactivos. El grafo perderá la cordura y tú perderás el control.
- Aislarás tus inputs cuando observes eventos – para que el botón tenga el control, y no el input rebelde.
- Silenciarás con isolate(), escucharás con observeEvent(), y responderás con eventReactive(). Cada cual tiene su momento.
- No leerás lo que escribís sin isolate, o el grafo caerá en el bucle eterno.
- No modificarás reactivos sin isolate, o verás tu app atrapada en el bucle eterno.
- Si el grafo se agita sin razón, rastrearás la turbulencia con reactlog().



Mandamientos para domar el flujo reactivo

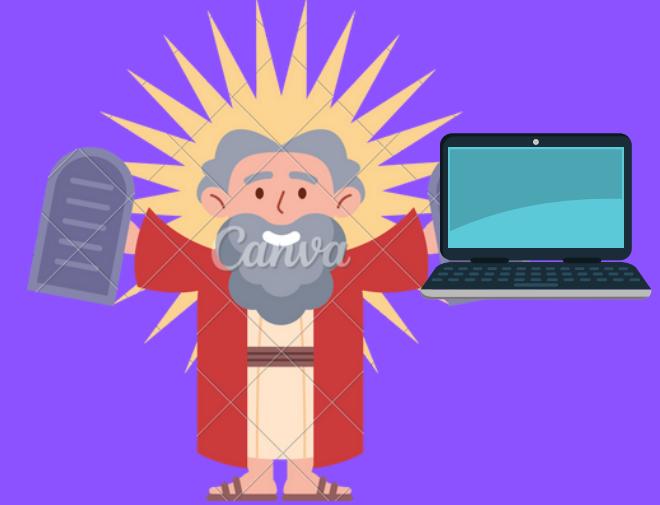


Sobre tiempo y control



- No asumirás que tu objeto es sólo una lista: verificarás con `class()` y entenderás su esencia reactiva.
- Programarás tus actualizaciones con `invalidateLater()`, incluso si nadie las pidió.
- Consultarás el mundo exterior con sabiduría: primero chequeo, después lectura – pues no todo cambio merece reactividad.
- Guardarás el momento con `proc.time()`, pues el tiempo no se mide en deseos sino en relojes.
- Actualizarás tus cálculos según el tiempo real transcurrido, no según el tiempo que esperabas.
- En Shiny, la distancia no siempre se mide en píxeles ni en inputs... también puede crecer al ritmo del reloj.

Mandamientos para domar el flujo reactivo



Sobre activación perezosa

- No dejarás tus reactivos solitos y sin consumidores, pues Shiny no activa lo que nadie invoca.

Con estos mandamientos, protegerás tu app del caos reactivo, iluminarás tu `server()` con sabiduría... y tal vez hasta entiendas `reactlog` sin miedo.

MUCHAS GRACIAS!