A promising approach for more responsive Shiny apps

Async programming

Sound complicated?

It is!

But when you need it, you **really** need it.

Why would I need it?

R performs tasks one at a time (“single threaded”).

While your Shiny app process is busy doing a long running calculation, it can’t do anything else.

At all.

Example

```
# time = 0:00.000
trainModel(Sonar, "Class")
# time = 0:15.553, ouch!
```

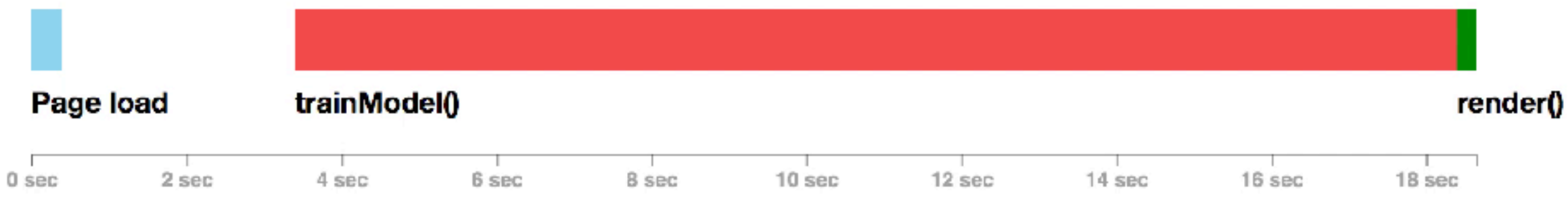
Example

```
ui <- basicPage(
  h2("Synchronous training"),
  actionButton("train", "Train"),
  verbatimTextOutput("summary"),
  plotOutput("plot")
)

server <- function(input, output, session) {
  model <- eventReactive(input$train, {
    trainModel(Sonar, "Class") # Super slow!
  })

  output$summary <- renderPrint({
    print(model())
  })

  output$plot <- renderPlot({
    plot(model())
  })
}
```

Demo

Synchronous

```
# time = 0:00.000
trainModel(Sonar, "Class")
# time = 0:15.553
```



Demo

Async to the rescue

Perform long-running tasks asynchronously: start the task but don’t wait around for the result. This leaves R free to continue doing other things.

We need to:

1. Launch tasks that run **away from the main R thread**

2. Be able to do something with the result (if success) or error (if failure), when the tasks completes, **back on the main R thread**

1. Launch async tasks

```
library(future)
plan(multiprocess)

# time = 0:00.000
f <- future(trainModel(Sonar, "Class"))
# time = 0:00.062
```

Potentially lots of ways to do this, but currently using the **`future`** package by Henrik Bengtsson.

Runs R code in a separate R process, freeing up the original R process.

1. Launch async tasks

```
library(future)
plan(multiprocess)

# time = 0:00.000
f <- future(trainModel(Sonar, "Class"))
# time = 0:00.062
value(f)
# time = 0:15.673
```

However, future's API for **retrieving** values (`value(f)`) is not what we want, as it is blocking: you run tasks asynchronously, but access their results synchronously

2. Do something with the results

The new **`promises`** package lets you access the results from async tasks.

A promise object represents the **eventual result** of an async task. It's an R6 object that knows:

1. Whether the task is running, succeeded, or failed

2. The result (if succeeded) or error (if failed)

Every function that runs an async task, should return a promise object, instead of regular data.

Promises

Directly inspired by JavaScript promises (plus some new features for smoother R and Shiny integration)

They work well with Shiny, but are generic—no part of promises is Shiny-specific

(Not the same as R's promises for delayed evaluation. Sorry about the name collision.)

Also known as tasks (C#), futures (Scala, Python), and CompletableFutures (Java 😂)

How don’t promises work?

You **cannot** wait for a promise to finish

You **cannot** ask a promise if it’s done

You **cannot** ask a promise for its value

How do promises work?

Instead of extracting the value out of a promise, you *chain* whatever operation you were going to do to the result, to the promise.

Sync (without promises):

```
query_db() %>%
  filter(cyl > 4) %>%
  head(10) %>%
  View()
```

How do promises work?

Instead of extracting the value out of a promise, you *chain* whatever operation you were going to do to the result, to the promise.

Async (with promises):

```
future(query_db()) %...>%
  filter(cyl > 4) %...>%
  head(10) %...>%
  View()
```

The promise pipe operator

```
promise %...>% (function(result) {
  # Do stuff with the result
})
```

The `%...>%` is the “promise pipe”, a promise-aware version of `%>%`.

Its left operand must be a promise (or, for convenience, a Future), and it returns a promise.

You don’t use `%...>%` to pull future values into the present, but to push subsequent computations into the future.

Asynchronous

```
# time = 0:00.000
future(trainModel(Sonar, "Class")) %...>%
  print() # time = 0:15.673
# time = 0:00.062
```

Demo

❌ Sync

```
# time = 0:00.000
trainModel(Sonar, "Class")
# time = 0:15.553
```

❌ Future

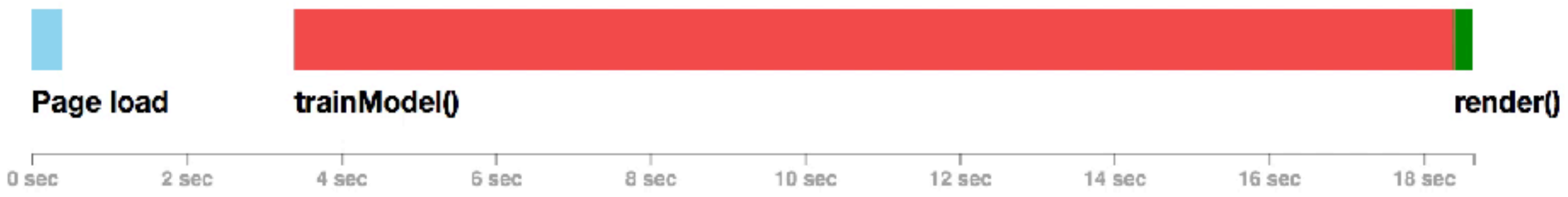
```
# time = 0:00.000
f <- future(trainModel(Sonar, "Class"))
# time = 0:00.062
value(f)
# time = 0:15.673
```

💍 Future + promises

```
# time = 0:00.000
future(trainModel(Sonar, "Class")) %...>%
  print() # time = 0:15.673
# time = 0:00.062
```

Asynchronous

```
# time = 0:00.000
future(trainModel(Sonar, "Class")) %...>%
  print() # time = 0:15.673
# time = 0:00.062
```



Example 2

```
ui <- basicPage(
  h2("Asynchronous training"),
  actionButton("train", "Train"),
  verbatimTextOutput("summary"),
  plotOutput("plot")
)

server <- function(input, output, session) {
  model <- eventReactive(input$train, {
    future(trainModel(Sonar, "Class")) # So fast!
  })

  output$summary <- renderPrint({
    model() %...>% print()
  })

  output$plot <- renderPlot({
    model() %...>% plot()
  })
}
```

Demo

Current status

- You must install Shiny from a branch for async support: **`install_github("rstudio/shiny")`**
- Documentation at https://rstudio.github.io/promises
- We want your testing/feedback before we release to CRAN

Other topics (covered in docs)

- Several other promise operators besides `%...>%`
- Promises and reactivity
- Error handling (promise equivalents to try, catch, finally)
- Composing promises and working with collections of promises
- Other syntax options

Thank you

https://speakerdeck.com/jcheng5/r-promises
https://github.com/nwstephens/shiny-async
https://github.com/rstudio/promises
https://www.rstudio.com/resources/videos/scaling-shiny-apps-with-async-programming/