Package: shinymeta (via r-universe)

September 21, 2024

Type Package

Title Export Domain Logic from Shiny using Meta-Programming

Version 0.2.1

Description Provides tools for capturing logic in a Shiny app and exposing it as code that can be run outside of Shiny (e.g., from an R console). It also provides tools for bundling both the code and results to the end user.

URL `https://rstudio.github.io/shinymeta/`, `https://github.com/rstudio/shinymeta`

License GPL-3

Imports callr, fastmap, fs, rlang, htmltools, shiny (>= 1.6.0), sourcetools, styler, utils

Encoding UTF-8

RoxygenNote 7.3.1

Suggests knitr, stringr, rmarkdown, testthat (>= 3.0), shinyAce, clipr, dplyr, ggplot2, cranlogs, xfun, magrittr, zoo

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

Collate 'archive.R' 'display.R' 'format.R' 'imports.R' 'utils.R' 'metareactive.R' 'observe.R' 'globals.R' 'output-code.R' 'print.R' 'render.R' 'report.R' 'utils-format.R' 'zzz.R'

Repository https://posit-dev-shinycoreci.r-universe.dev

RemoteUrl https://github.com/rstudio/shinymeta

RemoteRef HEAD

RemoteSha 3b7f077b182c54f0763b47c18cdabf86528a2e1a

Contents

`buildScriptBundle` *Produce a zip bundle of code and results*

Description

Produce a zip bundle of code and results

Usage

```
buildScriptBundle(
  code = NULL,
  output_zip_path,
  script_name = "script.R",
  include_files = list(),
  render = TRUE,
  render_args = list()
)

buildRmdBundle(
  report_template,
  output_zip_path,
  vars = list(),
  include_files = list(),
  render = TRUE,
  render_args = list()
)
```

Arguments

`code`	A language object.
`output_zip_path`	A filename for the resulting zip bundle.
`script_name`	A name for the R script in the zip bundle.

`include_files`	A named list consisting of additional files that should be included in the zip bundle. The element names indicate the destination path within the bundle, specified as a relative path; the element values indicate the path to the actual file currently on disk, specified as either a relative or absolute path.
`render`	Whether or not to call `rmarkdown::render()` on the R script.
`render_args`	Arguments to provide to `rmarkdown::render()`.
`report_template`	Filename of an Rmd template to be expanded by `knitr::knit_expand()`.
`vars`	A named list of variables passed along to `...` in `knitr::knit_expand()`.

Value

The path to a generated file.

See Also

knitr::knit_expand

`displayCodeModal` *Display a shinyAce code editor via shiny modal*

Description

Show a `shinyAce::aceEditor()` in a `shiny::modalDialog()`.

Usage

```
displayCodeModal(
  code,
  title = NULL,
  clip = "clipboard",
  footer = shiny::modalButton("Dismiss"),
  size = c("m", "s", "l"),
  easyClose = TRUE,
  fade = TRUE,
  session = shiny::getDefaultReactiveDomain(),
  ...
)
```

Arguments

`code`	Either a language object or a character string.
`title`	An optional title for the dialog.
`clip`	An `icon()` name that a user can press to copy code to the clipboard. If you wish to not have an icon, specify `clip = NULL`.
`footer`	UI for footer. Use `NULL` for no footer.

`size`	One of `"s"` for small, `"m"` (the default) for medium, `"l"` for large, or `"xl"` for extra large. Note that `"xl"` only works with Bootstrap 4 and above (to opt-in to Bootstrap 4+, pass `bslib::bs_theme()` to the `theme` argument of a page container like `fluidPage()`).
`easyClose`	If `TRUE`, the modal dialog can be dismissed by clicking outside the dialog box, or be pressing the Escape key. If `FALSE` (the default), the modal dialog can't be dismissed in those ways; instead it must be dismissed by clicking on a `modalButton()`, or from a call to `removeModal()` on the server.
`fade`	If `FALSE`, the modal dialog will have no fade-in animation (it will simply appear rather than fade in to view).
`session`	a shiny session object (the default should almost always be used).
`...`	arguments passed along to `shinyAce::aceEditor()`

Value

nothing. Call this function for its side effects.

See Also

outputCodeButton

Examples

```
if (interactive()) {
  library(shiny)
  ui <- fluidPage(
    sliderInput("n", label = "Number of samples", min = 10, max = 100, value = 30),
    actionButton("code", icon("code")),
    plotOutput("p")
  )
  server <- function(input, output) {
    output$p <- metaRender(renderPlot, {
      plot(sample(..(input$n)))
    })
    observeEvent(input$code, {
      code <- expandChain(output$p())
      displayCodeModal(code)
    })
  }
  shinyApp(ui, server)
}
```

`expandChain` *Expand code objects*

Description

Use `expandChain` to write code out of one or more metaReactive objects. Each meta-reactive object (expression, observer, or renderer) will cause not only its own code to be written, but that of its dependencies as well.

Usage

```
newExpansionContext()

expandChain(..., .expansionContext = newExpansionContext())
```

Arguments

`...`	All arguments must be unnamed, and must be one of: 1) calls to meta-reactive objects, 2) comment string (e.g. `"# A comment"`), 3) language object (e.g. `quote(print(1 + 1))`), or 4) `NULL` (which will be ignored). Calls to meta-reactive objects can optionally be `invisible()`, see Details.
`.expansionContext`	Accept the default value if calling `expandChain` a single time to generate a corpus of code; or create an expansion context object using `newExpansionContext()` and pass it to multiple related calls of `expandChain`. See Details.

Details

There are two ways to extract code from meta objects (i.e. `metaReactive()`, `metaObserve()`, and `metaRender()`): `withMetaMode()` and `expandChain()`. The simplest is `withMetaMode(obj())`, which crawls the tree of meta-reactive dependencies and expands each `..()` in place.

For example, consider these meta objects:

```
    nums <- metaReactive({ runif(100) })
    obs <- metaObserve({
      summary(..(nums()))
      hist(..(nums()))
    })
```

When code is extracted using `withMetaMode`:

```
    withMetaMode(obs())
```

The result looks like this:

```
    summary(runif(100))
    plot(runif(100))
```

Notice how `runif(100)` is inlined wherever `..(nums())` appears, which is not desirable if we wish to reuse the same values for `summary()` and `plot()`.

The `expandChain` function helps us workaround this issue by assigning return values of `metaReactive()` expressions to a name, then replaces relevant expansion (e.g., `..(nums())`) with the appropriate name (e.g. `nums`).

```
expandChain(obs())
```

The result looks like this:

```
nums <- runif(100)
summary(nums)
plot(nums)
```

You can pass multiple meta objects and/or comments to `expandChain`.

```
expandChain(
  "# Generate values",
  nums(),
  "# Summarize and plot",
  obs()
)
```

Output:

```
# Load data
nums <- runif(100)
nums
# Inspect data
summary(nums)
plot(nums)
```

You can suppress the printing of the `nums` vector in the previous example by wrapping the `nums()` argument to `expandChain()` with `invisible(nums())`.

Value

The return value of `expandChain()` is a code object that's suitable for printing or passing to `displayCodeModal()`, `buildScriptBundle()`, or `buildRmdBundle()`.

The return value of `newExpansionContext` is an object that should be passed to multiple `expandChain()` calls.

Preserving dependencies between `expandChain()` calls

Sometimes we may have related meta objects that we want to generate code for, but we want the code for some objects in one code chunk, and the code for other objects in another code chunk; for example, you might be constructing an R Markdown report that has a specific place for each code chunk.

Within a single `expandChain()` call, all `metaReactive` objects are guaranteed to only be declared once, even if they're declared on by multiple meta objects; but since we're making two `expandChain()` calls, we will end up with duplicated code. To remove this duplication, we need the second `expandChain` call to know what code was emitted in the first `expandChain` call.

We can achieve this by creating an "expansion context" and sharing it between the two calls.

```
exp_ctx <- newExpansionContext()
chunk1 <- expandChain(.expansionContext = exp_ctx,
  invisible(nums())
)
chunk2 <- expandChain(.expansionContext = exp_ctx,
  obs()
)
```

After this code is run, `chunk1` contains only the definition of `nums` and `chunk2` contains only the code for `obs`.

Substituting `metaReactive` objects

Sometimes, when generating code, we want to completely replace the implementation of a `metaReactive`. For example, our Shiny app might contain this logic, using `shiny::fileInput()`:

```
data <- metaReactive2({
  req(input$file_upload)
  metaExpr(read.csv(..(input$file_upload$datapath)))
})
obs <- metaObserve({
  summary(..(data()))
})
```

Shiny's file input works by saving uploading files to a temp directory. The file referred to by `input$file_upload$datapath` won't be available when another user tries to run the generated code.

You can use the expansion context object to swap out the implementation of `data`, or any other `metaReactive`:

```
ec <- newExpansionContext()
ec$substituteMetaReactive(data, function() {
  metaExpr(read.csv("data.csv"))
})

expandChain(.expansionContext = ec, obs())
```

Result:

```
data <- read.csv("data.csv")
summary(data)
```

Just make sure this code ends up in a script or Rmd bundle that includes the uploaded file as `data.csv`, and the user will be able to reproduce your analysis.

The `substituteMetaReactive` method takes two arguments: the `metaReactive` object to substitute, and a function that takes zero arguments and returns a quoted expression (for the nicest looking results, use `metaExpr` to create the expression). This function will be invoked the first time the `metaReactive` object is encountered (or if the `metaReactive` is defined with `inline = TRUE`, then every time it is encountered).

References

https://rstudio.github.io/shinymeta/articles/code-generation.html

Examples

```
input <- list(dataset = "cars")

# varname is only required if srcref aren't supported
# (R CMD check disables them for some reason?)
mr <- metaReactive({
  get(..(input$dataset), "package:datasets")
})

top <- metaReactive({
  head(..(mr()))
})

bottom <- metaReactive({
  tail(..(mr()))
})

obs <- metaObserve({
  message("Top:")
  summary(..(top()))
  message("Bottom:")
  summary(..(bottom()))
})

# Simple case
expandChain(obs())

# Explicitly print top
expandChain(top(), obs())

# Separate into two code chunks
exp_ctx <- newExpansionContext()
expandChain(.expansionContext = exp_ctx,
  invisible(top()),
  invisible(bottom()))
expandChain(.expansionContext = exp_ctx,
  obs())
```

---

formatCode — *Deparse and format shinymeta expressions*

Description

Turn unevaluated shinymeta expressions into (formatted or styled) text.

Usage

```
formatCode(code, width = 500L, formatter = styleText, ...)

styleText(code, ...)

deparseCode(code, width = 500L)
```

Arguments

`code`	Either an unevaluated expression or a deparsed code string.
`width`	The `width.cutoff` to use when `deparse()`-ing the code expression.
`formatter`	a function that accepts deparsed code (a character string) as the first argument.
`...`	arguments passed along to the `formatter` function.

Details

Before any formatting takes place, the unevaluated expression is deparsed into a string via `deparseCode()`, which ensures that shinymeta comment strings (i.e., literal strings that appear on their own line, and begin with one or more # characters.) are turned into comments and superfluous `\{` are removed. After deparsing, the `formatCode()` function then calls the `formatter` function on the deparsed string to format (aka style) the code string. The default `formatter`, `styleText()`, uses `styler::style_text()` with a couple differences:

- Pipe operators (`%>%`) are *always* followed by a line break.
- If the token appearing after a line-break is a comma/operator, the line-break is removed.

Value

Single-element character vector with formatted code

Examples

```
options(shiny.suppressMissingContextError = TRUE)

x <- metaReactive({
  "# Here's a comment"
  sample(5) %>% sum()
})

code <- expandChain(x())

deparseCode(code)
formatCode(code)
formatCode(code, formatter = styler::style_text)
```

`metaAction` *Run/capture non-reactive code for side effects*

Description

Most apps start out with setup code that is non-reactive, such as `library()` calls, loading of static data into local variables, or `source`-ing of supplemental R scripts. `metaAction` provides a convenient way to run such code for its side effects (including declaring new variables) while making it easy to export that code using `expandChain()`. Note that `metaAction` executes code directly in the `env` environment (which defaults to the caller's environment), so any local variables that are declared in the `expr` will be available outside of `metaAction` as well.

Usage

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

Arguments

`expr`	A code expression that will immediately be executed (before the call to `metaAction` returns), and also stored for later retrieval (i.e. meta mode).
`env`	An environment.
`quoted`	Is the expression quoted? This is useful when you want to use an expression that is stored in a variable; to do so, it must be quoted with `quote()`.

Value

A function that, when called in meta mode (i.e. inside `expandChain()`), will return the code in quoted form. If this function is ever called outside of meta mode, it throws an error, as it is definitely being called incorrectly.

Examples

```
setup <- metaAction({
  library(stats)

  "# Set the seed to ensure repeatable randomness"
  set.seed(100)

  x <- 1
  y <- 2
})

# The action has executed
print(x)
print(y)

# And also you can emit the code
expandChain(
  setup()
)
```

---

metaExpr *Mark an expression as a meta-expression*

Description

Mark an expression as a meta-expression

Usage

```
metaExpr(
  expr,
  env = parent.frame(),
  quoted = FALSE,
  localize = "auto",
  bindToReturn = FALSE
)
```

Arguments

`expr`	An expression (quoted or unquoted).
`env`	An environment.
`quoted`	Is the expression quoted? This is useful when you want to use an expression that is stored in a variable; to do so, it must be quoted with `quote()`.
`localize`	Whether or not to wrap the returned expression in `local()`. The default, `"auto"`, only wraps expressions with a top-level `return()` statement (i.e., return statements in anonymized functions are ignored).
`bindToReturn`	For non-`localized` expressions, should an assignment of a meta expression be applied to the *last child* of the top-level `\{` call?

Value

If inside meta mode, a quoted form of `expr` for use inside of `metaReactive2()`, `metaObserve2()`, or `metaRender2()`. Otherwise, in normal execution, the result of evaluating `expr`.

See Also

`metaReactive2()`, `metaObserve2()`, `metaRender2()`, `..`

---

metaObserve *Create a meta-reactive observer*

Description

Create a `observe()`r that, when invoked with meta-mode activated (i.e. called within `withMetaMode()` or `expandChain()`), returns a partially evaluated code expression. Outside of meta-mode, `metaObserve()` is equivalent to `observe()` (it fully evaluates the given expression).

Usage

```
metaObserve(
  expr,
  env = parent.frame(),
  quoted = FALSE,
  label = NULL,
  domain = getDefaultReactiveDomain(),
  localize = "auto",
  bindToReturn = FALSE
)

metaObserve2(
  expr,
  env = parent.frame(),
  quoted = FALSE,
  label = NULL,
  domain = getDefaultReactiveDomain()
)
```

Arguments

`expr`	An expression (quoted or unquoted).
`env`	The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression. If `x` is a quosure and `quoted` is `TRUE`, then `env` is ignored.
`quoted`	If it is `TRUE`, then the `quote()`ed value of `x` will be used when `x` is evaluated. If `x` is a quosure and you would like to use its expression as a value for `x`, then you must set `quoted` to `TRUE`.

`label`	A label for the observer, useful for debugging.
`domain`	See domains.
`localize`	Whether or not to wrap the returned expression in `local()`. The default, `"auto"`, only wraps expressions with a top-level `return()` statement (i.e., return statements in anonymized functions are ignored).
`bindToReturn`	For non-`localized` expressions, should an assignment of a meta expression be applied to the *last child* of the top-level \{ call?

Details

If you wish to capture specific code inside of `expr` (e.g. ignore code that has no meaning outside shiny, like `req()`), use `metaObserve2()` in combination with `metaExpr()`. When using `metaObserve2()`, `expr` must return a `metaExpr()`.

Value

A function that, when called in meta mode (i.e. inside `expandChain()`), will return the code in quoted form. If this function is ever called outside of meta mode, it throws an error, as it is definitely being called incorrectly.

See Also

`metaExpr()`, `..`

Examples

```
# observers execute 'immediately'
x <- 1
mo <- metaObserve({
  x <<- x + 1
})
getFromNamespace("flushReact", "shiny")()
print(x)

# It only makes sense to invoke an meta-observer
# if we're in meta-mode (i.e., generating code)
expandChain(mo())

# Intentionally produces an error
## Not run: mo()
```

`metaReactive`	*Create a meta-reactive expression*

Description

Create a `reactive()` that, when invoked with meta-mode activated (i.e. called within `withMetaMode()` or `expandChain()`), returns a code expression (instead of evaluating that expression and returning the value).

Usage

```
metaReactive(
  expr,
  env = parent.frame(),
  quoted = FALSE,
  varname = NULL,
  domain = shiny::getDefaultReactiveDomain(),
  inline = FALSE,
  localize = "auto",
  bindToReturn = FALSE
)

metaReactive2(
  expr,
  env = parent.frame(),
  quoted = FALSE,
  varname = NULL,
  domain = shiny::getDefaultReactiveDomain(),
  inline = FALSE
)
```

Arguments

`expr`	An expression (quoted or unquoted).
`env`	The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression. If `x` is a quosure and `quoted` is `TRUE`, then `env` is ignored.
`quoted`	If it is `TRUE`, then the `quote()`ed value of `x` will be used when `x` is evaluated. If `x` is a quosure and you would like to use its expression as a value for `x`, then you must set `quoted` to `TRUE`.
`varname`	An R variable name that this object prefers to be named when its code is extracted into an R script. (See also: `expandChain()`)
`domain`	See domains.
`inline`	If `TRUE`, during code expansion, do not declare a variable for this object; instead, inline the code into every call site. Use this to avoid introducing variables for very simple expressions. (See also: `expandChain()`)

`localize`	Whether or not to wrap the returned expression in `local()`. The default, `"auto"`, only wraps expressions with a top-level `return()` statement (i.e., return statements in anonymized functions are ignored).
`bindToReturn`	For non-`localized` expressions, should an assignment of a meta expression be applied to the *last child* of the top-level \{ call?

Details

If you wish to capture specific code inside of `expr` (e.g. ignore code that has no meaning outside shiny, like `req()`), use `metaReactive2()` in combination with `metaExpr()`. When using `metaReactive2()`, `expr` must return a `metaExpr()`.

If `varname` is unspecified, srcrefs are used in attempt to infer the name bound to the meta-reactive object. In order for this inference to work, the `keep.source` option must be `TRUE` and `expr` must begin with \{.

Value

A function that, when called in meta mode (i.e. inside `expandChain()`), will return the code in quoted form. When called outside meta mode, it acts the same as a regular `shiny::reactive()` expression call.

See Also

`metaExpr()`, `..`

Examples

```
library(shiny)
options(shiny.suppressMissingContextError = TRUE)

input <- list(x = 1)

y <- metaReactive({
  req(input$x)
  a <- ..(input$x) + 1
  b <- a + 1
  c + 1
})

withMetaMode(y())
expandChain(y())

y <- metaReactive2({
  req(input$x)

  metaExpr({
    a <- ..(input$x) + 1
    b <- a + 1
    c + 1
  }, bindToReturn = TRUE)
})
```

```
expandChain(y())
```

metaRender *Create a meta-reactive output*

Description

Create a meta-reactive output that, when invoked with meta-mode activated (i.e. called within `expandChain()` or `withMetaMode()`), returns a code expression (instead of evaluating that expression and returning the value).

Usage

```
metaRender(
  renderFunc,
  expr,
  ...,
  env = parent.frame(),
  quoted = FALSE,
  localize = "auto",
  bindToReturn = FALSE
)

metaRender2(renderFunc, expr, ..., env = parent.frame(), quoted = FALSE)
```

Arguments

`renderFunc`	A reactive output function (e.g., shiny::renderPlot, shiny::renderText, shiny::renderUI, etc).
`expr`	An expression that generates given output expected by `renderFunc`.
`...`	Other arguments passed along to `renderFunc`.
`env`	The parent environment for the reactive expression. By default, this is the calling environment, the same as when defining an ordinary non-reactive expression. If `x` is a quosure and `quoted` is `TRUE`, then `env` is ignored.
`quoted`	If it is `TRUE`, then the `quote()`ed value of `x` will be used when `x` is evaluated. If `x` is a quosure and you would like to use its expression as a value for `x`, then you must set `quoted` to `TRUE`.
`localize`	Whether or not to wrap the returned expression in `local()`. The default, `"auto"`, only wraps expressions with a top-level `return()` statement (i.e., return statements in anonymized functions are ignored).
`bindToReturn`	For non-`localized` expressions, should an assignment of a meta expression be applied to the *last child* of the top-level `\{` call?

Details

If you wish to capture specific code inside of `expr` (e.g. ignore code that has no meaning outside shiny, like `req()`), use `metaRender2()` in combination with `metaExpr()`. When using `metaRender2()`, `expr` must return a `metaExpr()`.

Since package authors are allowed to create their own output rendering functions, creating a meta-counterpart of an output renderer (e.g. `renderPlot()`) needs to be more general than prefixing `meta` to the function name (as with `metaReactive()` and `metaObserve()`). `metaRender()` makes some assumptions about the arguments taken by the render function, assumptions that we believe are true for all existing render functions. If you encounter a render function that doesn't seem to work properly, please let us know by filing an issue on GitHub.

Value

An annotated render function, ready to be assigned to an output slot. The function may also be called in meta mode (i.e., inside `expandChain()`) to return the code in quoted form.

See Also

`metaExpr()`, `..`

Examples

```
if (interactive()) {
  library(shiny)
  library(shinymeta)

  ui <- fluidPage(
    selectInput("var", label = "Choose a variable", choices = names(cars)),
    verbatimTextOutput("Summary"),
    verbatimTextOutput("code")
  )

  server <- function(input, output) {
    var <- metaReactive({
      cars[[..(input$var)]]
    })
    output$Summary <- metaRender(renderPrint, {
      summary(..(var()))
    })
    output$code <- renderPrint({
      expandChain(output$Summary())
    })
  }

  shinyApp(ui, server)
}
```

`outputCodeButton` *Overlay an icon on a shiny output*

Description

Intended for overlaying a button over a shiny output, that when clicked, displays code for reproducing that output. The button is similar to an `shiny::actionButton()`, but instead of providing an `inputId`, the id is determined by the id of the `outputObj`. The name of that input is a function of `outputObj`'s `outputId`: `input$OUTPUTID_output_code`.

Usage

```
outputCodeButton(
  outputObj,
  label = "Show code",
  icon = shiny::icon("code"),
  width = NULL,
  ...
)
```

Arguments

`outputObj`	A shiny output container (e.g., shiny::plotOutput, shiny::textOutput, etc)
`label`	The contents of the button or link–usually a text label, but you could also use any other HTML, like an image.
`icon`	An optional `icon()` to appear on the button.
`width`	The width of the input, e.g. `'400px'`, or `'100%'`; see `validateCssUnit()`.
`...`	Named attributes to be applied to the button or link.

Value

the `outputObj` wrapped in a card-like HTML container.

See Also

displayCodeModal

Examples

```
if (interactive()) {
  library(shiny)
  ui <- fluidPage(
    sliderInput("n", label = "Number of samples", min = 10, max = 100, value = 30),
    outputCodeButton(plotOutput("p"))
  )
  server <- function(input, output) {
    output$p <- metaRender(renderPlot, {
      plot(sample(..(input$n)))
    })
    observeEvent(input$p_output_code, {
      code <- expandChain(output$p())
      displayCodeModal(code)
    })
  }
  shinyApp(ui, server)
}
```

withMetaMode *Evaluate an expression with meta mode activated*

Description

Evaluate an expression with meta mode activated

Usage

```
withMetaMode(expr, mode = TRUE)
```

Arguments

expr	an expression.
mode	whether or not to evaluate expression in meta mode.

Value

The result of evaluating `expr`.

See Also

`expandChain()`

Index