shiny.worker: How Does It Work?

Here is the code of the shiny.worker demo app:

```
library(shiny)
library(shiny.info)

ui <- fluidPage(theme = "style.css",

  titlePanel("shiny.worker"),
  shiny.info::powered_by("shiny.worker", link = "https://appsilon.com
/shiny"),

  sidebarLayout(
    sidebarPanel(
      div("Play with the slider. Histogram will be still responsive,
even if job is running:"), br(),
      sliderInput("bins", "Number of bins:", min = 1, max = 50, value
= 30),
      div("Then try to run new job again:"), br(),
      actionButton("triggerButton", "Run job (5 sec.)")
    ),
    mainPanel(
      fluidRow(
        column(6, plotOutput("distPlot")),
        column(6,
           uiOutput("loader"),
           plotOutput("futurePlot")
        )
      )
    )
  )
)

server <- function(input, output) {

  output$distPlot <- renderPlot({
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })

  plotValuesPromise <- shiny.worker::job(
    job = function(args) {
      Sys.sleep(5)
      cbind(rnorm(args$n), rnorm(args$n))
    },
    trigger_args = reactive({
      input$triggerButton
      list(n = 1000)
    }),
    cancel_active_job_on_input_change = FALSE, # ignore input change,
wait until resolved
    value_until_not_resolved = NULL
  )
```

```
  output$loader <- renderUI({
    task <- plotValuesPromise()
    if (!task$resolved) {
      div(
        div(class = "loader-text", "Job is running..."),
        div(class = "loader")
      )
    }
  })

  output$futurePlot <- renderPlot({
    task <- plotValuesPromise()
    if (task$resolved) {
      plot(task$result, main = "There you go")
    }
  })

}

shiny.worker::init()
shinyApp(ui = ui, server = server)
```

The key fragment is the line with the *shiny.worker::job()* call where you schedule a job. The job is your long running calculation. When it was a regular reactive, it was blocking the UI, but now it is delegated to the worker. As the calculation is delegated to the worker, the UI is unfrozen while the calculation is being performed.

Arguments for the job are provided as a reactive (*trigger_args*). Its value will be passed to the job function as args. This means that every time the value of this reactive changes, shiny.worker will take action, depending on the strategy you choose. It can be triggering a new job and cancelling a running job or ignoring the change (no new job is scheduled until it is resolved). It is the developer's responsibility to implement app logic to avoid potential race conditions.

To access the worker's result, you call it like you do with a reactive (*plotValuesPromise()*). As a result you are able to read its state (*task$resolved*) and returned value (*task$result*). You decide what should be returned when the job is still running with the argument *value_until_not_resolved*.

How Can I Start Using shiny.worker?

shiny.worker has not yet been released to the public, but we have made it available to several of our clients. So far, we've seen dramatic improvements in UX after implementing shiny.worker.