Introduction to R Shiny

Elie Arnaud (PNDB)

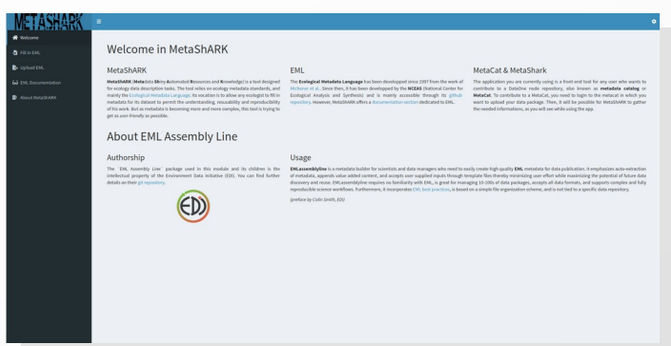
EcoInfoFAIR 2020

Introduction

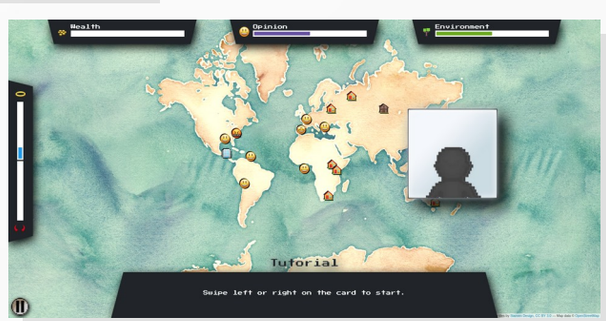
Shiny is …

- A R package: {shiny}
- An R – HTML interface
- A communication tool
- A learning tool

SeaClone – Ocean Hackathon 2020



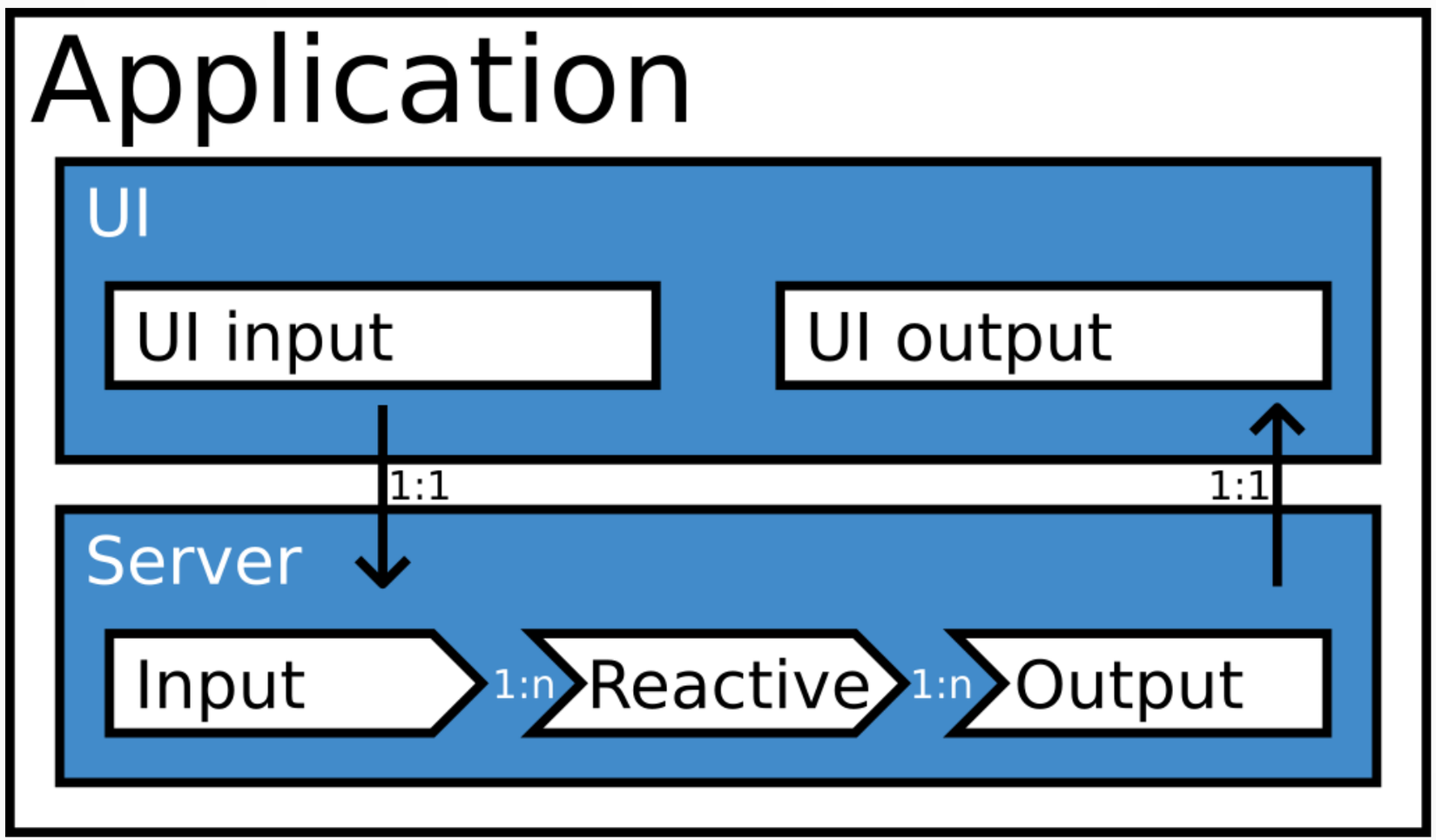
MetaShARK - PNDB



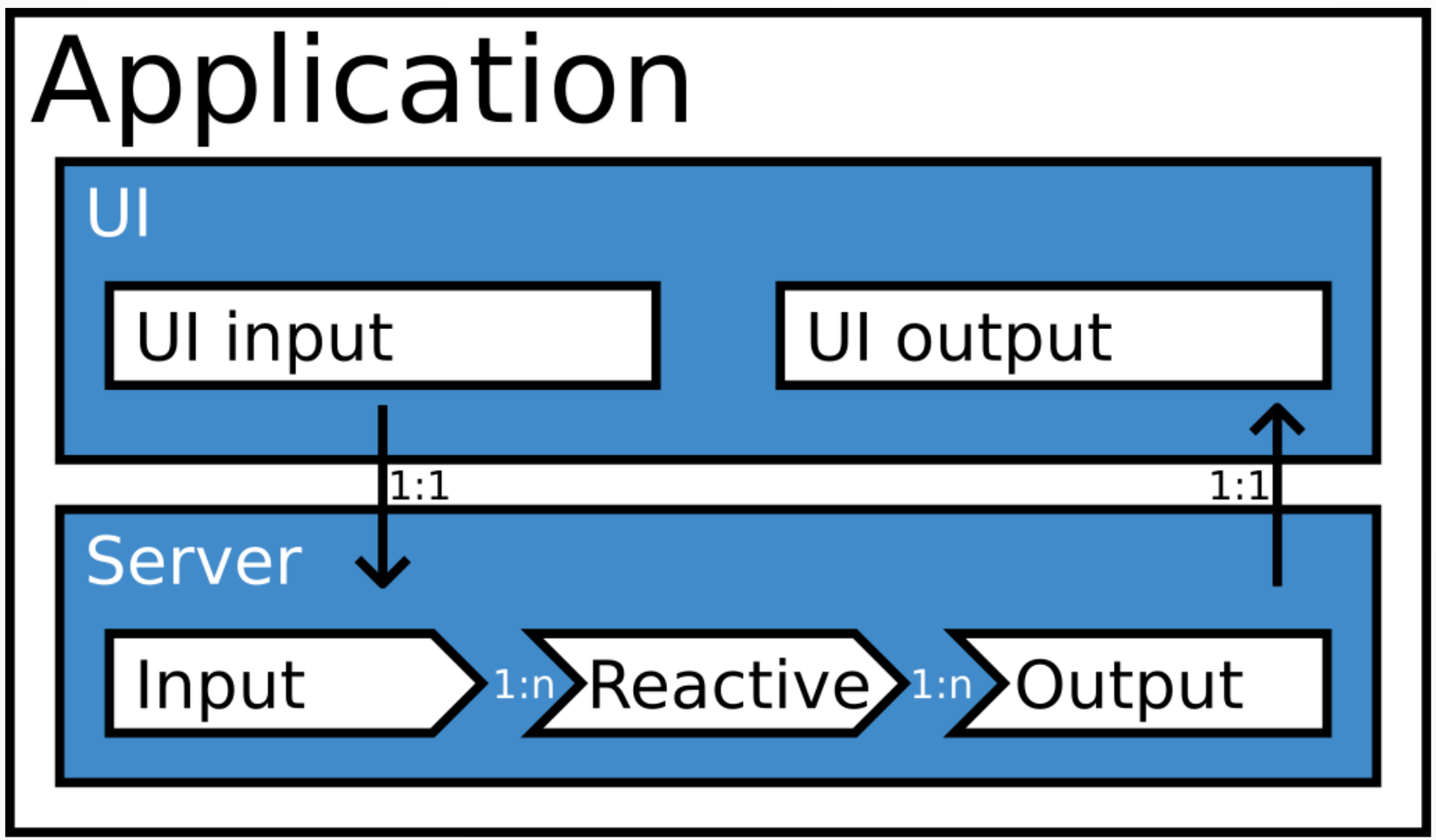
Shiny Decisions – by pedrocoutinhosilva

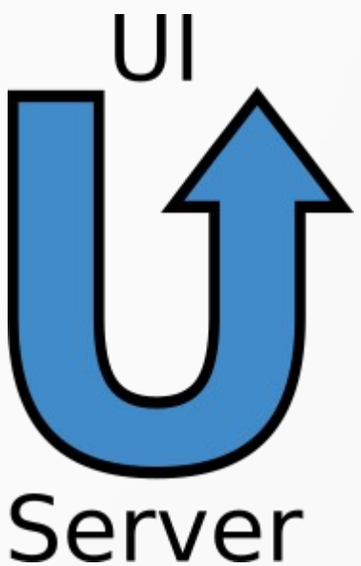
The “U” of Shiny

The “U” of Shiny

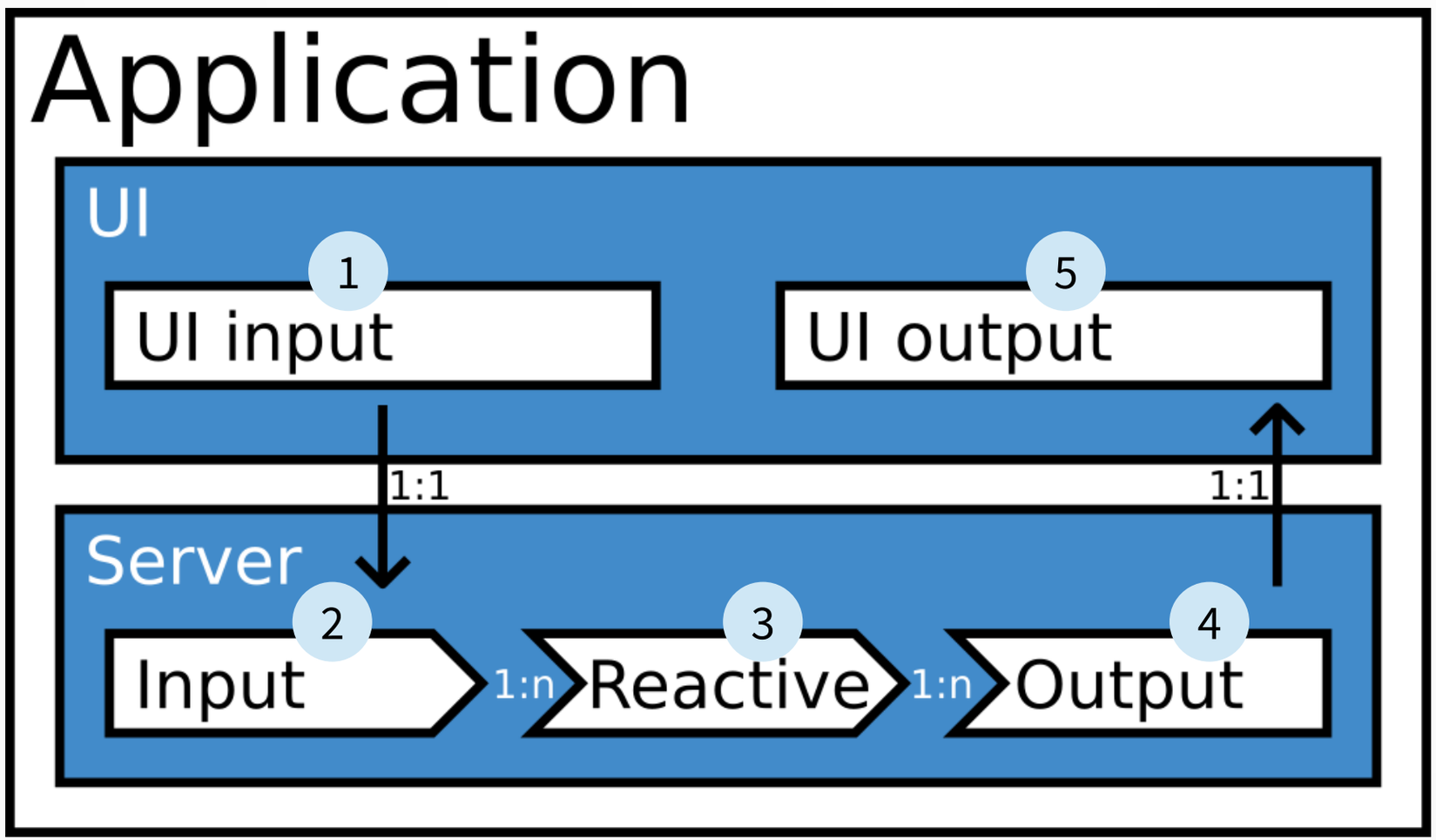


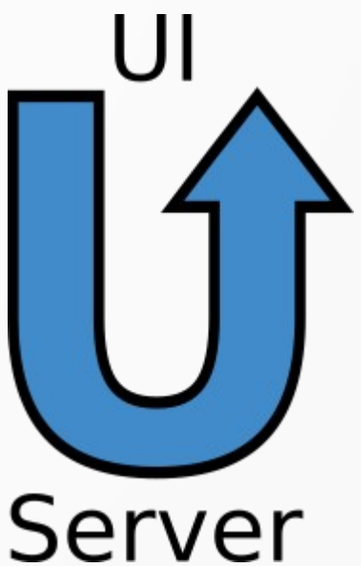
The “U” of Shiny





The “U” of Shiny





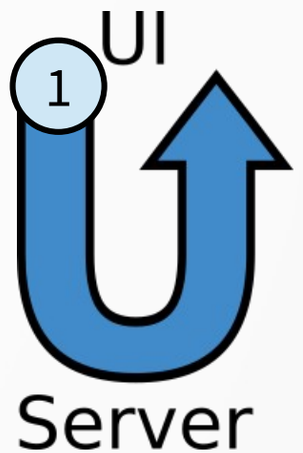
The “U” of Shiny

1. UI Input
2. Input
3. Reactive
4. Output
5. UI Output

```
# App.R
ui ← fluidPage(
  textInput(“text”, “Type here”),
  textOutput(“written”)
)

server ← function(input, output, session){
  my.input ← reactive({input$text})
  output$written ← renderText({my.input()})
}

shinyApp(ui, server)
```



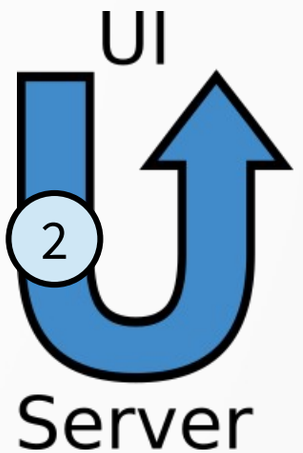
The “U” of Shiny

1. UI Input
2. **Input**
3. Reactive
4. Output
5. UI Output

```
# App.R
ui ← fluidPage(
  textInput(“text”, “Type here”),
  textOutput(“written”)
)

server ← function(input, output, session){
  my.input ← reactive({input$text})
  output$written ← renderText({my.input()})
}

shinyApp(ui, server)
```



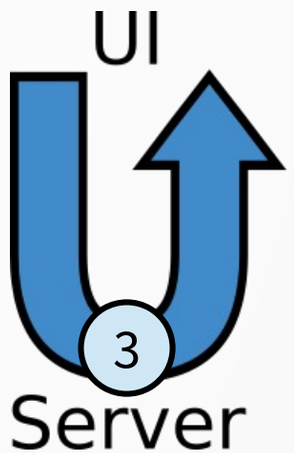
The “U” of Shiny

1. UI Input
2. Input
3. **Reactive**
4. Output
5. UI Output

```
# App.R
ui ← fluidPage(
  textInput(“text”, “Type here”),
  textOutput(“written”)
)

server ← function(input, output, session){
  my.input ← reactive({input$text})
  output$written ← renderText({my.input()})
}

shinyApp(ui, server)
```



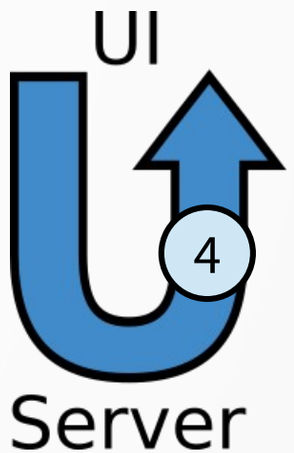
The “U” of Shiny

1. UI Input
2. Input
3. Reactive
4. **Output**
5. UI Output

```
# App.R
ui ← fluidPage(
  textInput(“text”, “Type here”),
  textOutput(“written”)
)

server ← function(input, output, session){
  my.input ← reactive({input$text})
  output$written ← renderText({my.input()})
}

shinyApp(ui, server)
```



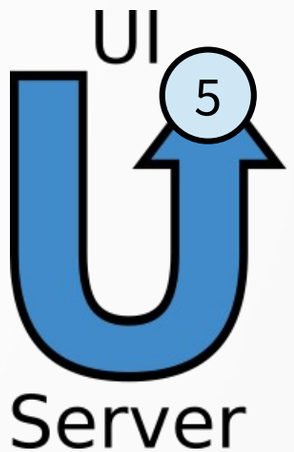
The “U” of Shiny

1. UI Input
2. Input
3. Reactive
4. Output
5. **UI Output**

```
# App.R
ui ← fluidPage(
  textInput(“text”, “Type here”),
  textOutput(“written”)
)

server ← function(input, output, session){
  my.input ← reactive({input$text})
  output$written ← renderText({my.input()})
}

shinyApp(ui, server)
```



The “U” of Shiny

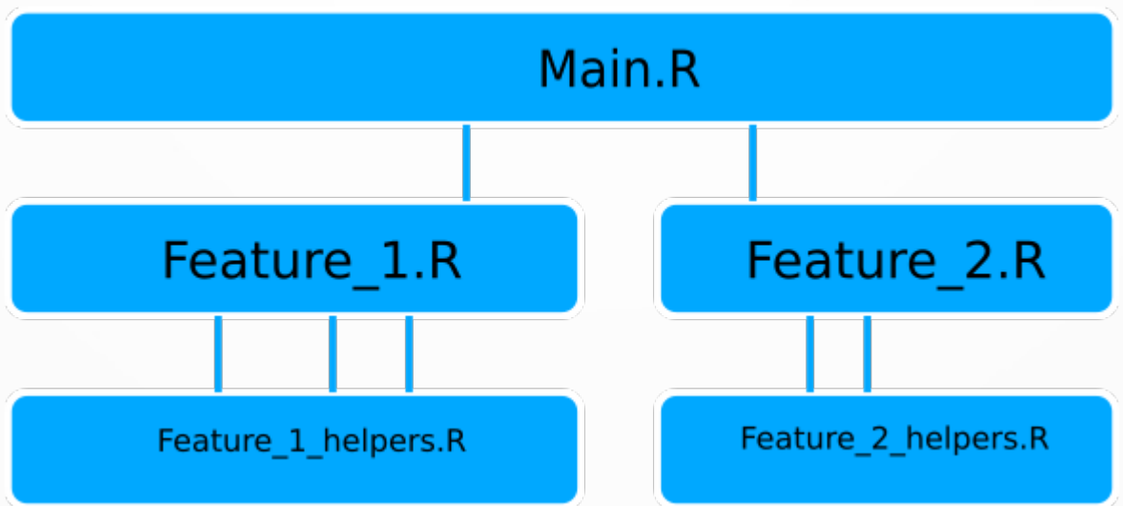
Result

Modules

Goals of modularization

Roles of modules

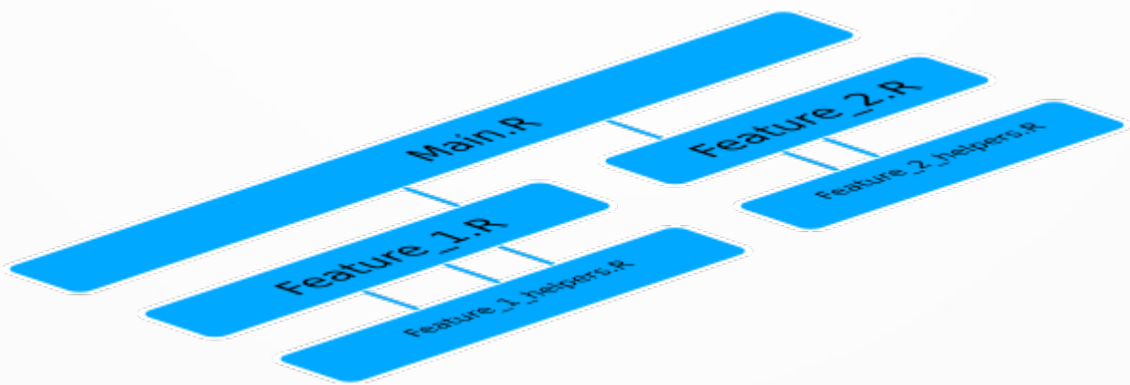
1) Structure the app



Goals of modularization

Roles of modules

1) Structure the app

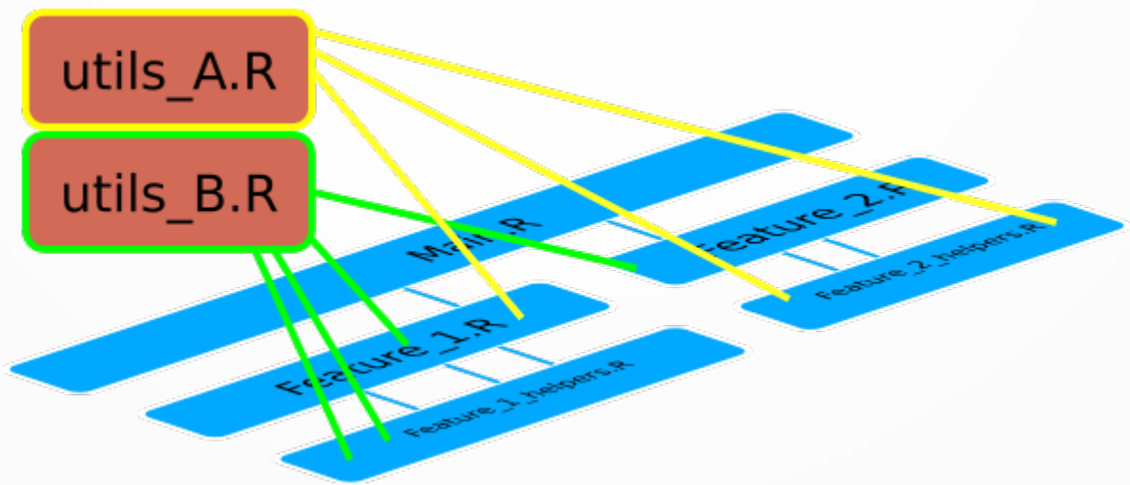


Goals of modularization

Roles of modules

1) Structure the app

2) Avoid repetition



Modules logic

Modules are structured similarly to an app: **UI + server**

But they can not stand alone and need to be **called** just as functions do.

Each part of a module is linked by a common **identifier** used to form its **namespace**.

Writing a module (UI part)

UI part of modules are **R function** with one needed argument: `inputId`.

(`inputId` might be renamed)

This argument is used in `NS()` to produce the namespace to all UI elements in this module.

```
my.UI ← function(
  inputId
) {
  tagList(
    textInput(
      NS(inputId, "text"),
      "Type text here"
    ),
    textOutput(
      NS(inputId, "out")
    )
  )
}
```

Writing a module (server part)

Server part of modules are **R function** with reserved **key arguments**:

- Input
- Output
- (session)
- … = other wanted arguments

```
my.server ← function(inputId, {
  moduleServer(id,
  function(input, output, …){

    # here: all the stuff you
    # need your module to manage.

  })
}
```

Calling a module

UI and server parts are each called in their own ways:

- **UI** with its function, **namespace** is filled with its `inputId` argument
- **server** with defined function, **namespace** is filled with the `inputId` argument

```
main.ui ← fluidPage(
  my.ui("module")
)

main.server ← function(
  input, output, session
){
  my.module("module")
}

shinyApp(main.ui,
main.server)
```

Nested modules

From inside a **module**, calling another **module** requires:

- The **UI**'s **inputId** to be **namespaced** (i.e. apply a `ns()` function as shown)
- The **server** is called as from a main server, and is automatically namespaced.

```
module1.ui ← function(id){
  ns ← NS(id) #define namespace

  module2.ui(ns("module"))
}

module1.server ← function(
  input, output, session
){
  callModule(
    module2.module,
    "module"
  )
}
```

App’s runtime

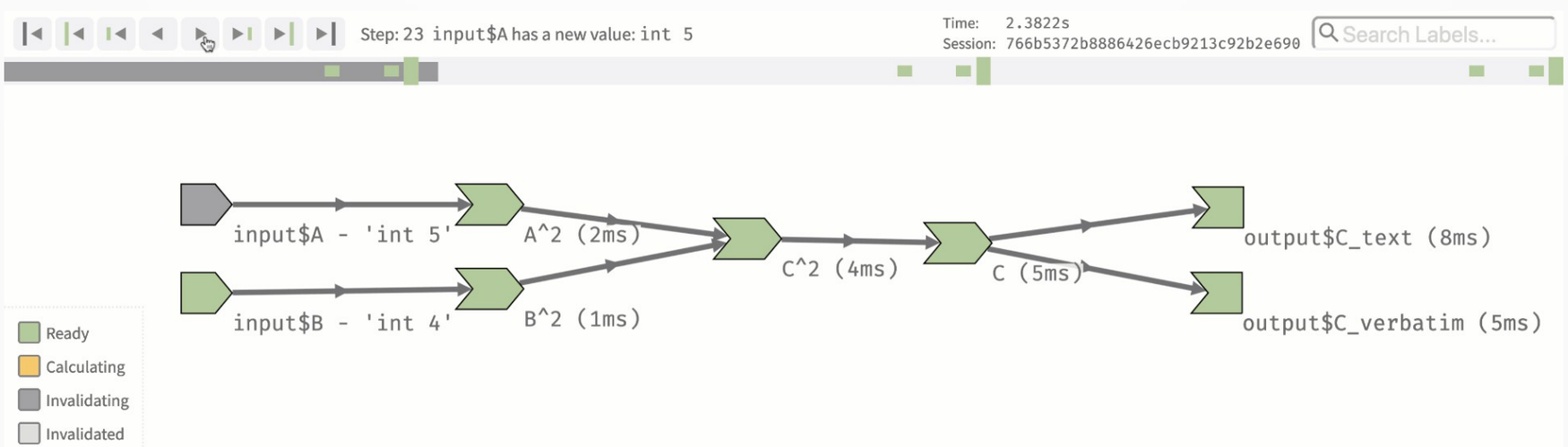
Time flow in a shiny app

- What is the exact flow of actions executed in a shiny app?
- How to control it?
- How to represent it?

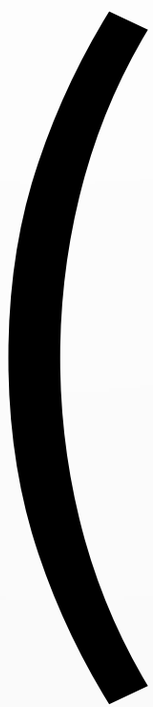
Time flow in a shiny app

- What is the exact flow of actions executed in a shiny app?
- How to control it?
- How to represent it?

Let's help ourselves with a dedicated tool:
{reactlog}



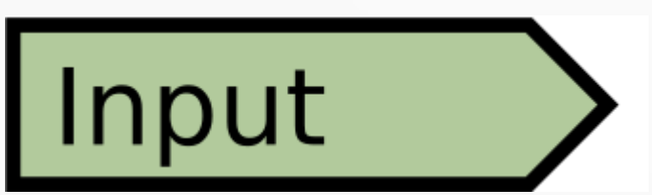
Notion of events



Notion of events

Any input change (i.e. clicking a button) triggers an **event**.

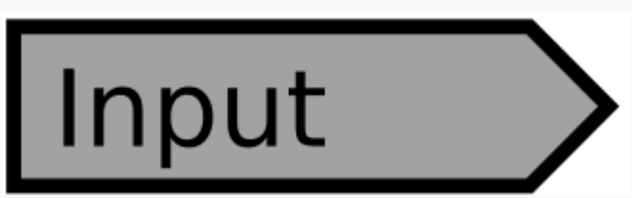
This causes a change in one of the slots of the `input` variable.



Notion of events

Any input change (i.e. clicking a button) triggers an **event**.

This causes a change in one of the slots of the `input` variable.



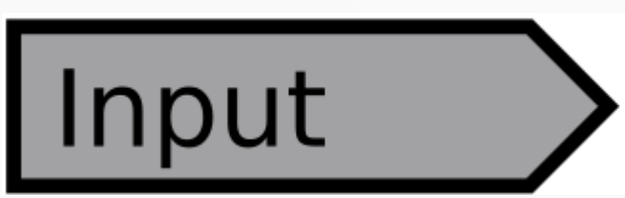
(Technical term: an event occurring
invalidates an element until being resolved)

Notion of events

Any input change (i.e. clicking a button) triggers an **event**.

This event is expected by **observers** (i.e. `observeEvent()`).

They directly rely on the values of the `input` variable.

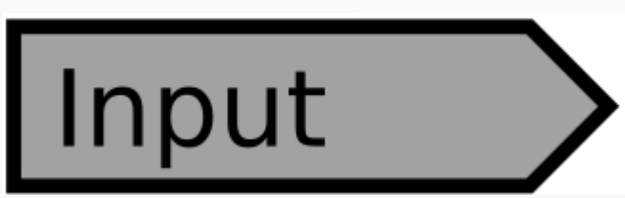


Notion of events

Any input change (i.e. clicking a button) triggers an **event**.

This event is expected by **observers** (i.e. `observeEvent()`).

They directly rely on the values of the `input` variable.



(It invalidates too.)

Notion of events

Any input change (i.e. clicking a button) triggers an **event**.

Observers generally focus on specific UI elements thanks to their **id** (cf. InputId).

This event is expected by **observers** (i.e. `observeEvent()`).

Notion of events

Steps for executing a shiny app

Setup step: each script is executed as usual.

- Variables are set
- Reactive structures are prepared
- UI is rendered
- Observers are run
- App idles

Execution step: user can interact with the app.

Upon interacting, events are triggered.

1) `input` variable changes as an input is changed.

2) The app **flushes** reactions due to `input` changes.

3) Once all executions have been flushed, the app idles back.

Dictionary of useful tools

UI / Layouts & CSS

- Stylish your app with CSS & layouts

```
includeCSS("<path-to-css-file>")
```

- Organize your GUI to provide the best ergonomy

UI / Layouts & CSS

`{shinydashboard}, {shinydashboardPlus}`

Specificities

- Bootstrap layout: containers principle
- Most used
- Robust but not flexible

https://stackoverflow.com/questions/41083412/modify-a-function-in-an-r-package-to-fit-for-purpose

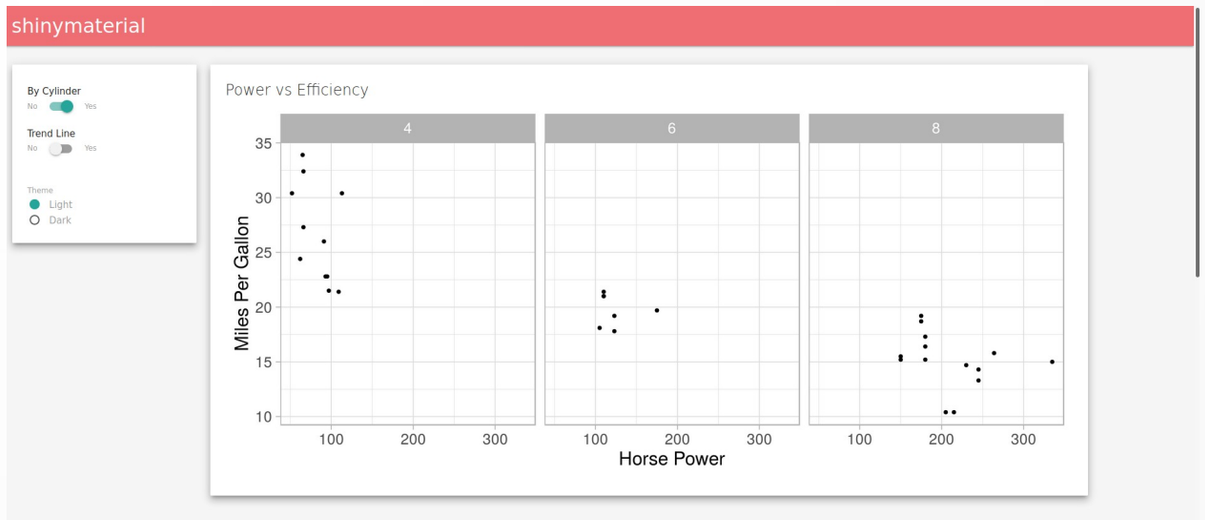
https://mdbootstrap.com/docs/jquery/layout/overview/

UI / Layouts & CSS

`{shinymaterial}`

Specificities

- Material Design layout
- Oriented for mobile applications
- Robust but not flexible



https://ericrayanderson.github.io/shinymaterial_showcase/

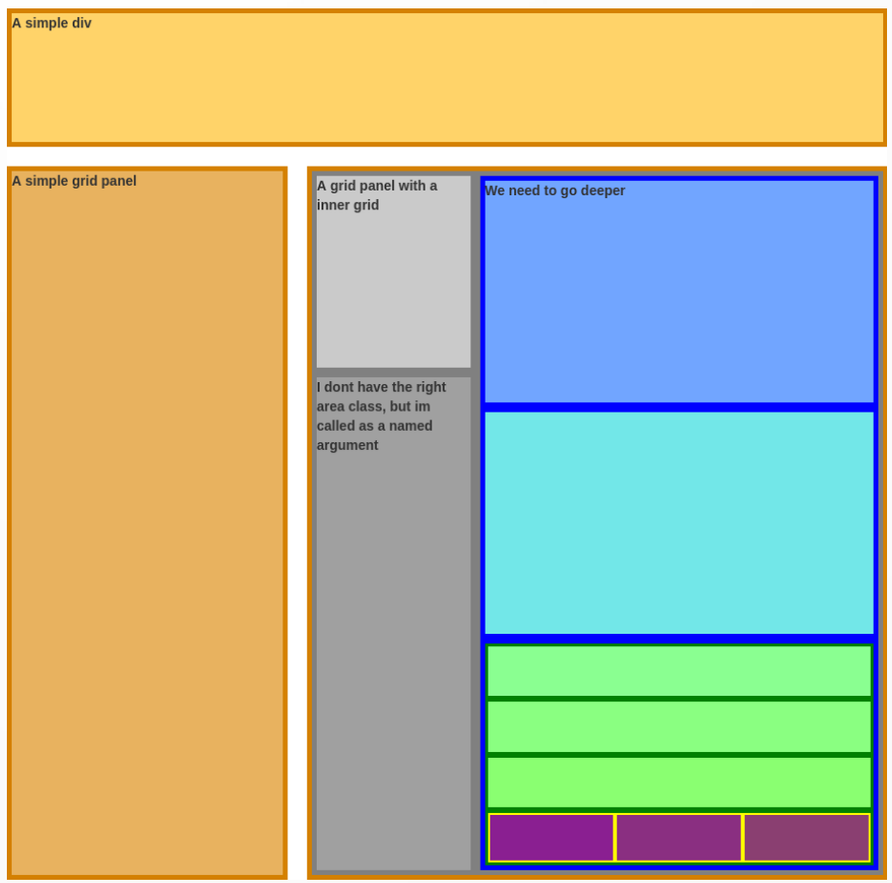
https://material.io/design/introduction#principles

UI / Layouts & CSS

`{shiny.grid}` (https://github.com/pedrocoutinhosilva/shiny.grid/)

Specificities

- Place elements in a grid, place grid in a grid.
- Very flexible and customizable
- Fewer support
- Nestable in dashboard



https://github.com/pedrocoutinhosilva/shiny.grid/blob/master/examples/example_2.R

UI / Elements

Structure	Principle	Value
*Input()	Input structures for * type.	An HTML input tag.
*Output()	Output structures for * type.	An HTML output tag
actionButton(), actionLink()	Button and link to trigger events.	An HTML button or link tag.
fluidRow(), column()	Division structures for bootstrap layout (most common).	An HTML with specific class.
tags	A list of HTML tags, useful for anything: https://shiny.rstudio.com/articles/tag-glossary.html	See link.

Server / Observers & reactive

Structure	Principle	Value
observe()	Execute its content if any of its dependencies* has changed.	None, only side effects.
reactive()	Returns its content when called, using dependencies'* current states.	A function.
observeEvent()	Same as observe(), but executes only on defined events.	None, only side effects.
eventReactive()	Same as reactive(), but executes only on defined events.	A function.
reactiveValues() reactiveVal()	Storage structures usable as dependencies* for observers.	Reactive lists.

*dependencies: invalidatable elements (i.e. `input` or a reactive()).

Server / Control structure

Structure	Principle	Value
validate() and need()	Stops execution if needs are not met. Return an error message without breaking the flow.	None
req()	Like validate() and need() but only for a single statement and without error message	None
update*	Allows to update values in the UI from the server.	None
isolate()	Allows to extract reactive content without a reactive context.	Content of a reactive element.

Useful packages

- `{shinyFeedback}` – inform the user about input validity.
- `{shinyFiles}` – alternative way to upload/download files.
- `{shinyjs}` – JavaScript integration, for more dynamic apps
- `{shinyBS}` – Bootstrap layout and design.
- `{shinyWidgets}` – a bunch of nice-looking widgets.
- `{shinyTree}` – used to render hierarchies, list … in which to select an element.

The end.