Package 'shinyreact'

September 23, 2026

Title Client-Side 'React' Interface for 'Shiny'

Version 0.1.0

Description Server-side plumbing for the 'ui.tsx' pattern in 'Shiny': the user interface is defined in a client 'React' (<https://react.dev/>) bundle, and the 'Shiny' server contains only reactive computation. Provides page builders that discover and serve the client bundle, a render function that publishes any JSON-serializable value to the client, and custom messages to 'React' components. Ships no user interface components, so the app author owns the whole front end. The 'React' runtime and the client hooks are bundled, so no JavaScript build step is required to get started.

License MIT + file LICENSE

URL https://posit-dev.github.io/shinyreact/r/, https://github.com/posit-dev/shinyreact

BugReports https://github.com/posit-dev/shinyreact/issues

Imports brio, cli, htmltools, jsonlite, rlang, shiny (>= 1.13.0), utils

Suggests knitr, later, rmarkdown, shinytest2, spelling, testthat (>= 3.0.0), withr

VignetteBuilder knitr, rmarkdown

Config/needs/check spelling

Config/Needs/website pkgdown, tidyverse/tidytemplate

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

Encoding UTF-8

Language en-US

NeedsCompilation no

Author Barret Schloerke [cre, aut] (ORCID: <https://orcid.org/0000-0001-9986-114X>), Winston Chang [ctb] (ORCID: <https://orcid.org/0000-0002-1576-2126>),

Garrick Aden-Buie [ctb] (ORCID:
<https://orcid.org/0000-0002-7111-0077>),
Carson Sievert [ctb] (ORCID: <https://orcid.org/0000-0002-4958-2844>),
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>),
Meta Platforms, Inc. [cph] ('React' and 'ReactDOM', bundled in
inst/lib/shiny/shinyreact.js)

Maintainer Barret Schloerke <barret@posit.co>

Repository CRAN

Date/Publication 2026-09-23 04:20:02 UTC

Contents

`page_bare` *Bare HTML page with Shiny dependencies*

Description

Escape hatch for custom setups. Wraps `shiny::bootstrapPage()`.

Usage

```
page_bare(..., title = NULL, lang = "en")
```

Arguments

`...`	Child tags or htmltools::htmlDependency objects. Named arguments pass through to `shiny::bootstrapPage()` — including its own `theme`. Deliberately not surfaced as named parameters: in the `ui.tsx` pattern the client owns styling, so Bootstrap theming is a passthrough, not part of this API. Mirrors Python's `page_bare(**kwargs)`.
`title`	Page title.
`lang`	HTML `lang` attribute.

Details

With no theme, the page carries **no Bootstrap**: only jQuery, Shiny's own JS/CSS, and a width=device-width viewport meta tag. Shiny's own default would attach Bootstrap 3 (plus its accessibility plugin, which errors against a newer jQuery), and in the ui.tsx pattern the client owns styling. Pass a theme — e.g. theme = bslib::bs_theme(), or bslib::bs_theme(version = 3) for the classic stack — to get Bootstrap back; then ... is a plain passthrough to shiny::bootstrapPage().

Value

A shiny.tag page.

Examples

```
# No Bootstrap: just jQuery, Shiny, and the children you pass.
page_bare(htmltools::tags$div(id = "root"), title = "My app")
```

---

page_react *Create a React page from conventional assets — no HTML file required*

Description

The zero-configuration page for the ui.tsx pattern: the server emits no body HTML at all. Attaches the shinyreact bundle plus your app's entry assets, discovered at www/ui.js and www/ui.css (relative to the working directory — the app directory under shiny::runApp()). Your JS owns the DOM: create and append your own mount container, e.g. ReactDOM.createRoot(document.body.appendChild(document

Usage

```
page_react(
  ...,
  src_dir = "www",
  js_file = "ui.js",
  css_file = "ui.css",
  title = NULL,
  lang = "en",
  shinyreact_js = "server"
)
```

Arguments

...	Extra children or htmltools::htmlDependency objects.
src_dir	Directory containing the assets. Defaults to "www", relative to the working directory.
js_file	JS entry filename within src_dir. Defaults to "ui.js".
css_file	CSS filename within src_dir. Defaults to "ui.css".

`title`	Page title. Defaults to the app folder's name (`src_dir`'s parent when `src_dir` is a `www` directory), or `"shinyreact-app"` when that resolves to nothing usable.
`lang`	HTML `lang` attribute.
`shinyreact_js`	Who supplies `shinyreact.js` (and `shinyreact.css`) to the page. `"server"` (the default) serves them from the shinyreact package as an htmltools::htmlDependency — what a no-build app needs, and what makes `window.shinyreact` exist. `"client"` is for an app whose own bundle imports `@posit-dev/shinyreact` and therefore ships its own copy; serving them too would put two copies of React and the hooks on one page. The `#shinyreact-config` tag is emitted either way — it carries the protocol version and any bookmark restore payload. Mirrors Python's `page_react(shinyreact_js=)`.

Details

`ui.js` is required (a missing file warns, pointing at the resolved path); `ui.css` is attached only when it exists. Both are served as an htmltools::htmlDependency versioned by `ui.js`'s mtime, so the browser re-fetches after every edit — unlike raw `<script src=...>` tags in a hand-written HTML file, which the browser caches.

Value

UI suitable for `shinyApp(ui = ...)`.

Examples

```
# In an app directory containing www/ui.js, `page_react()` with no
# arguments is the whole UI. Here the shipped hello example is pointed at
# explicitly:
www <- system.file("examples-shiny", "01-hello", "www", package = "shinyreact")
ui <- page_react(src_dir = www)

if (interactive()) {
  shiny::runApp(system.file("examples-shiny", "01-hello", package = "shinyreact"))
}
```

---

page_react_dep *HTML dependency for a downstream package's own JS/CSS bundle*

Description

Convenience mirroring Python's `page_react_dep()`. It is versioned by the JS file's mtime, so the `/lib/{name}-{version}/` URL changes on every rebuild and the browser re-fetches. That is what you want while developing and the wrong thing for a published package — an mtime is whatever the install happened to write, so it is neither stable across machines nor meaningful to a reader. There is no `version` argument on purpose: a package shipping a fixed version should build its own htmltools::htmlDependency (the same advice as for a classic, non-module bundle), which is five lines and leaves nothing about the dependency implicit.

Usage

```
page_react_dep(
  src_dir,
  js_file = "ui.js",
  css_file = "ui.css",
  name = basename(src_dir)
)
```

Arguments

src_dir	Directory containing the JS/CSS. Required; Python infers this from the calling module's __file__ when omitted, which R has no equivalent of.
js_file	JS filename within src_dir. Defaults to "ui.js", matching Python; attached only if the file exists.
css_file	CSS filename within src_dir. Defaults to "ui.css", matching Python; attached only if the file exists. NULL to skip.
name	Dependency name; defaults to basename(src_dir).

Details

The script tag is emitted as type="module". A classic <script defer> tag throws on the bundle's first import. type="module" is implicitly deferred, so no defer attribute is needed. If your bundle is a classic (non-module) script, build an htmltools::htmlDependency directly instead of using this helper.

Both the script and the stylesheet are attached only when the file exists inside src_dir, so a bundle that ships no CSS — or that has not been built yet — does not emit a tag pointing at a 404. Pass css_file = NULL to never attach a stylesheet. A missing js_file warns, since it is the entry point and an empty dependency would otherwise fail silently.

A missing src_dir **errors**, where a missing js_file only warns. Shiny serves the directory's files, so a directory that does not exist can only produce 404s for every asset the page references — a bug every time, and one that is far cheaper to see at page-build time than in the browser's network tab. Matches Python's page_react_dep(), which raises NotADirectoryError.

Value

An htmltools::htmlDependency.

Examples

```
www <- system.file("examples-shiny", "01-hello", "www", package = "shinyreact")
page_react_dep(www)
```

page_react_html *Serve a React* index.html *document (the* ui.tsx *pattern)*

Description

Reads a complete HTML document — the kind a Vite build emits — and injects the shinyreact page-level dependencies into it. The document must contain Shiny's dependency placeholder inside <head>:

Usage

```
page_react_html(
  path = "www/index.html",
  ...,
  extra_deps = NULL,
  shinyreact_js = "server"
)
```

Arguments

path	Path to the HTML document. Defaults to "www/index.html", relative to the working directory.
...	Ignored.
extra_deps	A list of additional htmltools::htmlDependency objects to render at the placeholder. A complete document has no tag tree to attach dependencies to, so this is the only way in — the counterpart of page_react()'s .... They render *after* Shiny's and shinyreact's, so they can rely on window.shinyreact existing. Mirrors Python's page_react_html(extra_deps=).
shinyreact_js	Who supplies shinyreact.js / shinyreact.css: "server" (the default) or "client" for an npm-tier app whose bundle imports @posit-dev/shinyreact — see page_react().

Details

<meta name="shiny-dependency-placeholder" content="">

Shiny's and shinyreact's script/link tags render in its place. It is an ordinary <meta> tag rather than template syntax, so the document stays valid HTML that a bundler's dev server can serve unchanged. Use as the ui argument: shinyApp(ui = page_react_html(), server = ...).

Assets the document references (your bundle's JS/CSS) should live in www/, where Shiny serves them statically.

For apps that don't need to own the HTML document, prefer page_react() — it requires no HTML file at all.

Value

UI suitable for shinyApp(ui = ...).

The whole document is a template

R places the dependencies with `htmltools::htmlTemplate()`, which evaluates **every** `{{ ... }}` in the document as R code — anywhere in it, `<head>` or `<body>`, with the global environment as parent. So a body containing `{{ 6*7 }}` renders `42`, and `{{ nonexistent() }}` is an error at page render.

A document written for a JS templating engine that also uses `{{ }}` (Handlebars, Mustache, Vue's text interpolation) is therefore not safe to pass here as is — those braces will be evaluated as R. Escape them, or use `page_react()`, which needs no HTML file at all.

Python's `page_react_html()` differs: it replaces the placeholder and leaves the rest of the document untouched. Documented as a deliberate divergence rather than a bug — see `FEATURES.md` and issue #223.

Path resolution

A relative `path` resolves against the process working directory. Under `shiny::runApp()` / `shiny::shinyApp()` that is the app directory, so the default `"www/index.html"` just works. R has no per-caller `__file__`, so unlike Python — which resolves a relative path against the calling module's directory — there is nothing to resolve against outside the working directory. Pass an absolute path if you need to be independent of it.

The placeholder must be spelled exactly as above — the check is a fixed-string match, so a differently-quoted or reordered `<meta>` tag is rejected.

Examples

```
index <- tempfile(fileext = ".html")
writeLines(
  c(
    "<!doctype html>",
    "<html><head>",
    '<meta name="shiny-dependency-placeholder" content="">',
    '<script type="module" src="ui.js"></script>',
    "</head><body></body></html>"
  ),
  index
)
ui <- page_react_html(index)
unlink(index)
```

`reactive_output` *Publish a reactive value to a shinyreact client (the `ui.tsx` pattern)*

Description

Server-side counterpart to `useShinyOutputValue()`. Assign to `output[[id]]`; a React client reads the value by id. There is no UI placeholder — the client owns all UI. Accepts any JSON-serializable value (passed through unchanged).

Usage

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

Arguments

expr	An expression returning a JSON-serializable value.
env	The environment in which to evaluate expr.
quoted	Is expr already quoted?

Value

A Shiny render function.

Examples

```
# The client reads this with useShinyOutputValue("greeting")
# and writes input$name with useShinyInput("name", "world").
server <- function(input, output, session) {
  output$greeting <- reactive_output({
    paste0("Hello, ", input$name, "!")
  })
}

if (interactive()) {
  shiny::shinyApp(page_react(), server)
}
```

---

send_message *Send a custom message to client React components*

Description

Messages are consumed by useShinyMessageHandler(id, handler) on the React side. The id is namespaced to the current Shiny module (if any) so module-scoped handlers match, just like input/output ids.

Usage

```
send_message(session, id, data)
```

Arguments

session	The Shiny session.
id	Message id; must match the id passed to useShinyMessageHandler() in the React component.
data	Any JSON-serializable data.

Value

Invisibly NULL.

Examples

```
# Paired with useShinyMessageHandler("notify", (msg) => ...) on the client.
server <- function(input, output, session) {
  shiny::observeEvent(input$save, {
    send_message(session, "notify", list(text = "Saved", level = "success"))
  })
}

if (interactive()) {
  shiny::shinyApp(page_react(), server)
}
```

---

wire_tap *Tap the websocket wire of a shinytest2 app*

Description

wire_tap() gives tests access to the JSON payloads that actually crossed the Shiny websocket — the contract between the server and the React client. Create the shinytest2::AppDriver with options = list(shiny.trace = TRUE) so shinytest2 records every websocket frame in app$get_logs(); the tap parses those frames into per-channel views.

Usage

```
wire_tap(app)
```

Arguments

app	A shinytest2::AppDriver started with options = list(shiny.trace = TRUE) — or any object whose $get_logs() returns a data frame with location, level, and message columns in shinytest2's shape.

Details

Cross-channel frame order (which output lands first, how outputs batch into a single values frame, busy/progress interleaving) is reactive-scheduling coincidence, not contract — so the tap deliberately does not expose a global frame stream. Within one channel (one output id, one message type, one input id) wire order is guaranteed, and the expect_* methods consume it through a cursor: each expectation scans the recorded history from just past the previous match, so values that arrive between checks are never missed — capture is lossless; polling only decides when to re-scan. Successive expectations on one channel therefore assert an ordered subsequence.

The Python counterpart is shinyreact.playwright.WireTap, with the same methods and semantics. One small divergence: jsonlite::fromJSON() maps a JSON null output value to NULL, indistinguishable from an absent key, so early output: null frames are dropped where Python sees None.

Value

A list of functions:

`all_output_values(output_id)` Every value the server delivered for `output_id`, in order.

`all_messages(message_id)` Every `send_message()` payload of `message_id`, in order.

`all_input_values(input_id)` Every value the client sent for `input_id`, in order. Matches the bare id or any `id:type` wire id (e.g. the implicit `:shinyreact.default` suffix), so use the id you wrote in `useShinyInput()`.

`expect_output_value(output_id, matcher, timeout = 10)` Retrying expectation. A function `matcher` is satisfied by a truthy return; any other object is compared with `identical()`. Returns the matched value invisibly, or errors at `timeout` (seconds). A matcher that errors on a value's shape counts as a non-match.

`expect_message(message_id, matcher, timeout = 10)` As above, for `send_message()` payloads.

`expect_input_value(input_id, matcher, timeout = 10)` As above, for client-sent input values.

Examples

```
app <- shinytest2::AppDriver$new(
  "path/to/app",
  options = list(shiny.trace = TRUE)
)
tap <- wire_tap(app)

# The JSON 30 parses to integer; identical() needs the right type.
tap$expect_input_value("bins", 30L)
tap$expect_output_value("dist_data", function(d) {
  d$breaks[[1]] == 43 && sum(unlist(d$counts)) == 272
})
```

Index