Package 'irid'

August 2, 2026

Title Component-Based 'Shiny' UI with Fine-Grained Reactivity

Version 0.3.0

Description A component-based reactive UI framework for 'Shiny' that binds reactive values directly to DOM attributes, enabling fine-grained updates without re-rendering or 'update*Input' callbacks.

License MIT + file LICENSE

URL https://irid.kylehusmann.com, https://github.com/khusmann/irid

BugReports https://github.com/khusmann/irid/issues

Depends R (>= 4.1.0)

Imports cli, htmltools, rlang (>= 1.2.0), shiny

Suggests bslib, callr, chromote, DT, ggplot2, httpuv, jsonlite, knitr, pkgload, plotly, purrr, rmarkdown, testthat (>= 3.0.0), tibble, vctrs, withr

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.3

Config/testthat/edition 3

NeedsCompilation no

Author Kyle Husmann [aut, cre]

Maintainer Kyle Husmann <irid@kylehusmann.com>

Repository CRAN

Date/Publication 2026-08-02 16:10:02 UTC

Contents

`Case` *Define a case for* `Match()`

Description

Define a case for `Match()`

Usage

```
Case(predicate, body)
```

Arguments

`predicate`	One of: a function `\(v) cond` of the bound value, a function `\() cond` ignoring the bound value (cross-cutting), or a literal value (matched against the bound value via `identical()`).
`body`	A function `\(v) tag_tree` or `\() tag_tree` that returns the tag tree to render when this case is active. The function is called fresh on each case activation — the active case's reactives and DOM are torn down on transition, so a tag tree captured outside the function would reference dead closures.

Value

A case definition (a list).

Default — *Define a default (fallback) case for* `Match()`

Description

Sugar for `Case(\() TRUE, body)` — matches when no earlier `Case` does. Place this as the last argument to `Match`.

Usage

```
Default(body)
```

Arguments

`body`	A function returning the tag tree to render when no other case matches. Same arity rules as `Case()` body.

Value

A case definition (a list).

DTOutput — *Embed a DT DataTable output in a irid tag tree*

Description

Shorthand for `Output(DT::renderDT, DT::DTOutput, ...)`. Requires the **DT** package.

Usage

```
DTOutput(fn, ...)
```

Arguments

`fn`	A function that produces a DataTable.
`...`	Additional arguments passed to `DT::DTOutput()`.

Value

A irid output node.

`Each` *Render a reactive list*

Description

Iterates over a reactive list and calls `fn` for each item. The callback receives a per-item callable (a mini-store for record items, a scalar accessor for atomic items) and an optional position accessor. The reconciliation strategy is selected by `by`:

Usage

```
Each(items, fn, by = NULL)
```

Arguments

`items`	A reactive expression that returns a list.
`fn`	A function of `(item)`, `(item, pos)`, or `()`. `item` is the per-item callable (mini-store or scalar accessor); `pos` is a 0-arg reactive accessor returning the item's current 1-indexed slot. `pos` is constant under `by = NULL` (positional identity) and live under `by = fn` (fires on reorder).
`by`	`NULL` for positional reconciliation, or a function that extracts a unique comparable key from each item for keyed reconciliation.

Details

- **Positional** (`by = NULL`, the default) — slot *i* is slot *i*. The list can grow or shrink at the end; in-place value changes update each slot's accessor without DOM recreation.
- **Keyed** (`by = \(x) x$id`) — items are tracked across reorders, adds, and removes by their key. Kept items propagate new values through their mini-store (only changed leaves fire); reordered items have their DOM nodes moved (no recreation).

Records are projected as a per-item mini-store: `item()` reads the whole record, `item(record)` writes it back, `item$field()` reads a leaf, and `item$field(v)` is a synthetic setter that writes through the parent. Scalars are passed as a per-item callable: `item()` reads, `item(value)` writes back to the parent's slot.

Records may be heterogeneous in shape — different leaf trees per entry. Each slot's mini-store is sized to its own item, and a `Match()` inside the body dispatches on the discriminator:

```
Each(state$blocks, by = \(b) b$id, \(block) {
  Match(block,
    Case(\(b) b$type == "heading",   \(b) Heading(b)),
    Case(\(b) b$type == "paragraph", \(b) Paragraph(b)),
    Case(\(b) b$type == "todo",      \(b) Todo(b))
  )
})
```

When a record's shape changes (different key set, or a sub-record's shape shifts), that one entry is torn down and rebuilt with the new mini-store. Shape-stable updates use the fine-grained in-place path.

Mixing records and scalars in the same list is rejected at flush time as a likely data-modeling slip — wrap scalars in `list(value = ...)` to mix them.

Value

A irid control-flow node.

`iridApp` *Create a irid application*

Description

Builds a Shiny app from a function that returns a irid tag tree. The function is called once per session so that each client gets its own reactive state.

Usage

```
iridApp(fn, ...)
```

Arguments

`fn`	A zero-argument function that returns a irid tag tree (e.g. a `page_sidebar()` call containing reactive attributes and event handlers).
`...`	Additional arguments passed to `shiny::shinyApp()`.

Value

A Shiny app object.

`iridExample` *Run a irid example application*

Description

Launches one of the example apps shipped with the package, mirroring `shiny::runExample()`. The examples are the same apps published as editable editors on the package website.

Usage

```
iridExample(
  example = NA,
  port = getOption("shiny.port"),
  launch.browser = getOption("shiny.launch.browser", interactive()),
  host = getOption("shiny.host", "127.0.0.1"),
  display.mode = c("showcase", "normal", "auto")
)
```

Arguments

`example`	Name of the example to run (e.g. `"todo"`, `"old-faithful"`) — the same name used in its website URL. If omitted, the available examples are listed.
`port`	The TCP port the application should listen on. Defaults to `getOption("shiny.port")`, or a random port if unset.
`launch.browser`	If `TRUE`, the system's default web browser is launched automatically. Defaults to `TRUE` in interactive sessions.
`host`	The IPv4 address the application should listen on. Defaults to `getOption("shiny.host", "127.0.0.1")`.
`display.mode`	The display mode. Defaults to `"showcase"` — the annotated source shown alongside the running app; use `"normal"` to run the app by itself, or `"auto"` to defer to the app's own setting.

Details

Called with no argument (or an unrecognised name), it lists the available examples instead of launching one.

Some examples depend on packages beyond irid's hard dependencies (e.g. bslib, plotly). The runner checks for these up front and, if any are missing, stops with a message naming what to install. A couple of examples (`codemirror`, `plotly`) also load JavaScript from a CDN at runtime, so they need an internet connection even when run locally.

By default the app runs in Shiny's *showcase* display mode, so the annotated source appears beside the running app (as with `shiny::runExample()`). Pass `display.mode = "normal"` to run the app on its own.

The remaining arguments mirror `shiny::runExample()`.

Value

If `example` is supplied, launches the app and blocks until it is stopped. Otherwise returns the available example names invisibly as a character vector (after listing them).

Examples

```
# List the available examples
iridExample()

if (interactive()) {
```

```
  iridExample("todo")
}
```

`iridOutput` *Create a irid UI output placeholder*

Description

Creates a `shiny::uiOutput()` with the irid JavaScript dependency attached. Use this in a standard Shiny UI to mark where `renderIrid()` should inject its content.

Usage

```
iridOutput(id)
```

Arguments

`id`	The output ID, matching the corresponding `renderIrid` call.

Value

An HTML tag with the irid dependency.

`IridWidget` *Construct a widget — a wrapper for an arbitrary JavaScript library*

Description

`IridWidget()` is the third irid process-tags citizen (alongside control-flow nodes and `Output`). It emits a container element plus an init record that mount turns into a `widget-init` op. The client's `irid.defineWidget("<name>", factory)` registration is looked up by `name` and called once per mount. The factory may return its `{update, destroy}` handle directly or a Promise of it — make it `async` and `await` whatever its construction needs first (a script-tag library global, an ESM `import`, a WASM init). irid buffers updates during the wait and disposes cleanly on a teardown mid-construction. See the JS-side API in `ARCHITECTURE.md`.

Usage

```
IridWidget(
  name,
  props = list(),
  events = list(),
  deps = NULL,
  container = NULL
)
```

Arguments

name	Widget registry name, matching a JS-side `irid.defineWidget("<name>", ...)` call. Required, non-empty character scalar.
props	Named list of props. Callable values are two-way-capable bindings; non-callable values are init-only constants. `NULL` entries are forwarded to JS as `null` (not dropped). Wrap a value in `wire()` to tune its write-back timing.
events	Named list of notifications (client → server), keyed by wire event name. Each value is a handler, a `wire()`, or `NULL` (dropped). `dom_opts` is not allowed.
deps	Optional `html_dependency` or list of them. Required for any widget whose JS isn't already loaded by some other means.
container	Optional `shiny.tag` for the wrapper element. Defaults to `tags$div()`. irid sets `id` and `data-irid-widget` automatically. Configure any DOM events on the container by wrapping their handlers in `wire()` on the container directly.

Details

Props are two-way-capable by default, exactly like DOM `value` / `checked`. A callable prop (`reactiveVal`, store leaf, `reactiveProxy`, ...) reads inbound to the widget (server → client, routed to the factory's `update(key, value)` hook) *and* accepts write-back: when the widget JS calls `setProp(key, value)`, irid writes the value through the bound reactive (gated by writability — a read-only reactive's write is dropped and the canonical value is snapped back). Whether a prop is *actually* two-way depends on whether the widget JS pushes through `setProp`; the snap-back machinery is latent until it does. A non-callable prop rides in the init message as a constant and is never re-sent.

Wrap a prop in `wire()` only to **tune** its round-trip timing (`content = wire(content, wire_debounce(200))`) — never to enable or disable two-way. To react to a prop's change, observe the bound reactive or pass a `reactiveProxy`; a bound prop is not *also* handled.

`events` carries genuine notifications the widget emits that correspond to no prop (e.g. `cursor-changed`). Keys are the wire event names the widget JS passes to `sendEvent()` (lowercase kebab-case by web `CustomEvent` convention). Each value is a handler or a `wire()` (to tune timing); `NULL` entries are dropped so optional handlers forward declaratively. `dom_opts` is illegal on a widget event.

Value

A irid widget construct with class `irid_widget`.

Match *Render the first matching case*

Description

Evaluates cases in order against a bound value and renders the body of the first case whose predicate is `TRUE`. Records are projected as a mini-store for the active case's body; scalars are passed as the bare callable. On active-case change, the previous case is torn down (its reactives, DOM, and mini-store) and the new case is mounted fresh.

Usage

```
Match(callable, ...)
```

Arguments

`callable`	The bound value — any 0-arg callable. Records (named lists) are projected as a mini-store; scalars are passed as the bare callable.
`...`	One or more `Case()` / `Default()` values.

Details

Use a choice function as the leading callable to fold unrelated reactive state into a tagged variant on the fly:

```
Match(\() if (loading()) list(tag = "loading") else list(tag = "data", x = data()),
  Case(\(r) r$tag == "loading", \() Spinner()),
  Case(\(r) r$tag == "data",    \(r) Items(r$x))
)
```

Conceptually, `When()` is a fixed-shape binary specialization of `Match`.

Value

A irid control-flow node.

`Output` *Embed a Shiny render/output pair in a irid tag tree*

Description

A generic wrapper that pairs a Shiny render function with its corresponding output function. For common cases, use the convenience wrappers `PlotOutput()`, `TableOutput()`, or `DTOutput()`.

Usage

```
Output(render_fn, output_fn, fn, ...)
```

Arguments

`render_fn`	A Shiny render function (e.g. `renderPlot`).
`output_fn`	A Shiny output function (e.g. `plotOutput`).
`fn`	A function passed to `render_fn`.
`...`	Additional arguments passed to `output_fn`.

Value

A irid output node.

PlotlyOutput *Reactive plotly output*

Description

A first-class output primitive for interactive plotly charts, on par with PlotOutput and TableOutput. Unlike the {plotly} htmlwidget — which destroys and recreates the chart on every reactive update — PlotlyOutput uses Plotly.react() for incremental updates, so a data change preserves the user's zoom, pan, and selection (via plotly's uirevision).

Usage

```
PlotlyOutput(
  spec,
  ...,
  onClick = NULL,
  onHover = NULL,
  onUnhover = NULL,
  onDoubleclick = NULL,
  onDeselect = NULL,
  onSelecting = NULL,
  onBrushing = NULL,
  onLegendClick = NULL,
  onLegendDoubleclick = NULL,
  onClickAnnotation = NULL,
  onSunburstClick = NULL,
  onRelayout = NULL,
  container = NULL
)
```

Arguments

spec	A zero-argument function returning a plotly object (from plotly::plot_ly() or plotly::ggplotly()). Re-evaluated reactively; Plotly.react() diffs the result client-side.
...	Named state arguments, each a callable (reactiveVal, store leaf, reactiveProxy) or constant. Recognized: xaxis_range, yaxis_range, xaxis<n>_range, yaxis<n>_range, dragmode, hovermode, selected_ids, trace_visibility. NULL means "don't override the spec". Unknown names error — use onRelayout for fields outside the table. A *_range value is a length-2 vector of the axis's own units: numeric for a continuous, log, or category axis; ISO date-time **strings** (c("2020-04-01", "2020-06-30")) for a date axis, matching how plotly reports and accepts them. selected_ids is the box/lasso selection, keyed on plotly's per-point ids (a character vector of id values). Binding it requires the plot to supply ids (plot_ly(ids = ~key) / aes(ids = key)); because the value is keyed on identity rather than

position, the selection survives data changes that renumber points (filtering, sorting, trace recomposition).

`trace_visibility` is the per-trace on/off state, a sparse named character vector keyed by trace `name` (`c("8" = "legendonly")`), values in `"true"` / `"false"` / `"legendonly"`. Like `selected_ids` it is identity-keyed, so it stays correct across trace recomposition; binding it requires the traces to be named.

`onClick`, `onHover`, `onUnhover`, `onDoubleclick`
: Discrete pointer callbacks. Each receives the (slimmed) plotly event payload.

`onDeselect`, `onSelecting`, `onBrushing`
: Selection-lifecycle notifications. `onDeselect` is a side-effect notification only — when `selected_ids` is bound, the clear already flows through its prop channel.

`onLegendClick`, `onLegendDoubleclick`, `onClickAnnotation`, `onSunburstClick`
: Discrete interaction callbacks.

`onRelayout`
: Escape hatch — receives the raw `plotly_relayout` payload (flat dot-notation keys) for fields outside the translation table.

`container`
: Optional `shiny.tag` for the wrapper element.

Details

`PlotlyOutput` is a thin wrapper over IridWidget. User-controllable UI state (axis ranges, drag mode, selection, trace visibility) is exposed as **named reactive arguments**, each a two-way prop: bind a `reactiveVal`, store leaf, or reactiveProxy and the user's interaction writes back to it. A `reactiveProxy` that rejects a write snaps the plot back. Discrete events (clicks, hovers, legend interactions) are plain `on*` callbacks.

Value

An `irid_widget` construct.

PlotOutput *Embed a plot output in a irid tag tree*

Description

Shorthand for `Output(renderPlot, plotOutput, ...)`.

Usage

```
PlotOutput(fn, ...)
```

Arguments

`fn`
: A function that produces a plot.

`...`
: Additional arguments passed to `shiny::plotOutput()`.

Value

A irid output node.

`reactiveProxy` *Wrap a reader and optional writer as a callable*

Description

Builds a callable proxy from a 0-arg `get` reader and an optional 1-arg `set` writer. The proxy is itself a callable: `proxy()` invokes `get()`, `proxy(value)` invokes `set(value)`. Auto-bind treats the proxy like any other callable, so it composes with `value` and `checked` props without any special handling.

Usage

```
reactiveProxy(get, set = NULL)
```

Arguments

`get`	A 0-arg callable returning the read value. Typically a `reactiveVal`, a `reactiveStore` leaf, another `reactiveProxy`, or a closure like `\() transform(rv())`.
`set`	A 1-arg function called with the incoming value on write, or `NULL` for a read-only proxy. Defaults to `NULL`.

Details

`set` is a side-effectful handler, not a pure transform. It receives the incoming value and decides what to do — write to a target, write a transformed value, set an error flag, trigger a side effect, or drop the write entirely. Because `set` is a closure, it can read sibling state for cross-field validation.

Pass `set = NULL` (or omit it) to make the proxy read-only — writes are silently dropped. With auto-bind, this lets the input snap back to the current value via the optimistic-update protocol.

Proxies compose: a proxy is itself a callable (0-arg → `get`, 1-arg → `set`), so another `reactiveProxy` can use it as either its `get` or its `set`.

Value

A callable with class `c("reactiveProxy", "reactive", "function")`.

`reactiveStore` *Hierarchical reactive state container*

Description

Builds a callable hierarchical state tree from a nested list. The shape rule is simple: a bare named list (length > 0) becomes a navigable `reactiveStore` node (every sub-tree is itself a `reactiveStore`); everything else becomes a plain `shiny::reactiveVal()` leaf that accepts any value on write. To force a bare named list to be treated as a leaf instead of a store, wrap it in `base::I()`.

Usage

```
reactiveStore(initial)
```

Arguments

`initial`	A bare named list describing the initial shape. Sub-positions classify as follows: bare named lists (length > 0) become store sub-nodes; anything else (scalars, vectors, `NULL`, unnamed/empty bare lists, classed lists, or any value wrapped in `I()`) becomes a leaf. Partially-named bare lists are an error.

Details

Every node is callable: `node()` reads, `node(value)` writes. Branch writes replace: the input must list every locked key for that branch. Both unknown keys and missing keys are an error. Types are not enforced. Per-field writes (`node$key(value)`) remain the dedicated single-slot path — use them when you want to update one field without naming the rest.

`length()` on a leaf returns 1 (the underlying `reactiveVal` is a callable, and neither R's closure default nor shiny override `length`). htmltools' `dropNullsOrEmpty` calls `length()` on every attribute value, so this is what lets `value = state$leaf` survive tag construction. Use `length(leaf())` for the underlying length.

Value

A callable `reactiveStore` node with class `c("reactiveStore", "reactive", "function")`. Sub-trees share the same class. Leaf positions are plain `shiny::reactiveVal()`s with class `c("reactiveVal", "reactive", "function")` — the shared `"reactive"` class is what auto-bind dispatches on.

`reactiveVal` *Shiny reactive primitives*

Description

These functions are re-exported from **shiny** for convenience, so you can use them in irid apps without loading shiny explicitly.

Details

- `reactiveVal()`: Create a reactive value
- `reactive()`: Create a reactive expression
- `observe()`: Create an observer
- `observeEvent()`: Create an event-driven observer
- `isolate()`: Run an expression without reactive dependencies
- tags: HTML tag builder
- `tagList()`: Combine tags into a list

See the **shiny** documentation for full details.

Value

The return values are those of the underlying **shiny** functions:

- `reactiveVal()` returns a function that reads the current value when called with no argument and sets it when called with one value.
- `reactive()` returns a reactive expression (a function of class `reactiveExpr/reactive`) that yields the cached expression value when called.
- `observe()` and `observeEvent()` return an observer reference object (class `Observer`) invisibly; they are called for their side effects.
- `isolate()` returns the value of `expr`, evaluated without taking a reactive dependency.
- `tags` is a named list of functions that construct HTML tag objects, and `tagList()` returns a list of tags (class `shiny.tag.list`).

renderIrid *Render irid content inside a Shiny app*

Description

A render function for use with `iridOutput()`. Evaluates `expr` to produce a irid tag tree, processes it, and mounts reactive bindings and event handlers after the UI is flushed.

Usage

```
renderIrid(expr, env = parent.frame(), quoted = FALSE)
```

Arguments

`expr`	An expression that returns a irid tag tree.
`env`	The environment in which to evaluate `expr`.
`quoted`	If `TRUE`, `expr` is already a quoted expression.

Value

A `shiny::renderUI()` result.

TableOutput *Embed a table output in a irid tag tree*

Description

Shorthand for `Output(renderTable, tableOutput, ...)`.

Usage

```
TableOutput(fn, ...)
```

Arguments

`fn`	A function that produces a table.
`...`	Additional arguments passed to `shiny::tableOutput()`.

Value

A irid output node.

When *Conditionally render content*

Description

Renders the `yes` branch when `condition` is `TRUE`, and `otherwise` (if provided) when it is `FALSE`. The active branch is fully mounted and the inactive branch is destroyed.

Usage

```
When(condition, yes, otherwise = NULL)
```

Arguments

`condition`	A reactive expression that returns a logical value.
`yes`	A 0-arg function returning the tag tree to render when the condition is `TRUE`.
`otherwise`	An optional 0-arg function returning the tag tree to render when the condition is `FALSE`. With `NULL`, nothing mounts on the false branch.

Details

Conceptually a fixed-shape binary specialization of `Match()`: `When(\() cond, \() yes, \() no)` is equivalent to `Match(\() cond, Case(TRUE, \() yes), Case(FALSE, \() no))`.

Bodies are 0-arg functions that return tag trees — not tag trees directly. `When` mounts and unmounts the active branch on transition, so each activation must construct a fresh tag tree (the previous branch's closures were torn down with its reactives). Reach for `Match()` when the branch needs to consume the dispatching value.

Value

A irid control-flow node.

wire *Event & binding dispatch config*

Description

Configure how an event handler or value binding is dispatched between the browser and the server. A single per-slot carrier, `wire()`, rides the slot it configures — an `on*` handler slot or a `value`/`checked` binding slot — so an event's timing, backpressure, and DOM-listener options live next to the handler/reactive they govern rather than in separate element-level lists.

Usage

```
wire_immediate()

wire_throttle(ms, leading = TRUE)

wire_debounce(ms)

wire_dom_opts(
  prevent_default = FALSE,
  stop_propagation = FALSE,
  capture = FALSE,
  passive = FALSE,
  filter = NULL
)

wire(subject = NULL, timing = NULL, coalesce = NULL, dom_opts = NULL)
```

Arguments

ms	Minimum interval (throttle) or quiet period (debounce) in milliseconds.
leading	If TRUE (default), fire immediately on the first event. If FALSE, wait for the timer before firing.
prevent_default	Call event.preventDefault() before dispatch.
stop_propagation	Call event.stopPropagation() before dispatch.
capture	Register the listener in the capture phase.
passive	Register the listener as passive.
filter	A JavaScript expression string, evaluated client-side with the event object bound to e; the event is dropped when it is falsy. NULL (the default) applies no filter.
subject	The handler or reactive the wire configures. The slot decides which: a bare callable means "bind" in value = / checked = and "handle" in an on* slot. NULL (the default) carries config only — used by widget wrappers that supply a default shape for the caller to override via merge().
timing	An irid_wire_timing shape (wire_immediate(), wire_throttle(), wire_debounce()), or NULL for the per-event default.
coalesce	Logical scalar, or NULL to derive from the timing mode.
dom_opts	A wire_dom_opts() record, or NULL.

Value

wire() returns an irid_wire; the timing constructors return an irid_wire_timing; wire_dom_opts() returns an irid_dom_opts.

Timing shapes

The timing constructors are pure shapes — they describe *when* an event fires and carry no other config:

- `wire_immediate()`: Fires on every event with no rate limiting.
- `wire_throttle(ms, leading)`: Fires at most every `ms` milliseconds while the event is active.
- `wire_debounce(ms)`: Waits until the user pauses for `ms` milliseconds before firing.

When a wire carries no `timing`, irid applies a per-event default keyed on the DOM event name: `input` → `wire_debounce(200)` (typing produces a flood of intermediate values); the high-frequency continuous streams (`mousemove`, `pointermove`, `touchmove`, `drag`, `dragover`, `scroll`, `wheel`, `resize`) → `wire_throttle(100)` (paced "latest position" stream, with the derived `coalesce = TRUE` keeping it from outrunning the server); every other event → `wire_immediate()`. Explicit `timing` always wins over the default.

Backpressure (`coalesce`)

`coalesce` is universal across timing modes, so it lives on the carrier, not in the shapes. When `TRUE`, dispatch gates on server-idle state so events never queue faster than the server can process them. When `NULL` (the default), it derives from the timing mode: `FALSE` for `wire_immediate()`, `TRUE` for `wire_throttle()` / `wire_debounce()`.

DOM listener options

`wire_dom_opts()` bundles the DOM-only listener flags. It is legal only where the event is backed by a real DOM listener (a plain tag, a custom element emitting cancelable events); placing it on a widget-emitted event errors. Whether `prevent_default` has any effect further depends on the event being cancelable — a runtime fact.

`filter` is a JavaScript expression string evaluated client-side with the DOM event object bound to `e`. When it evaluates falsy the event is dropped entirely — no `prevent_default`/`stop_propagation`, no server round-trip — so a handler that only cares about some events never floods the server with the rest. For example, an `onKeyDown` that acts only on Enter takes `filter = "e.key === 'Enter'"`.

The event object

The `event` argument passed to handlers is a list containing all primitive-valued properties (string, numeric, logical) from the browser event object, plus these element properties:

`value` The element's current value (character).

`valueAsNumber` Numeric value of the element, or `NA` if the input is empty or non-numeric (e.g. a blank text box). Useful for range and number inputs.

`checked` Logical, for checkbox and radio inputs.

Keyboard events additionally include `key`, `code`, `ctrlKey`, `shiftKey`, `altKey`, and `metaKey`.

Index