А.Ю. КУЗЬМЕНКОВ
А.Г. ВИНОГРАДОВА
И.В. ТРУШИН
А.А. АВРАМЕНКО
М.Ю. КУЗЬМЕНКОВ

НАУКА О ДАННЫХ
И ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ В МЕДИЦИНЕ

e-noto
Москва, 2026

Оглавление

Полная версия издания доступна на сайте

https://ds-book.ru

Глава 8
Вспомогательные инструменты для организации исследовательских проектов

8.4. Создание интерактивных приложений с Shiny

В предыдущем разделе пошагово была представлена разработка аналитического дашборда с использованием готового набора данных и пакета Quarto. Данный пакет обеспечивает некоторые интерактивные возможности, такие как масштабирование карты или переключение столбцов в графиках. Однако созданный таким образом дашборд не позволяет пользователям динамически фильтровать данные в реальном времени. Чтобы сформировать новую выборку данных (например данные за определенный период времени) придется фильтровать данные на уровне программного кода и заново генерировать дашборд. Для достижения полной интерактивности необходимо интегрировать специализированные элементы, такие как выпадающие списки или ползунки. Эти элементы позволят исследователям выбирать определенные группы пациентов, временные интервалы или другие параметры, повышая гибкость и глубину аналитических возможностей.

Решение данных задач возможно с помощью пакета *shiny* (https://shiny.posit.co). Он расширяет возможности R для создания полноценных интерактивных веб-приложений Shiny. Пакет *shiny* позволяет разработчикам использовать HTML, CSS и JavaScript в сочетании с возможностями языка R. Дополнительно доступен широкий спектр пакетов с компонентами, специально разработанными для интеграции с R и *shiny*, что упрощает разработку сложных интерактивных приложений.

С примерами приложений, созданных с использованием данного пакета, можно ознакомиться по ссылке https://shiny.posit.co/r/gallery

```
install.packages("shiny")
```

Далее рассмотрены основные концепции, необходимые для создания интерактивных приложений Shiny.

8.4.1. Основные понятия

Все приложения Shiny устроены схожим образом и состоят из двух частей: серверной и клиентской.

Подразумевается, что на серверной части производятся все необходимые вычисления и преобразования с наборами данных, подключения к внешним источникам данных, базам данных, работа с файлами. На клиентской части отображаются элементы с результатами обработки данных, элементы оформления интерфейса приложения и «виджеты» для взаимодействия с пользователем. Виджеты — это поля ввода, выпадающие списки, кнопки и т. д. С помощью виджетов можно передавать данные на серверную часть и влиять на работу приложения. С примерами виджетов,

которые используются в приложении Shiny, можно ознакомиться по ссылке https://shiny.posit.co/r/gallery/widgets/widget-gallery

Следует отметить, что разделение Shiny-приложения на серверную и клиентскую сторону во многом условно и обе части приложения могут быть описаны в одном файле R.

Самое простейшее приложение Shiny можно представить следующим образом.

```
library(shiny)

shinyApp(
  ui = list(),
  server = function(input, output, session) {
  }
)
```

Приложение *shinyApp* состоит из двух составляющих:

- *ui* — представляет собой список элементов и виджетов, которые формируют клиентскую часть;
- *server* — отвечает за работу серверной части. Серверная часть описывается как функция, с тремя аргументами:
  - *input* — для передачи событий от виджетов пользовательского ввода;
  - *output* — объект в который передаются результаты работы серверной части, чтобы отобразить на клиентской части;
  - *session* — объект для хранения данных о текущем сеансе работы пользователя.

Связь между серверной и клиентской частями в Shiny приложениях реализуется с помощью механизма реактивности. Этот механизм позволяет автоматически передавать любые изменения, вносимые в виджеты на клиентской стороне, на серверную сторону для дальнейшей обработки.

Для примера далее рассмотрены два способа построения приложения с использованием Shiny: с использованием Quarto и как самостоятельное приложение.

8.4.2. Использование Shiny с Quarto

Приложения Shiny можно легко встраивать в документы Quarto, устраняя необходимость создания отдельных объектов *shinyApp*, *ui* и *server*. Однако разделение на серверную и клиентскую части сохраняется, что позволяет получать события из объекта *input* и передавать результаты через объект *output*.

Далее рассмотрен пример создания такого приложения, на основе созданного ранее дашборда.

Чтобы создаваемый документ Quarto стал готов к использованию Shiny, необходимо:

- добавить в преамбулу дополнительный атрибут *server*, который указывает, что в качестве сервера приложений будет использоваться Shiny;
- убрать описание output-file для выходного файла.

Было:

```
---
title: “Новый красивый дашборд”
author: “Data Science Course”
format:
  dashboard:
    theme:
    - cosmo
    logo: logo.png
    nav-buttons:
    - icon: github
      href: https://github.com/data-science-course/quarto-dashboard
---
```

Стало:

```
---
title: “Наш красивый дашборд”
author: “Ivan Trushin”
format:
  dashboard:
    theme:
    - cosmo
    logo: logo.png
    nav-buttons:
    - icon: github
      href: https://github.com/rymbln/my-new-dashboard
server: shiny
---
```

В настройках проекта Quarto в файле *_quarto.yml* нужно также убрать строку *output-dir: docs*.

Было:

```
project:
  title: “quarto_dashboard”
  output-dir: docs
```

Стало:

```
project:
  title: “quarto_dashboard”
```

После этого необходимо описать серверную часть приложения-дашборда. Для этого в рамках дашборда нужно объявить контекст сервера. Такие контексты определяются с помощью дополнительного атрибута для кода в чанках. Для дашборда рекомендуется объявить как минимум два контекста:

- *setup* — для общего глобального кода, который будет выполняться при старте приложения: загружать пакеты, создавать общие объекты, читать файлы и т.д.

- *server* — для кода, который будет определять логику серверной части приложения.

Весь остальной код по умолчанию будет считаться кодом клиентской части и отображаться на странице. Более подробная информация о контекстах выполнения кода в документах Quarto доступна по ссылке https://quarto.org/docs/ interactive/ shiny/execution.html.

Например, в представленном коде чтение файла будет выполняться в контексте *setup*.

```
```{r}
#| label: Загрузка данных
#| include: false
#| context: setup
patients <- read.csv2("patients.csv", dec = ".")
patients$DATEBIRTH <- ymd(patients$DATEBIRTH)
patients$DATESTRAIN <- ymd(patients$DATESTRAIN)
patients$DATEFILL <- ymd(patients$DATEFILL)
```
```

Рекомендуется объявить загрузку необходимых пакетов в блоке *setup* контекста Quarto. В приведенном примере подключаются пакеты shiny и shinydashboard (https:// rstudio.github.io/shinydashboard) для расширения доступных виджетов.

```
```{r}
#| label: Загрузка пакетов
#| include: false
#| context: setup
library(tidyverse)
# ... #
library(shinydashboard)
library(shiny)
```
```

Настройки таблиц *reactable* для примера будут определены в контексте сервера.

```
```{r}
#| label: reacttable-options
#| include: false
#| context: server
options(reactable.language = reactableLang(
  pageSizeOptions   = "показано {rows} значений",
  pageInfo          = "с {rowStart} по {rowEnd} из {rows} строк",
  pagePrevious      = "назад",
  pageNext          = "вперед",
  searchPlaceholder = "Поиск...",
  noData            = "Значения не найдены"
))
```
```

Теперь, когда определен контекст сервера, можно добавить виджеты на страницу. Для этого необходимо создать боковую панель (сайдбар) для размещения виджетов и добавить выпадающий список (*selectInput*) для выбора группы пациентов со следующими параметрами:

- *inputId* — идентификатор виджета для работы с ним в серверной части;
- *label* — подпись элемента на странице;
- *choices* — доступные для выбора значения в списке.

Этот виджет будет доступен в объекте input внутри приложения Shiny.

```
# {.sidebar}
Данное исследование рассматривает  `{r} nrow(patients)` пациентов из `{r}
length(unique(patients$CITYNAME))` городов России и Беларуси.

```{r}
#| label: Контролы фильтров

selectInput(inputId = 'selPatgroup',
            label = 'Группа пациентов',
            choices = c(Все = '.', sort(unique(patients$PAT_GROUP)))
            )
```
```

Следует обратить внимание, что в значения для выбора (*choices* =) кроме значений групп пациентов из набора данных добавлено значение *Все = '.'*, которое будет выбрано по умолчанию. Результат представлен на рис. 8.161.

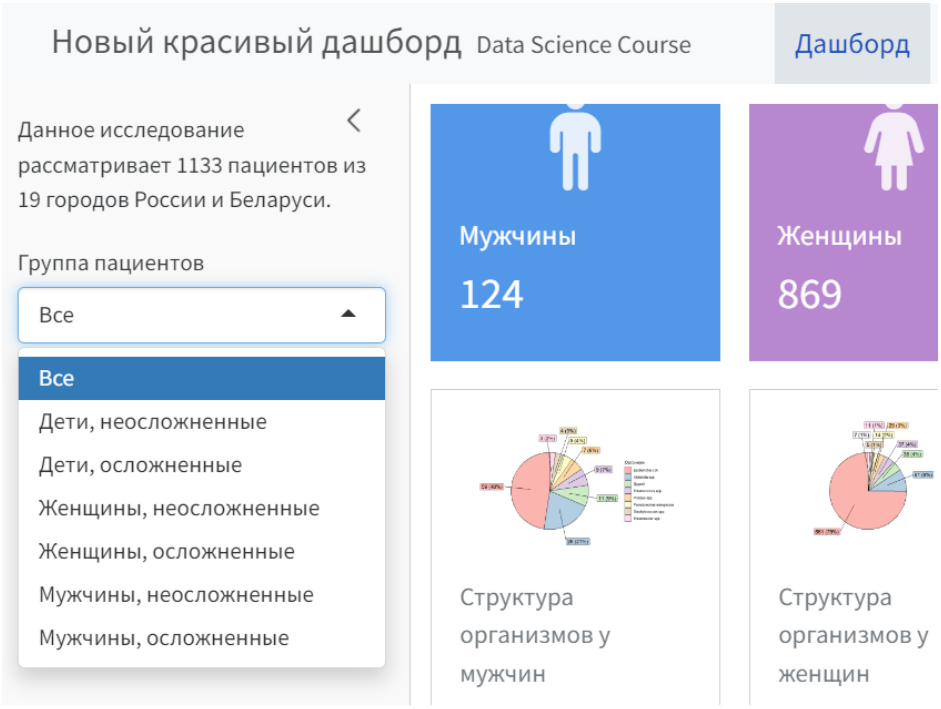

Рис. 8.161. Дашборд с добавленным фильтром для выбора группы пациентов.

Теперь необходимо добавить реакцию на изменение выбранного значения в виджете. В зависимости от выбранного значения в выпадающем списке *selPatgroup* (название идентификатора виджета, которое было задано ранее в *inputId = 'selPatgroup'*) следует отфильтровать данные из исходного набора данных *patients* и создадать новый реактивный набор данных *data*. Этот новый набор данных будет использоваться для дальнейших расчетов в дашборде. Поскольку вычисления будут выполняться на стороне сервера, необходимо установить соответствующий контекст для этого кода.

```
```{r}
#| label: отбор данных по фильтрам
#| context: server
data <- reactive({
  d <- patients
  if (input$selPatgroup != "." ) {
    d <- d %>% filter(PAT_GROUP == input$selPatgroup)
  }
  d
})
```
```

Для создания реактивной связи используется функция *reactive()*, которая выполняется автоматически при каждом изменении значений, определенных в *input*. Внутри этой функции описываются действия, которые нужно произвести. В ней происходит обращение к виджету со списком групп пациентов через *input$selPatgroup*. Если его значение отлично от значения по умолчанию, которое определяет все группы пациентов, то написанный программный код фильтрует исходный набор данных и записывает результат в переменную *d*. В конце, как и в любой функции, необходимо вернуть результат, чтобы он присвоился объекту *data*.

Теперь в приложении появился реактивный объект *data*, который содержит отфильтрованные данные и который будет использоваться для расчетов. Можно подсчитать количество отфильтрованных строк и отобразить его на дашборде. Для этого в контексте сервера необходимо создать виджет вывода, который будет использовать данный реактивный элемент и присвоить его объекту *output*. Существует огромное количество различных виджетов для вывода, все они определяются своими функциями. Можно добавлять новые виджеты вывода с помощью различных пакетов. На данном этапе будет выведено простое текстовое значение с количеством образцов и количеством центров в отфильтрованном наборе данных. Для этого используется функция *renderText()*.

```
```{r}
#| label: создание выходного результата
#| context: server
output$count_text <- renderText({
 paste("Выбрано", nrow(data()), "образцов", sep = " ")
})

output$data_count <- renderText(nrow(data()))
output$center_count <- renderText(length(unique(data()$CENTER)))
```
```

Следует обратить внимание, что для получения результата из реактивного объекта *data* необходимо написать название со скобками ***data()***, потому что на самом деле это не переменная, а функция, значение которой зависит от значения виджетов. Таким образом, при изменении значения виджета *input$selPatgroup* меняется реактивный объект *data()*, и далее по цепочке при каждом изменении *data* меняется и виджет вывода значений *output$count_text*, отображающийся на интерфейсе. Общая логика взаимодействия реактивных элементов представлена на рис. 8.162.

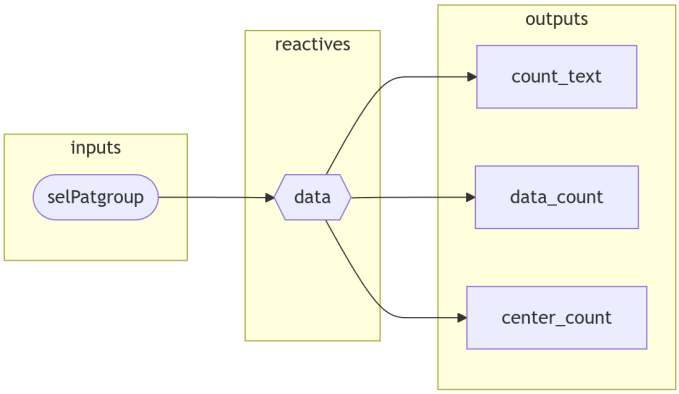

Рис. 8.162. Общая логика взаимодействия реактивных элементов.

Теперь, когда в объекте *output* есть реактивные виджеты, их можно разместить на дашборде. Виджеты можно добавлять в чанки без указания контекста, поскольку по умолчанию используется контекст *ui*, отвечающий за клиентскую часть. Для вывода различных типов виджетов, определенных в объекте output, используются разные функции. Для вывода текста используется функция *textOutput()*, в которую передается название объекта. Тестовую строку с количеством строк в таблице можно отобразить прямо под выпадающим списком.

```
```{r}
#| label: Контролы фильтров

selectInput(inputId = 'selPatgroup',
            label = 'Группа пациентов',
            choices = c(Все = '.', sort(unique(patients$PAT_GROUP)))
            )
# вывод количества отфильтрованных строк
textOutput('count_text')
```
```

Количество пациентов и центров будет выводиться непосредственно на дашборде с использованием разметки Markdown.

```
|              |                                        |
|--------------|----------------------------------------|
| **Центров **  | `{r} textOutput('center_count')`      |
| **Пациентов** | `{r} textOutput('data_count')`        |
| **Дата**      | `{r} Sys.Date()`                      |
```

Полный код сайдбара на текущий момент выглядит следующим образом. Взаимное расположение блоков не так важно, реактивные компоненты в серверной части могут определяться и после их использования в разметке.

```
# {.sidebar}

Данное исследование рассматривает  `{r} nrow(patients)` пациентов из `{r}
length(unique(patients$CITYNAME))` городов России и Беларуси.

```{r}
#| label: Контролы фильтров

selectInput(inputId = 'selPatgroup',
            label = 'Группа пациентов',
            choices = c(Все = '.', sort(unique(patients$PAT_GROUP)))
            )
# вывод количества отфильтрованных строк
textOutput('count_text')
```

**Центров **	`{r} textOutput('center_count')`
Пациентов	`{r} textOutput('data_count')`
Дата	`{r} Sys.Date()`

```{r}
#| label: отбор данных по фильтрам
#| context: server
data <- reactive({
  d <- patients
  if (input$selPatgroup != "." ) {
    d <- d %>% filter(PAT_GROUP == input$selPatgroup)
  }
  d
})
```

```{r}
#| label: создание выходного результата
#| context: server
output$count_text <- renderText({
 paste("Выбрано",nrow(data()), "образцов", sep = " ")
})

output$data_count <- renderText(nrow(data()))
output$center_count <- renderText(length(unique(data()$CENTER)))
```

На дашборде результат выглядит следующим образом (рис. 8.163).

Следует обратить внимание, что если выбрать группу пациентов «Дети, неосложненные», то числа под выпадающим списком изменятся, а над ним — нет. Таким образом, можно наглядно увидеть разницу с обычным набором данных *`patients {r} nrow(patients)`* и реактивным *`data(): {r} textOutput('center_count')`*, который фильтрует данные в зависимости от выбранной группы пациентов (рис. 8.164).
```

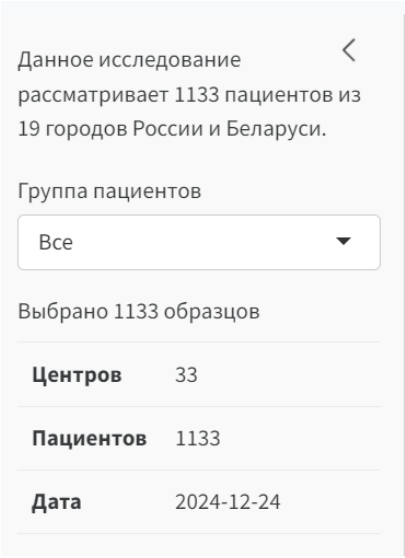

Рис. 8.163. Визуальное отображение боковой панели с результатами работы реактивных элементов и виджетов вывода.

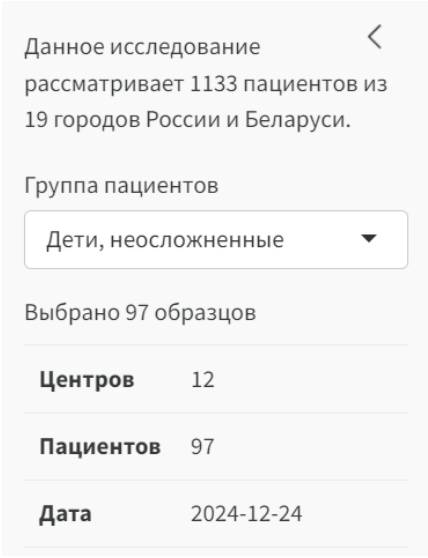

Рис. 8.164. Визуальное отображение боковой панели с результатами работы реактивных элементов и виджетов вывода после выбора группы пациентов «Дети, неосложненные».

Далее следует добавим дополнительные фильтры в сайдбар. Например, список с возможностью выбора нескольких элементов selectInput() для отбора городов.

```
selectInput('selCity', 'Город',
            choices = c(Все = '.', sort(unique(patients$CITYNAME))),
            selected = c("."),
            multiple = TRUE)
```

Также следует добавить виджет для определения временного периода, когда был получен образец в рамках исследования — *dateRangeInput()*.

```
minDate <- min(patients$DATESTRAIN, na.rm = TRUE)
maxDate <- max(patients$DATESTRAIN, na.rm = TRUE)
dateRangeInput('selDateRange', 'Дата взятия образца',
  start = minDate, end = maxDate, min = minDate, max = maxDate,
  format = "yyyy-mm-dd", startview = "month", weekstart = 1,
  language = "ru", separator = " - ", width = NULL, autoclose = TRUE
)
```

А также виджет для отбора пациентов по возрасту — *sliderInput()*.

```
minAge <- min(patients$AGE)
maxAge <- max(patients$AGE)
sliderInput('selAge', 'Возраст',
  min = minAge, max = maxAge, value = c(minAge, maxAge),
  step = 1, dragRange = TRUE
)
```

В результате сайдбар примет следующий вид (рис. 8.165).

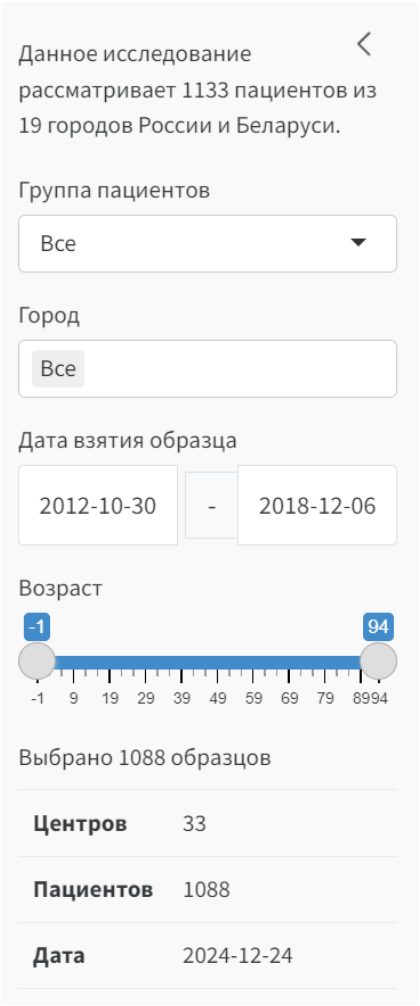

Рис. 8.165. Визуальное отображение боковой панели с дополнительными виджетами для фильтрации данных.

Теперь можно сформировать реактивный объект *data()*, который учитывает значения всех четырех входных виджетов для фильтрации данных.

```
data <- reactive({
  d <- patients
  if (input$selPatgroup != "." ) {
    d <- d %>% filter(PAT_GROUP == input$selPatgroup)
  }
  if ( !("." %in% input$selCity ) ) {
    d <- d %>% filter(CITYNAME %in% input$selCity)
  }
  if (length(input$selDateRange) == 2) {
  d <- d %>% filter(DATESTRAIN >= input$selDateRange[1] & DATESTRAIN <= in-
```

```
put$selDateRange[2])
  }
  if (length(input$selAge) == 2) {
  d <- d %>% filter(AGE >= input$selAge[1] & AGE <= input$selAge[2])
  }

  d
})
```

Таким образом, реактивный набор данных *data()* уже зависит от четырех реактивных *inputs*. Эти четыре фильтра будут определять результат, выводимый в таблице сайдбара (рис. 8.166).

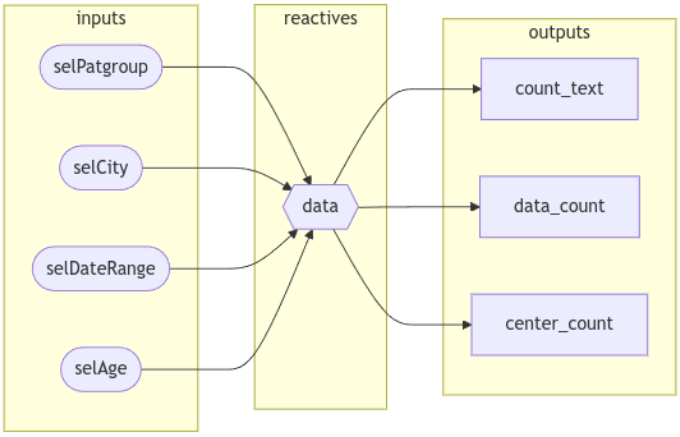

Рис. 8.166. Общая логика взаимодействия реактивных элементов с учетом четырех виджетов для фильтрации данных.

Реактивные объекты, создаваемые в контексте сервера могут зависеть не только от виджетов *input*, но и от других реактивных объектов. Например, можно определить новые реактивные значения, которые будут подсчитывать количество мужчин, женщин и детей в отфильтрованном наборе данных.

```
# Количество мужчин.
mens_count <- reactive({
  data() %>% filter(grepl("Мужчины",PAT_GROUP)) %>% nrow()
})
# Количество женщин.
womens_count <- reactive({
  data() %>% filter(grepl("Женщины",PAT_GROUP)) %>% nrow()
})
# Количество детей.
children_count <- reactive({
  data() %>% filter(grepl("Дети",PAT_GROUP)) %>% nrow()
})
```

В этом случае значения *mens_count*, *womens_count* и *children_count* зависят уже от реактивного объекта *data()*, а не напрямую зависят input. Следует обратить внимание, что с реактивными объектами можно работать точно также, как и с обычными, несмотря на скобки.

Для каждого из этих значений можно создать отдельные информационные элементы интерфейса — *valueBox* в объекте *output*, которые заменят прежние статические панели с числами. В отличие от вывода текста, здесь будет использоваться функция *renderValueBox()*.

```
# Количество мужчин.
output$graph_mens_count <- renderValueBox({
    valueBox(mens_count(),"Мужчины", color = "blue")
  })
# Количество женщин.
output$graph_womens_count <- renderValueBox({
    valueBox(womens_count(),"Женщины", color = "purple")
  })
# Количество детей.
output$graph_children_count <- renderValueBox({
    valueBox(children_count(),"Дети", color = "orange")
  })
```

Теперь схема реактивного взаимодействия выглядит следующим образом (рис. 8.167).

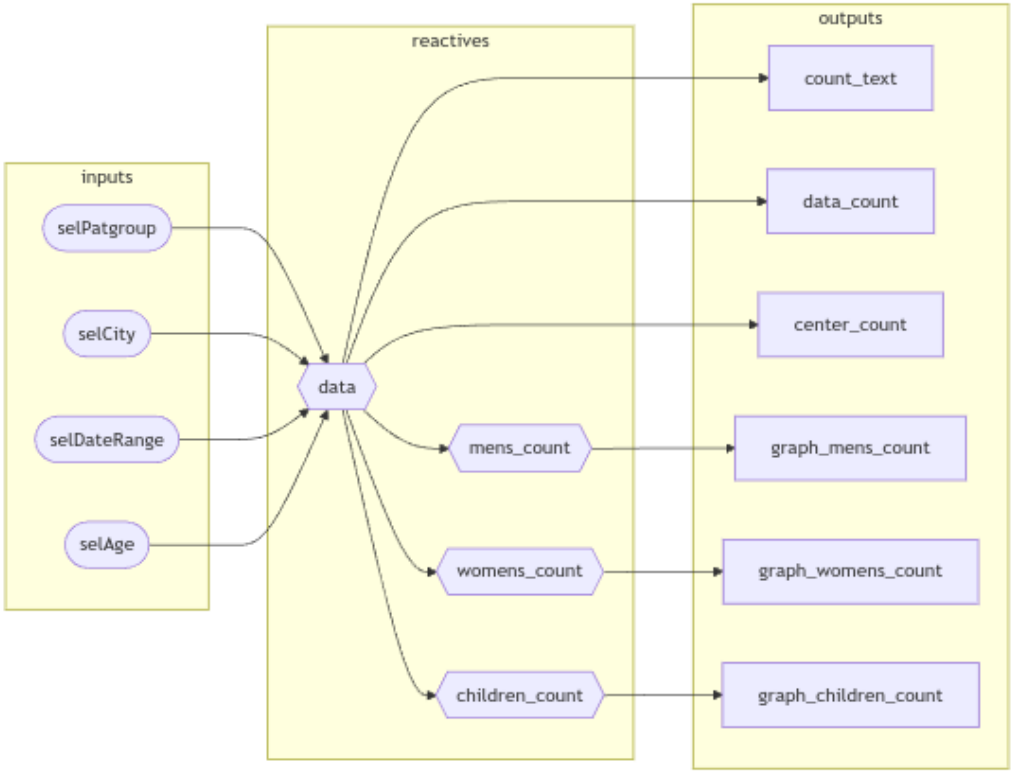

Рис. 8.167. Общая логика взаимодействия реактивных элементов с учетом четырех виджетов для фильтрации данных и добавлением *valueBox*.

Осталось заменить в дашборде **статические панели**

```
```{r}
#| label: Кол-во мужчин
#| content: valuebox
#| title: "Мужчины"
list(
  icon = "person-standing",
  color = "primary",
  value = mens_count
)
```
```

на динамические.

```
```{r}
#| label: Кол-во мужчин
valueBoxOutput("graph_mens_count")
```
```

Создавать отдельные реактивные объекты только для того, чтобы вывести их на странице, как было сделано в предыдущем случае, вовсе не обязательно. Обычно так поступают, если предполагается многократное использование реактивных данных другими реактивными объектами или несколькими виджетами. Например, в рамках дашборда создается объект *citypat*, который затем отображается в трех таблицах на отдельных вкладках.

В случае же, если данные используются только для отображения одного единственного виджета, можно их формировать внутри функции *render*. Например, в случае отображения структуры организмов можно поступить именно таким образом. Результатом рендеринга будет график *ggplot2*, поэтому следует использовать функцию *renderPlot()* для создания виджета и функцией *plotOutput()* для отображения на дашборде.

```
```{r}
#| label: Подсчет количества организмов у мужчин
#| context: server

output$plot_org_male <- renderPlot({
  # Получение данных
  org_male <- data() %>%
  filter(grepl("Мужчины",PAT_GROUP)) %>%
  group_by(STRAIN) %>%
  summarise(Count = n()) %>%
  ungroup() %>%
  mutate(Percent = round(100 * Count / sum(Count))) %>%
  arrange(desc(Percent)) %>%
   mutate(csum = rev(cumsum(rev(Count))),
          pos = Count/2 + lead(csum, 1),
          pos = if_else(is.na(pos), Count/2, pos))
```
```

```
  # Сохдание графика
  ggplot(org_male, aes(x = "" , y = Count, fill = fct_inorder(STRAIN))) +
  geom_col(width = 1, color = 1) +
  coord_polar(theta = "y") +
  scale_fill_brewer(palette = "Pastel1") +
  geom_label_repel(data = org_male,
                   aes(y = pos, label = paste0(Count, " (", Percent, "%)")),
                   size = 4.5, nudge_x = 1, show.legend = FALSE) +
  guides(fill = guide_legend(title = "Организм")) +
  theme_void()
})
```

```
```{r}
#| label: Вывод количества организмов у мужчин
#| fig-cap: "Структура организмов у мужчин"

plotOutput('plot_org_male')
```
```

Полученный результат аналогичен обычному дашборду Quarto (рис. 8.168).

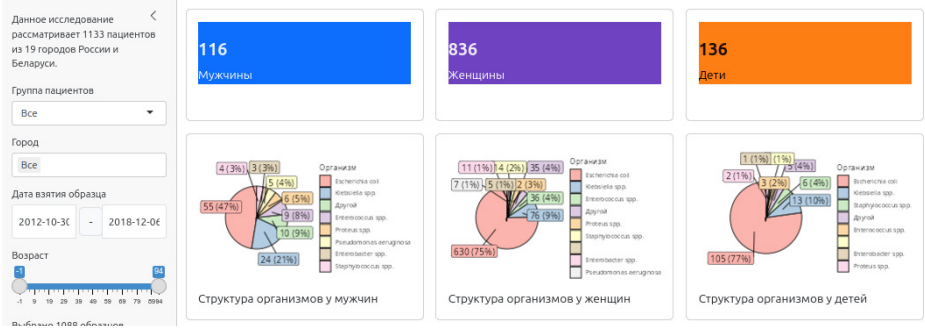

Рис. 8.168. Визуальное отображение дашборда.

Однако если попытаться выбрать какую-либо конкретную группу пациентов, то графики могут «сломаться», потому что для их построения будут отсутствовать необходимые данные (рис. 8.169).

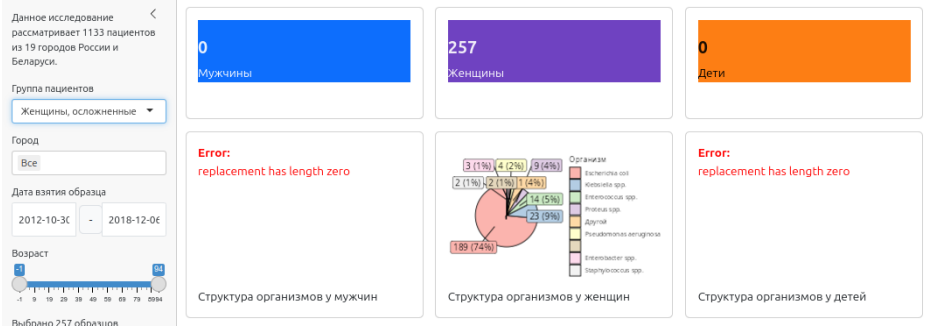

Рис. 8.169. Визуальное отображение дашборда с ошибками отображения графика после выбора группы пациентов.

Чтобы избежать такой ситуации необходимо добавить дополнительный код для проверки корректности данных.

```
# Получение данных
# ...
# Проверка результата
validate(
    need(nrow(org_male) > 0, “Данные отсутствуют”)
)
# Создание графика
# ...
```

В данном примере после создания набора данных для графика проверяется, содержит ли он строки (код *validate(need(nrow(org_male) > 0, "Данные отсутствуют"))*), и если набор данных пустой — выводится сообщение-заглушка. Теперь графики выглядят следующим образом (рис. 8.170).

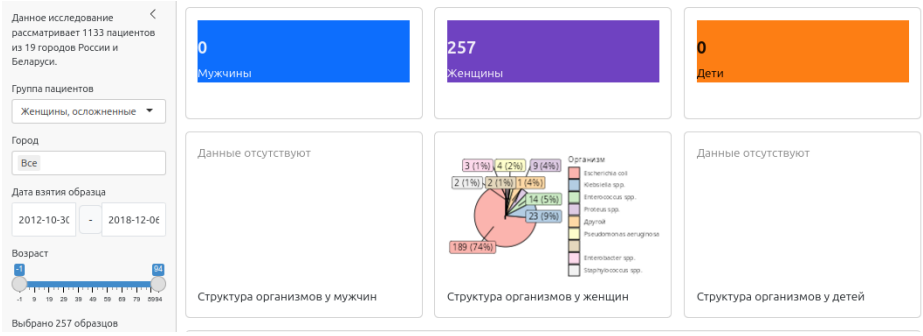

Рис. 8.170. Визуальное отображение дашборда с сообщением-заглушкой в случае отсутствия данных для графиков.

В рамках приложения Shiny можно заменить различные элементы дашборда на реактивные виджеты (табл. 8.1).

Таблица 8.1. Функции для создания и отображения различных элементов в приложении Shiny

Элемент	Создание	Отображение
График *plotly*	*renderPlotly()*	*plotlyOutput()*
Карта *leaflet*	*renderLeaflet()*	*leafletOutput()*
Таблица *reactable*	*renderReactable()*	*reactableOutput()*
Таблица *kable*	*function()*	*tableOutput()*
Таблица *gt*	*render_gt()*	*gt_output()*
Таблица *flextable*	*renderUI()*	*tableOutput()*

Следует обратить внимание на особенности создания виджетов для различных видов статических таблиц. Для создания виджета из таблиц *flextable* используется не совсем привычная функция *renderUI()*, а сам результирующий объект таблицы дополнительно преобразуется с помощью функции *htmltools_value()*. Это необходимо, чтобы преобразовать таблицу в привычный для дашборда HTML-формат.

```
output$citypat_ft <- renderUI({
  citypat() %>%
  flextable() %>%
  set_caption("Распределение пациентов по городам") %>%
  theme_zebra() %>%
  htmltools_value()
})
```

Виджеты из таблиц *kable* создаются также специфичным образом.

```
output$citypat_kb <- function(){
  citypat() %>%
  kbl(caption = "Распределение пациентов по городам") %>%
  kable_styling(bootstrap_options = c("striped"))
}
```

Поэтому в контексте Shiny-приложений наиболее «естественный» способ отображения таблиц — с помощью пакета *gt*.

```
output$citypat_gt <- render_gt({
  citypat() %>%
  gt() %>%
  tab_header(title = "Распределение пациентов по городам") %>%
  opt_row_striping()
})
```

В результате, после замены всех элементов на виджеты, получится дашборд, в котором все графики и таблицы автоматически изменяются в зависимости от выбранных фильтров.

Побочным эффектом после добавления Shiny-приложения в дашборд стала невозможность сохранения дашборда в качестве статического сайта и размещения его на Github Pages. Такой интерактивный дашборд можно опубликовать в качестве приложения на других сервисах с помощью кнопки **Publish Document** в RStudio, или разместить на собственном сервере (рис. 8.171).

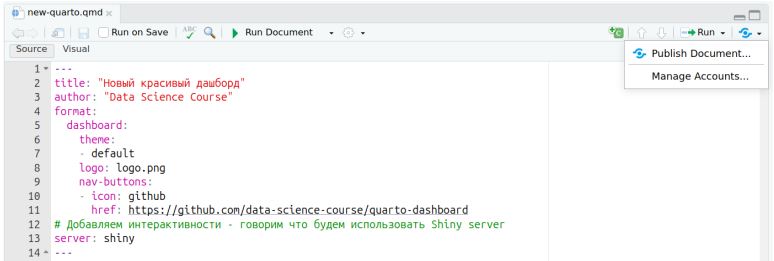

Рис. 8.171. Меню для публикации Shiny-приложения в интерфейсе Rstudio.

По умолчанию предлагаются следующие сервисы (рис. 8.172).

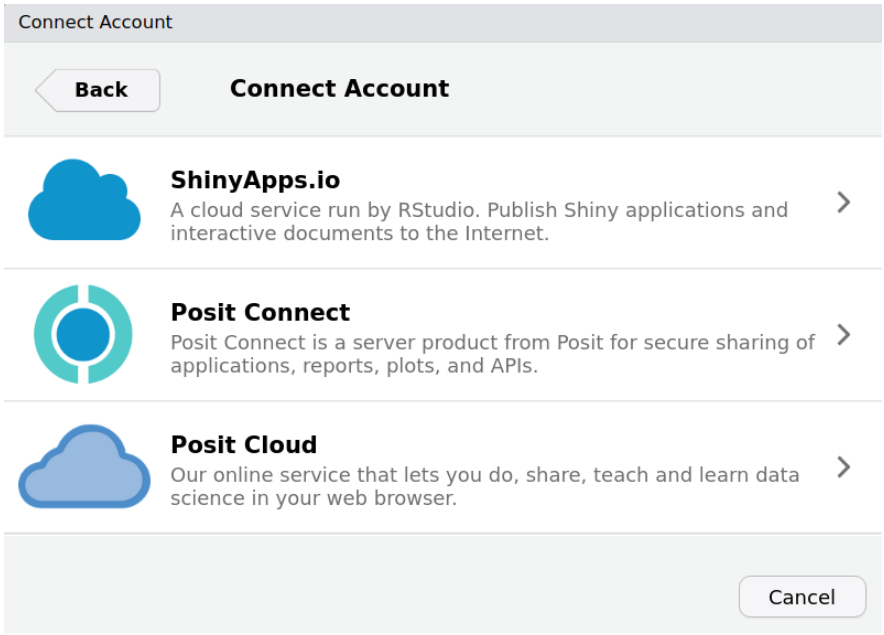

Рис. 8.172. Варианты размещения Shiny-приложения в интерфейсе Rstudio.

Все из них требуют предварительной регистрации учетной записи пользователя и предусматривают как платные тарифы, так и бесплатные тарифы с ограничениями на количество публикуемых приложений и время их работы. Сам процесс публикации происходит практически автоматически в несколько шагов с помощью мастера и обычно не вызывает затруднений. Однако, в реальной практике наиболее частым сценарием является размещение созданного приложения на собственном сервере. Это обеспечивает больший контроль и возможности дополнительных настроек над проектом и не требует дополнительных затрат (кроме непосредственно аренды сервера при его отсутствии). Подробнее данный подход представлен в разделе 8.5.

8.4.3. Создание Shiny приложения

Более продвинутым подходом является создание самостоятельного Shiny-приложения, которое можно использовать как инструмент самостоятельной аналитики. Оно позволит загружать файлы, визуализировать данные и генерировать подробные отчеты.

Для создания подобного приложения необходимо создать новый проект Shiny в RStudio и сразу отметить галочками создание git-репозитория, а также добавления *.renv* в проект (рис. 8.173).

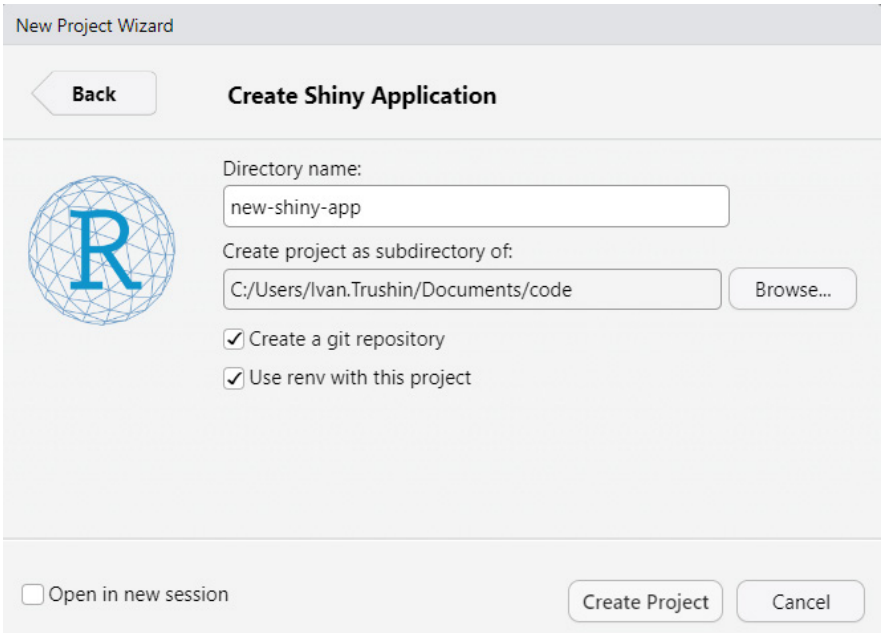

Рис. 8.173. Инициализация Shiny-приложения в интерфейсе Rstudio.

Основная логика приложения описывается в файле *app.R*, который содержит стартовый шаблон приложения с примером разметки интерфейса, виджетов и построения графика.

```
library(shiny)
# Define UI for application that draws a histogram
ui <- fluidPage(
    # Application title
    titlePanel("Old Faithful Geyser Data"),
    # Sidebar with a slider input for number of bins
    sidebarLayout(
        sidebarPanel(
            sliderInput("bins",
                        "Number of bins:",
                        min = 1,
                        max = 50,
                        value = 30)
        ),
        # Show a plot of the generated distribution
        mainPanel(
           plotOutput("distPlot")
        )
    )
)
# Define server logic required to draw a histogram
server <- function(input, output, session) {
    output$distPlot <- renderPlot({
        # generate bins based on input$bins from ui.R
```

```
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)
        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', border = 'white',
             xlab = 'Waiting time to next eruption (in mins)',
             main = 'Histogram of waiting times')
    })
}
# Run the application
shinyApp(ui = ui, server = server)
```

Данное приложение можно запустить с помощью кнопки *Run App* и оно откроется в отдельном окне (рис. 8.174).

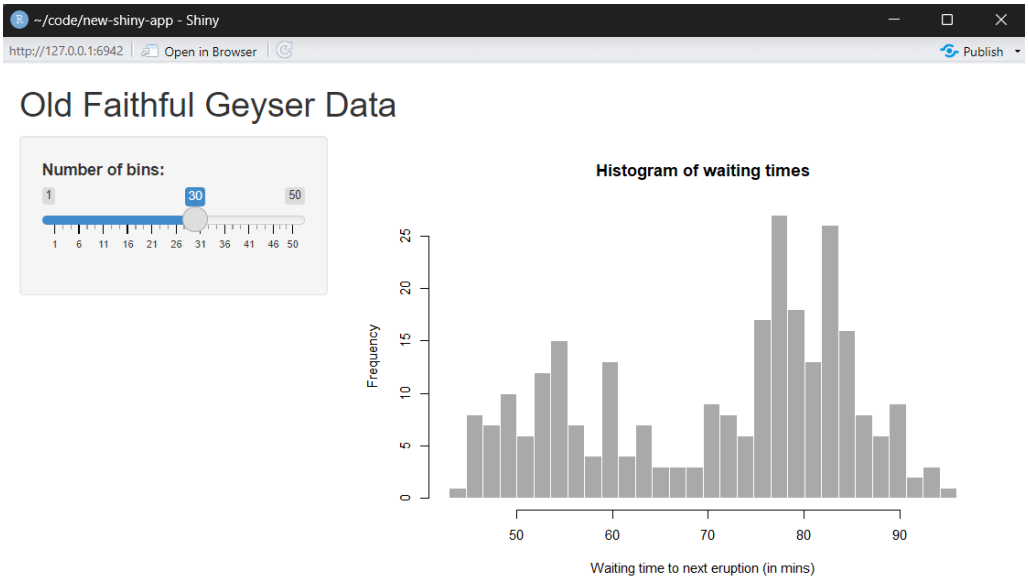

Рис. 8.174. Запущенный пример Shiny-приложения в интерфейсе Rstudio.

Из кода примера видно, что приложение Shiny состоит из двух частей:

- *ui* — в котором описывается интерфейс;
- *server* — в котором выполняется логика приложения и обрабатываются действия пользователя.

Финальным шагом является запуск приложения с помощью функции *shinyApp()* и передачи туда созданных объектов *ui* и *server*.

Интерфейс приложения создается путем «вкладывания» друг в друга различных компонентов. В созданном примере сначала создается страница приложения *fluidPage*, для нее определяется заголовок *titlePanel*, и задается разметка — *sidebarLayout*. В левой части будет расположена управляющая панель (*sidebarPanel*), в правой — основная рабочая область (*mainPanel*). Структура такого интерфейса представлена на рис. 8.175.

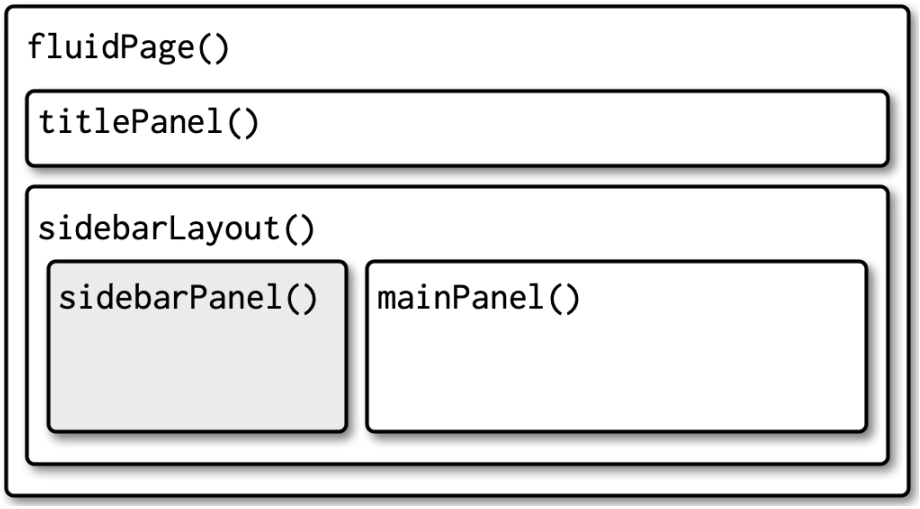

Рис. 8.175. Структура интерфейса Shiny-приложения из стартового шаблона.

Кроме вышеперечисленных способов компоновки, можно использовать размещение сеткой из строк и столбцов, как это ранее использовалось при построении дашбордов (рис. 8.176).

```
fluidPage(
  fluidRow(
    column(4,
    ...
    ),
    column(8,
      ...
    )
  ),
  fluidRow(
    column(6,
      ...
    ),
    column(6,
      ...
    )
  )
)
```

В случае, когда одной страницы приложения недостаточно, Shiny предоставляет компоненты для размещения виджетов на нескольких вкладках. Функция tabsetPanel() позволяет создавать на странице набор вкладок, каждая из которых может содержать свой набор виджетов, обеспечивая удобную навигацию и организацию сложных приложений (рис. 8.177).

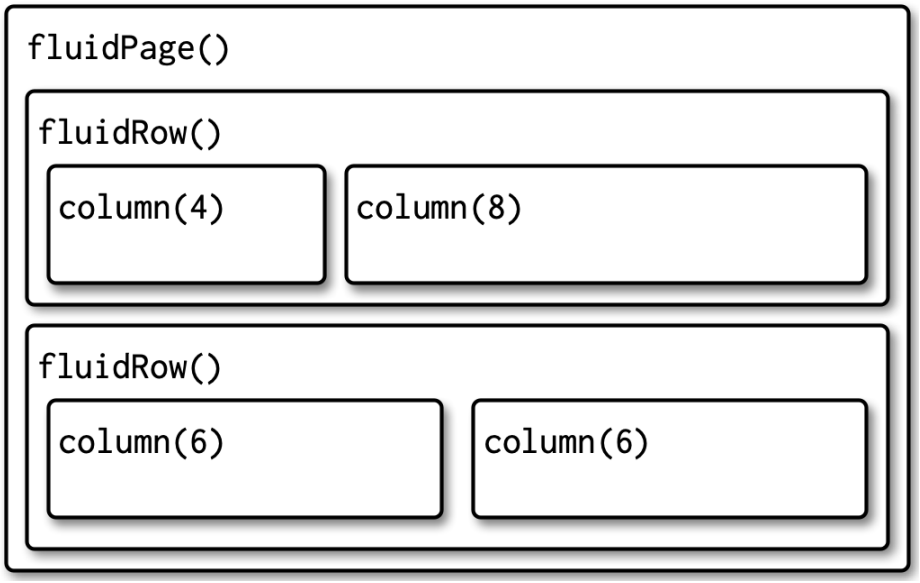

Рис. 8.176. Структура интерфейса Shiny-приложения с использованием компоновки с помощью строк и столбцов.

```
ui <- fluidPage(
  tabsetPanel(
    tabPanel("Import data",
      fileInput("file", "Data", buttonLabel = "Upload..."),
      textInput("delim", "Delimiter (leave blank to guess)", ""),
      numericInput("skip", "Rows to skip", 0, min = 0),
      numericInput("rows", "Rows to preview", 10, min = 1)
    ),
    tabPanel("Set parameters"),
    tabPanel("Visualise results")
  )
)
```

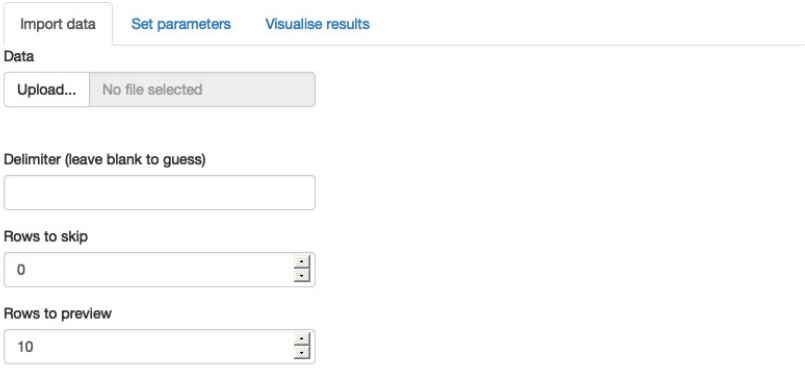

Рис. 8.177. Структура интерфейса Shiny-приложения с использованием вкладок (функция *tabsetPanel()*).

Есть и другой способ организации компоновки вкладок — с помощью навигационной панели. Она создается функцией *navlistPanel()* и размещается в левой части страницы, как и сайдбар (рис. 8.178).

```
ui <- fluidPage(
  navlistPanel(
    id = "tabset",
    "Heading 1",
    tabPanel("panel 1", "Panel one contents"),
    "Heading 2",
    tabPanel("panel 2", "Panel two contents"),
    tabPanel("panel 3", "Panel three contents")
  )
)
```

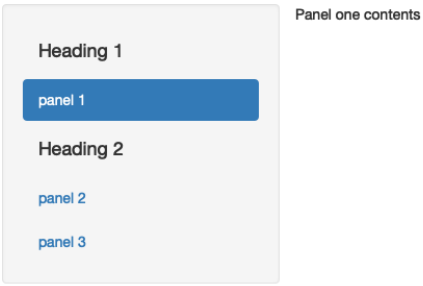

Рис. 8.178. Структура интерфейса Shiny-приложения с использованием навигационной панели (функция *navlistPanel()*).

Оба представленных способа размещают вкладки внутри страницы *fluidPage()*. Если же необходимо оформить приложение в виде отдельных «независимых» страниц, нужно использовать отдельный тип страницы — *navbarPage()* (рис. 8.179). Эта страница размещает переключатели вкладок непосредственно в строке заголовка страницы, создавая эффект навигации по разным страницам приложения. Каждый переключатель вкладки соответствует отдельной странице, содержащей свой собственный набор виджетов, что обеспечивает четкую организацию и навигацию для сложных приложений.

```
ui <- navbarPage(
  "Page title",
  tabPanel("panel 1", "one"),
  tabPanel("panel 2", "two"),
  tabPanel("panel 3", "three"),
  navbarMenu("subpanels",
    tabPanel("panel 4a", "four-a"),
    tabPanel("panel 4b", "four-b"),
    tabPanel("panel 4c", "four-c")
  )
)
```

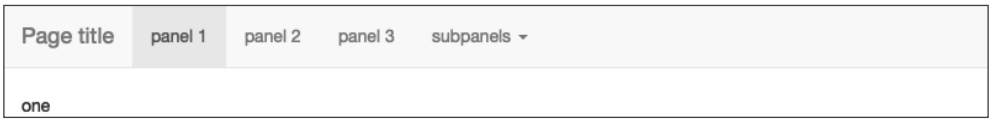

Рис. 8.179. Структура интерфейса Shiny-приложения с использованием «независимых» страниц (функция *navbarPage()*).

Следует отметить, что по умолчанию при оформлении виджетов используются стили фреймворка Bootstrap (https://getbootstrap.com). Благодаря этому, в приложениях Shiny можно достаточно быстро задавать стили оформления приложений. Однако, по умолчанию Shiny использует третью версию Bootstrap, которая в данный момент считается устаревшей. Чтобы использовать более актуальную версию Bootstrap, а также применять новые виджеты в Shiny, был создан пакет *bslib* (https://rstudio.github.io/bslib). На данный момент он уже представляет собой практически отдельный фреймворк со своими функциями для создания страниц, компоновки элементов и самостоятельных виджетов. При дальнейшем построении Shiny-приложения с дашбордом будет использоваться данный пакет.

Страницы приложения

Разрабатываемое приложение будет повторять структуру ранее созданного дашборда: оно будет состоять из двух страниц и сайдбара для элементов управления. Логика компоновки сохраняется, однако вместо стандартных функций Shiny будут использоваться функции из пакета *bslib*:

- вместо navbarPage используется *page_navbar*;
- вместо navbarMenu используется *nav_panel*.

Внутри панелей пока будет размещен обычный текст с помощью функции *p()*, которая при компиляции создает HTML-элемент * * на странице приложения. В интерфейсе можно использовать и другие HTML-теги. С полным списком доступных тегов можно ознакомиться с помощью команды names(tags), а примеры использования доступны по ссылке: https://shiny.posit.co/r/articles/build/tag-glossary.

Следует также подключить дополнительный пакет bsicons, чтобы использовать иконки. Далее представлен код, определяющий заготовку страниц приложения. На рис. 8.180 представлен результат компиляции данного кода.

```
ui <- page_navbar(
  title = "Shiny Дашборд",
  sidebar = sidebar(p("Сайдбар")),
  nav_panel("Дашборд", p("Первая страница")),
  nav_panel("Набор данных", p("Вторая страница"))
)
```

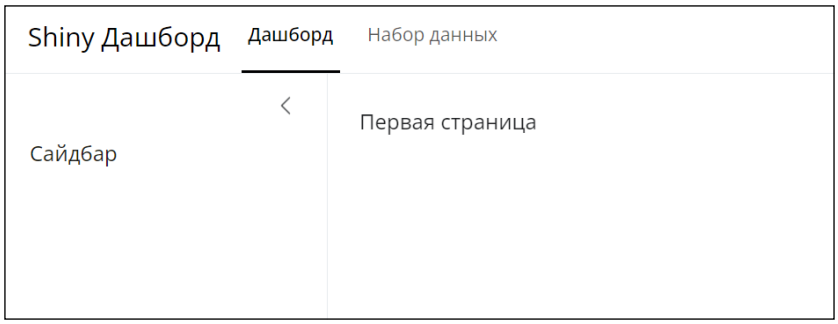

Рис. 8.180. Структура интерфейса Shiny-приложения с использованием пакета *bslib*.

Для улучшения внешнего вида приложения можно настроить его тему с помощью функции *bs_theme()*. Она позволяет переопределять основные переменные, отвечающие за цвета, шрифты и другие стили темы Bootstrap. Например, можно изменить основные и второстепенные цвета, масштаб и тип шрифта, создав уникальный и привлекательный дизайн для своего приложения.

```
bs_theme(
  version = version_default(),
  preset = NULL,
  ...,
  bg = NULL,
  fg = NULL,
  primary = NULL,
  secondary = NULL,
  success = NULL,
  info = NULL,
  warning = NULL,
  danger = NULL,
  base_font = NULL,
  code_font = NULL,
  heading_font = NULL,
  font_scale = NULL,
  bootswatch = NULL
)
```

В рамках разбираемого примера создание уникальной темы является несколько избыточной задачей, поэтому будет использоваться заранее определенную тему «Cosmo» из набора Bootswatch (https://bootswatch.com) (рис. 8.181).

```
ui <- page_navbar(
  title = "Shiny Дашборд",
  theme = bs_theme(bootswatch = "cosmo"),
  sidebar = sidebar(p("Сайдбар")),
  nav_panel("Дашборд", p("Первая страница")),
  nav_panel("Набор данных", p("Вторая страница"))
)
```

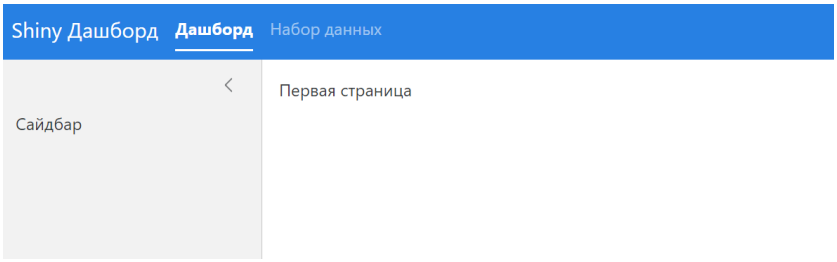

Рис. 8.181. Структура интерфейса Shiny-приложения с использованием пакета *bslib* и темы «Cosmo».

Загрузка файла

Для удобства загрузки пользователями собственных данных в сайдбар следует добавить виджет загрузки файлов (рис. 8.182). Он должен поддерживать файлы в формате CSV. Кроме того, необходимо расширить сайдбар для размещения дополнительных элементов управления. Подписи кнопок загрузчика следует отредактировать с целью предоставления пользователям понятных инструкций по его использованию.

```
ui <- page_navbar(
  title = "Shiny Дашборд",
  theme = bs_theme(bootswatch = "cosmo"),
  sidebar = sidebar(
    # Указание ширины сайдбара.
    width = 350,
    # Выбор файла для загрузки.
    fileInput("fileInputCsv", accept = ".csv",
              label = "Загрузите таблицу",
              buttonLabel = "Выберите CSV",
              placeholder = "Файл не загружен" )
  ),
  nav_panel("Дашборд", p("Первая страница")),
  nav_panel("Набор данных", p("Вторая страница"))
)
```

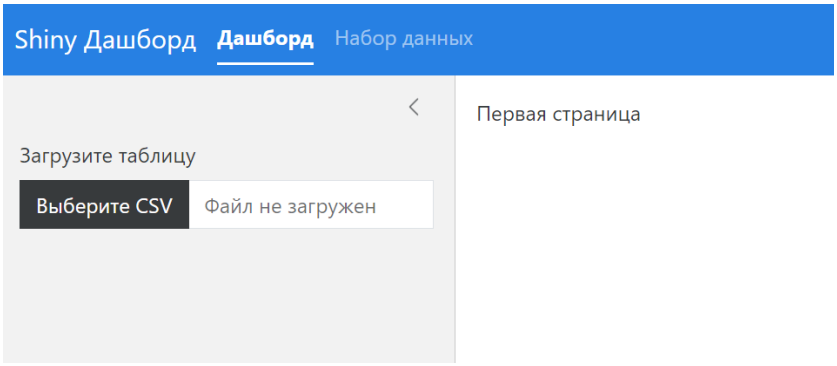

Рис. 8.182. Структура интерфейса Shiny-приложения с виджетом для загрузки файла.

После нажатия на кнопку «Выберите CSV» отображается диалоговое окно выбора файла. При выборе файла генерируется реактивное событие виджета *fileInputCsv*, которое должно обрабатываться серверной частью приложения. Во время обработки файл считывается, и на его основе создается набор данных для дальнейшей работы.

```
# Определение логики приложения.
server <- function(input, output, session) {

  # Создание набора данных из загруженного файла.
  dataset <- reactive({
    # Получение файла.
    file <- input$fileInputCsv
    # Получение расширения файла.
    ext <- tools::file_ext(file$datapath)
    # Проверка, что файл действительно был выбран.
    req(file)
    # Проверка, что файл действительно csv.
    validate(need(ext == "csv", "Пожалуйста, загрузите CSV-файл"))
    # Чтение содержимого файла.
    dataset <- read.csv2(file$datapath, dec = ".")
    dataset$DATESTRAIN <- as.Date(dataset$DATESTRAIN)
    dataset$DATEBIRTH <- as.Date(dataset$DATEBIRTH)
    dataset$DATEFILL <- as.Date(dataset$DATEFILL)
    # Возвращение результата чтения.
    dataset
  })
}
```

Создание фильтров

Теперь, когда в приложении есть реактивный набор данных *dataset*, на основе его содержимого можно сформировать набор фильтров (выпадающих списков и ползунков).

```
ui <- page_navbar(
  # ...
  sidebar = sidebar(
    # Указание ширины сайдбара.
    width = 350,
    # Выбор файла для загрузки.
    fileInput("fileInputCsv", accept = ".csv",
              label = "Загрузите таблицу",
              buttonLabel = "Выберите CSV",
              placeholder = "Файл не загружен" ),
    # Выбор группы пациентов.
    selectInput(inputId = 'selPatgroup', label = 'Группа пациентов',
                choices = c(Все = '.'), selected = "."),
    # Выбор города.
    selectInput('selCity', 'Город', choices = c(Все = '.'), selected = c("."),
                multiple = TRUE),
    # Выбор даты.
    dateRangeInput('selDateRange','Дата взятия образца',
```

```
                    start = Sys.Date(), end = Sys.Date(), min = Sys.Date(), max =
Sys.Date(),
                    format = "yyyy-mm-dd", startview = "month",
                    weekstart = 1, language = "ru", separator = " - ",
                    width = NULL, autoclose = TRUE),
    # Выбор возраста.
    sliderInput('selAge', 'Возраст',
                min = 0, max = 0, value = c(0, 0), step = 1, dragRange = TRUE)
  ),
  # ...
)
```

Со стороны серверной части нужно добавить логику для наполнения созданных фильтров реальными данными. Для таких ситуаций можно использовать функцию *observe()*, которая срабатывает при каждом событии, которое формируется в приложении. Внутри данной функции можно определить на какие именно события следует реагировать.

```
server <- function(input, output, session) {
  # ...
  observe({
    # Проверка, что набор данных не пустой
    if (!is.null(dataset())) {
      # Обновляем фильтр группы пациентов.
      updateSelectInput(session, "selPatgroup",
                        choices = c(Все = ".",  sort(unique(dataset()$PAT_
GROUP))),
                        selected = ".")
      # Обновление фильтра городов.
      updateSelectInput(session, "selCity",
                        choices = c(Все = ".",  sort(unique(dataset()$CI-
TYNAME))),
                        selected = ".")
      # Обновление фильтра дат.
      updateDateRangeInput(session, 'selDateRange',
                           start = min(dataset()$DATESTRAIN, na.rm = TRUE),
                           end = max(dataset()$DATESTRAIN, na.rm = TRUE),
                           min = min(dataset()$DATESTRAIN, na.rm = TRUE),
                           max = max(dataset()$DATESTRAIN, na.rm = TRUE)
      )
      # Обновление фильтра возраста.
      updateSliderInput(session, 'selAge',
                        min = min(dataset()$AGE, na.rm = TRUE),
                        max = max(dataset()$AGE, na.rm = TRUE),
                        value = c(min(dataset()$AGE, na.rm = TRUE), max(data-
set()$AGE, na.rm = TRUE))
      )
    }
  })
  # ...
}
```

В результате в сайдбаре появились новые виджеты, содержимое которых меняется после загрузки файла *patients.csv* в приложение. Следует отметить что, выпадающие списки и бегунки соответствуют содержимому загружаемого файла (рис. 8.183).

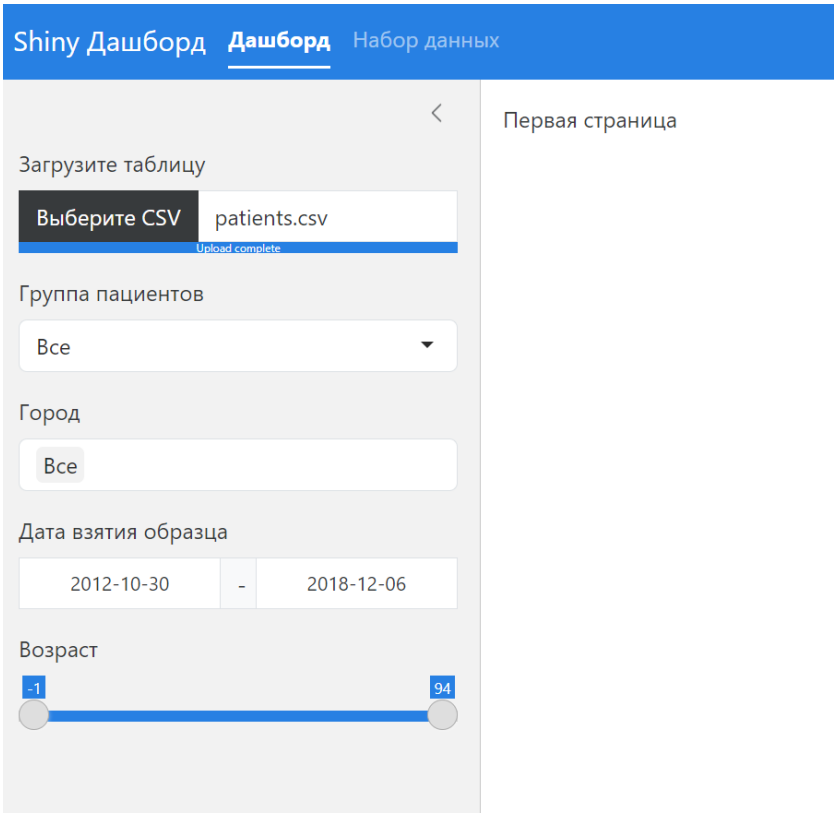

Рис. 8.183. Структура интерфейса Shiny-приложения с дополнительными виджетами для последующей фильтрации данных.

Фильтрация данных

При помощи фильтров исследователи могут отбирать конкретные данные из загруженного файла, создавая новый набор данных data для последующего использования в построении инфографики. Исходный набор данных dataset остается неизменным, пока пользователь не загрузит новый файл в приложение. Ниже представлена схема общей логики приложения, которое фильтрует данные с использованием описанных выше виджетов (рис. 8.184).

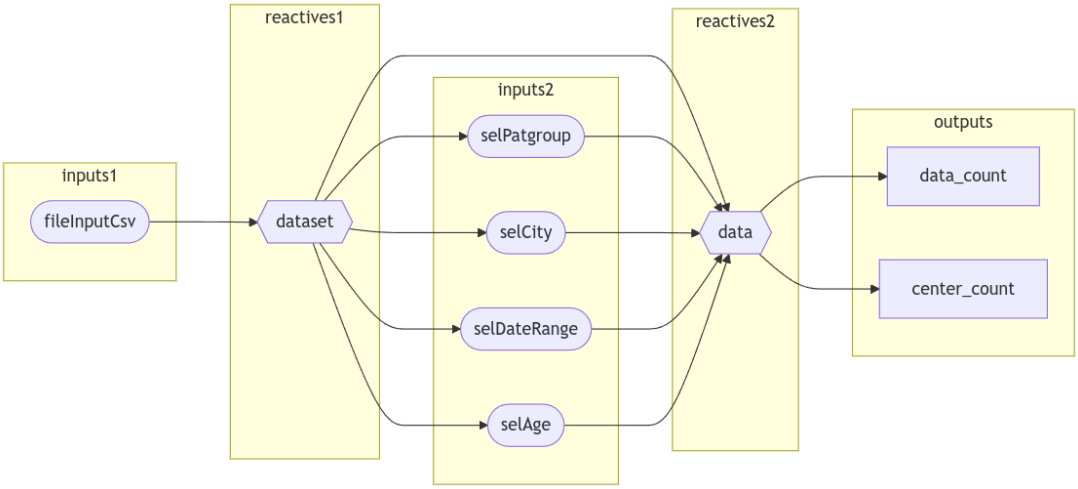

Рис. 8.184. Общая логика взаимодействия реактивных элементов.

Для достижения автоматической фильтрации данных при изменении значений виджета необходимо определить реактивный набор данных *data*.

```
server <- function(input, output, session) {
  # ...
  data <- reactive({
    # Проверка, на наличие данных из прочитанного файла.
    if (is.null(dataset())) return(NULL);
    # Копирование исходного набора.
    data <- dataset()
    # Фильтрация по группе пациентов.
    if (input$selPatgroup != "." ) {
      data <- data %>% filter(PAT_GROUP == input$selPatgroup)
    }
    # Фильтрация по городам.
    if ( !("." %in% input$selCity ) ) {
      data <- data %>% filter(CITYNAME %in% input$selCity)
    }
    # Фильтрация по дате.
    if (length(input$selDateRange) == 2) {
      data <- data %>% filter(DATESTRAIN >= input$selDateRange[1] & DATESTRAIN <=
input$selDateRange[2])
    }
    # Фильтрация по возрасту.
    if (length(input$selAge) == 2) {
      data <- data %>% filter(AGE >= input$selAge[1] & AGE <= input$selAge[2])
    }
    # Возвращение результата.
    data
  })
  # ...
}
```

Для наглядности в сайдбаре следует вывести текст с количеством отобранных образцов и центров. Для этого в сервере необходимо создает реактивный рендер виджетов на основе отфильтрованного набора *data()*.

```
server <- function(input, output, session) {
  # ...
  #### Расчет количества образцов ####
  output$data_count <- renderText({
    value <- 0
    if (!is.null(data())) {
      value <-  nrow(data())
    }
    paste('Выбрано', value, 'образцов', sep = ' ')
  })
  #### Расчет количества центров ####
  output$center_count <- renderText({
    value <- 0
    if (!is.null(data())) {
      value <- length(unique(data()$CENTER))
    }
    paste('Выбрано', value, 'центров', sep = ' ')
  })
  # ...
}
```

На стороне интерфейса необходимо вывести данные значения в виде абзацев с текстом.

```
ui <- page_navbar(
  # ...
  sidebar = sidebar(
    # ...
    # Количество образцов.
    p(textOutput("data_count")),
    # Количество центров.
    p(textOutput("center_count"))
  ),
  # ...
)
```

В результате выполнения этого программного кода пользователь может наблюдать динамическое изменение цифр в зависимости от выбранных значений фильтров (рис. 8.185).

Было написано уже достаточно много кода, описывающего разметку сайдбара. Чтобы упростить и повысить удобство кода, разметку сайдбара можно вынести в отдельный объект. Этот объект затем можно передавать в функцию создания страницы приложения в качестве параметра.

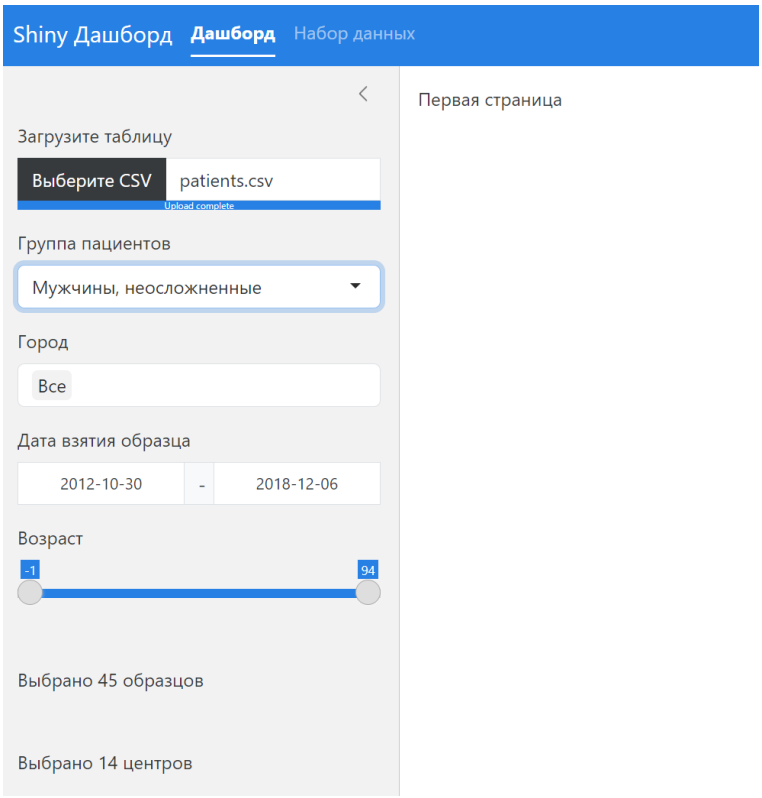

Рис. 8.185. Изменение результатов в зависимости от выбранных фильтров.

```
#### Определение сайдбара ####
sidebar <- sidebar(
  # Настройка ширины сайдбара
  width = 350,
  # Выбор файла для загрузки.
  fileInput("fileInputCsv", accept = ".csv",
            label = "Загрузите таблицу",
            buttonLabel = "Выберите CSV",
            placeholder = "Файл не загружен" ),
  # Выбор группы пациентов.
  selectInput(inputId = 'selPatgroup', label = 'Группа пациентов',
              choices = c(Все = '.'), selected = "."),
  # Выбор города.
  selectInput('selCity', 'Город', choices = c(Все = '.'), selected = c("."),
              multiple = TRUE),
  # Выбор даты.
  dateRangeInput('selDateRange','Дата взятия образца',
                 start = Sys.Date(), end = Sys.Date(), min = Sys.Date(), max =
Sys.Date(),
                 format = "yyyy-mm-dd", startview = "month",
                 weekstart = 1, language = "ru", separator = " - ",
                 width = NULL, autoclose = TRUE),
```

```
  # Выбор возраста.
  sliderInput('selAge', 'Возраст',
              min = 0, max = 0, value = c(0, 0), step = 1, dragRange = TRUE),
  # Количество образцов.
  p(textOutput("data_count")),
  # Количество центров.
  p(textOutput("center_count"))
)

#### Формирование интерфейса приложения ####
ui <- page_navbar(
  # Заголовок страницы.
  title = "Shiny Дашборд",
  # Тема оформления приложения.
  theme = bs_theme(bootswatch = "cosmo"),
  # Сайдбар с виджетами фильтрации. Код стал компактнее.
  sidebar = sidebar,
  # Первая страница с инфографикой.
  nav_panel("Дашборд", p("Первая страница")),
  # Вторая страница с таблицей.
  nav_panel("Набор данных", p("Вторая страница"))
)
```

Страница с графиками

После создания реактивного объекта *data()* с отфильтрованным набором данных можно приступить к наполнению первой страницы графиками дашборда. Для этого будут использованы функции из пакета *bslib*, а также определена компоновка элементов.

Для оптимального использования пространства необходимо добавить страницу *page_fillable()*, которая займет всю свободную область. Внутри нее с помощью функции *layout_column()* следует определить два столбца: в первом будут размещены панели с информацией о количестве пациентов, организмов и диагнозах, а во втором — карта и распределение пациентов по городам.

Для наглядности пока будут выведены только текстовые «заглушки» на месте будущих виджетов. Следует отметить, что функция *layout_columns()* предоставляет гибкие возможности для настройки размещения вложенных элементов. Подробное описание доступно по ссылке: https://rstudio.github.io/bslib/articles/column-layout/index.html. Ниже представлен код, позволяющий воссоздать описанный ранее дашборд (см. раздел 8.4.2).

```
ui <- page_navbar(
  # ...
  nav_panel("Дашборд",
    # Страница.
    page_fillable(
      # Столбцы в рамках страницы.
      layout_columns(
        # Первый столбец - количество, организмы, диагнозы.
        layout_columns(),
```

```
        # Второй столбец - карта, распределение по городам.
        layout_columns()
      )
    )
  )
  # ...
)
```

Первый столбец должен заполнять все свободное пространство — поэтому следует определить параметр *fill* = TRUE.

Все размещенные элементы внутри функции *layout_columns()* согласно концепции стилей Bootstrap считаются столбцами, а значит эти столбцы размечаются в виртуальной строке из 12 колонок слева направо. Таким образом для каждого столбца можно задать ширину от 1 до 12. Если же в строке слишком много столбцов и их суммарная ширина превосходит 12, то они будут переноситься на новую строку.

Для создания планируемого дашборда необходимо расположить элементы таким образом, чтобы в первой строке оказались цифры с количеством пациентов по полу, во второй — выделенные организмы, в третьей — диагнозы. Поэтому каждый такой элемент должен занимать максимально возможную ширину — 12 единиц. Для этого в функции *layout_columns()* следует задать параметр *col_widths = c(12, 12, 12)*, который указывает, что внутри будет размещено три элемента, каждый с шириной 12.

При такой компоновке элементы будут переноситься и формировать строки - для них тоже можно задать соотношение высоты. В этом случае вся доступная область делится на относительные части. С помощью параметра *row_heights = c(1,2,4)* можно сообщить, что первая строка должна занимать одну часть высоты, вторая должна быть в два раза больше, третья — в четыре раза больше.

В результате компоновка первого столбца будет выглядеть следующим образом.

```
layout_columns(fill = TRUE,
  # Ширина столбцов.
  col_widths = c(12, 12, 12),
  # Высота строк в относительных единицах.
  row_heights = c(1,2,4),
  # Строка 1 - Блоки с цифрами.
  layout_columns(
    p("Кол-во мужчин"),
    p("Кол-во женщин"),
    p("Кол-во детей")
  ),
  # Строка 2 - Графики с организмами.
  layout_columns(
    p("Организмы мужчин"),
    p("Организмы женщин"),
    p("Организмы детей")
  ),
  # Строка 3 -График с диагнозами.
  p("Диагнозы")
)
```

Если не задавать параметры *col_widths* и *row_heights*, элементы внутри *layout_columns()* будут размещаться таким образом, чтобы занимать одинаковую ширину и высоту, заполняя все доступное пространство.

Во втором столбце, где находится всего два элемента, достаточно задать ширину блоков с помощью *col_widths = c(12, 12)*. Определение высоты строк можно опустить, и виджеты автоматически разделят доступное пространство поровну.

```
layout_columns(fill = TRUE,
  # Ширина столбцов.
  col_widths = c(12, 12),
  # Строка 1 - Карта.
  p("Карта"),
  # Строка 2 - Таблица.
  p("Таблица с городами")
)
```

Целиком разметка страницы будет выглядеть следующим образом (рис. 8.186).

```
ui <- page_navbar(
  # Заголовок страницы.
  title = "Shiny Дашборд",
  # Тема оформления приложения.
  theme = bs_theme(bootswatch = "cosmo"),
  # Сайдбар с виджетами фильтрации.
  sidebar = sidebar,
  # Первая страница с инфографикой.
  nav_panel("Дашборд",
    # Страница.
    page_fillable(
      # Столбцы в рамках страницы.
      layout_columns(
        # Первый столбец - количество, организмы, диагнозы.
        layout_columns(fill = TRUE,
          # Ширина столбцов.
          col_widths = c(12, 12, 12),
          # Высота строк в относительных единицах.
          row_heights = c(1,2,4),
          # Строка 1 - Блоки с цифрами.
          layout_columns(
            p("Кол-во мужчин"),
            p("Кол-во женщин"),
            p("Кол-во детей")
          ),
          # Строка 2 - Графики с организмами.
          layout_columns(
            p("Организмы мужчин"),
            p("Организмы женщин"),
            p("Организмы детей")
          ),
          # Строка 3 -График с диагнозами.
          p("Диагнозы")
        ),
```

```
        # Второй столбец - карта, распределение по городам.
        layout_columns(fill = TRUE,
          # Ширина столбцов.
          col_widths = c(12, 12),
          # Строка 1 - Карта.
          p("Карта"),
          # Строка 2 - Таблица.
          p("Таблица с городами")
        )
      )
    )
  ),
  # Вторая страница с таблицей.
  nav_panel("Набор данных", p("Вторая страница"))
)
```

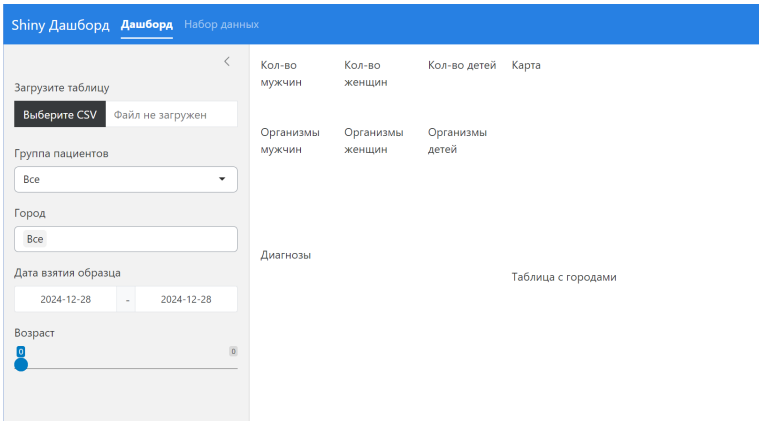

Рис. 8.186. Дашборд с разметкой и текстовыми «заглушками».

Для упрощения процесса работы и повышения модульности перед наполнением страницы виджетами можно вынести элементы кода в отдельные объекты, аналогично тому, как это было сделано с сайдбаром.

```
info_numbers <- layout_columns(
  p("Кол-во мужчин"),
  p("Кол-во женщин"),
  p("Кол-во детей")
)
info_org <-  layout_columns(
  p("Организмы мужчин"),
  p("Организмы женщин"),
  p("Организмы детей")
)
info_diag <- p("Диагнозы")

info_left <-  layout_columns(fill = TRUE,
 # Ширина столбцов.
 col_widths = c(12, 12, 12),
 # Высота строк в относительных единицах.
```

```
 row_heights = c(1,2,4),
 info_numbers, # Строка 1 - Блоки с цифрами.
 info_org, # Строка 2 - Графики с организмами.
 info_diag # Строка 3 -График с диагнозами.
)

#### Определение правого столбца инфографики ####
info_map <- p("Карта")
info_map_table <- p("Таблица с городами")

info_right <-   layout_columns(fill = TRUE,
 # Ширина столбцов.
 col_widths = c(12, 12),
 # Высота строк.
 row_heights = c(1, 1),
 info_map, # Строка 1 - Карта.
 info_map_table # Строка 2 - Таблица.
)

#### Определение интерфейса приложения ####
ui <- page_navbar(
  # Заголовок страницы.
  title = "Shiny Дашборд",
  # Тема оформления приложения.
  theme = bs_theme(bootswatch = "cosmo"),
  # Сайдбар с виджетами фильтрации.
  sidebar = sidebar,
  # Первая страница с инфографикой.
  nav_panel("Дашборд",
            # Страница.
            page_fillable(
              # Столбцы в рамках страницы.
              layout_columns(
                info_left, # Первый столбец - количество, организмы, диагнозы.
                info_right # Второй столбец - карта, распределение по городам.
              )
            )
  ),
  # Вторая страница с таблицей.
  nav_panel("Набор данных", p("Вторая страница"))
)
```

Виджет valueBox

Для создания блоков с количеством пациентов по полу будет использован виджет value_box (https://rstudio.github.io/bslib/articles/value-boxes/index.html) из пакета *bslib*, который позволяет вывести текстовое значение, добавить к нему иконку, дополнительное описание и т. д. (рис. 8.187).

Данный виджет состоит из 4 основных частей:

- *value* — значение, которое необходимо вывести;
- *title* — заголовок, который будет размещен над значением;
- *showcase* — опциональный элемент, который будет размещен рядом со значением. В данном случае это будет иконка;

- *showcase_layout* — позволяет задать принцип компоновки элементов внутри блока. По умолчанию используется *'left center'*. Возможно размещение *'top right'*;
- *theme* — тема оформления блока, позволяет задать его цвет и внешний вид.

Дополнительно внутри блока можно размещать еще виджеты, они будут размещаться непосредственно под значением. Для наполнения виджета *value_box* прежде всего необходимо в server рассчитать необходимые показатели для групп пациентов.

```
# Количество мужчин.
output$male_count <- reactive({
  data() %>% filter(grepl("Мужчины",PAT_GROUP)) %>% nrow()
})
# Количество женщин.
output$female_count <- reactive({
  data() %>% filter(grepl("Женщины",PAT_GROUP)) %>% nrow()
})
# Количество детей.
output$children_count <- reactive({
  data() %>% filter(grepl("Дети",PAT_GROUP)) %>% nrow()
})
```

Далее необходимо создать блоки для вывода этих показателей в *info_numbers*. Можно также задать альтернативную компоновку элементов, например, для количества детей, с помощью параметра *showcase_layout*. Кроме того, можно добавить дополнительную текстовую подпись к этим блокам.

```
info_numbers <- layout_columns(
  value_box(
    title = "Мужчины", fill = TRUE,
    value = textOutput("male_count"),
    showcase = bs_icon("person-standing"),
    theme = "blue"
  ),
  value_box(
    title = "Женщины", fill = TRUE,
    value = textOutput("female_count"),
    showcase = bs_icon("person-standing-dress"),
    theme = "purple"
  ),
  value_box(
    title = "Дети",  fill = TRUE,
    value = textOutput("children_count"),
    showcase = bs_icon("person-circle"),
    showcase_layout = 'top right',
    theme = "orange",
    p("Без учета наличия/отсутсвия осложнений")
  )
)
```

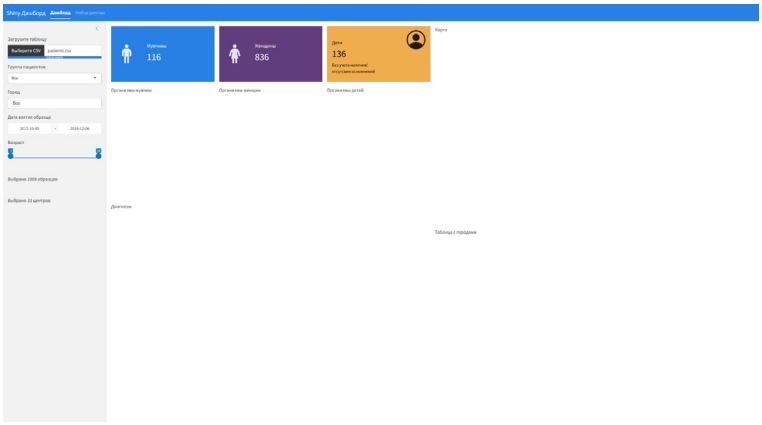

Рис. 8.187. Дашборд с виджетами *value_box*.

Виджет card

Одним из самых часто используемых виджетов для представления информации в разрабатываемом дашборде является карточка *card*. В документации к пакету *bslib* описано множество вариантов для ее настройки (подробнее см. https://rstudio.github.io/bslib/articles/cards/index.html).

В разрабатываемом приложении в карточках будут размещаться графики, при этом карточки должны иметь способность разворачиваться на весь экран, чтобы можно было рассмотреть все детали (рис. 8.188).

Кроме того в *server* следует добавить расчет графиков для выделенных организмов среди групп пациентов. Для упрощения, необходимо вынести повторяющиеся участки кода в отдельные функции, а именно — фильтрацию набора данных и непосредственное построение графика.

```
# Функция для фильтрации данных по группам.
org_filter <- function(data, group_name) {
  data() %>%
    filter(grepl(group_name,PAT_GROUP)) %>%
    group_by(STRAIN) %>%
    summarise(Count = n()) %>%
    ungroup() %>%
    mutate(Percent = round(100 * Count / sum(Count))) %>%
    arrange(desc(Percent)) %>%
    mutate(csum = rev(cumsum(rev(Count))),
           pos = Count/2 + lead(csum, 1),
           pos = if_else(is.na(pos), Count/2, pos))
}

# Функция для отрисовки графика ggplot2.
org_plot <- function(data) {
  ggplot(data, aes(x = "" , y = Count, fill = fct_inorder(STRAIN))) +
    geom_col(width = 1, color = 1) +
    coord_polar(theta = "y") +
    scale_fill_brewer(palette = "Pastel1") +
```

```
    geom_label_repel(data = data,
                     aes(y = pos, label = paste0(Count, " (", Percent, "%)")),
                     size = 4.5, nudge_x = 1, show.legend = FALSE) +
    guides(fill = guide_legend(title = "Организм")) +
    theme_void()
}

# Организмы у мужчин.
output$plot_org_male <- renderPlot({
  org_male <- org_filter(data(), "Мужчины")
  validate(need(nrow(org_male) > 0, "Данные отсутствуют"))
  org_plot(org_male)
})

# Организмы у женщин.
output$plot_org_female <- renderPlot({
  org_female <- org_filter(data(), "Женщины")
  validate(need(nrow(org_female) > 0, "Данные отсутствуют"))
  org_plot(org_female)
})

# Организмы у детей.
output$plot_org_children <- renderPlot({
  org_children <- org_filter(data(), "Дети")
  validate(need(nrow(org_children) > 0, "Данные отсутствуют"))
  org_plot(org_children)
})
```

На интерфейсе графики будут отображаться в карточках, а чтобы их можно было развернуть на весь экран следует установить параметр full_screen = TRUE.

```
info_org <-  layout_columns(
  card(full_screen = TRUE, plotOutput("plot_org_male")),
  card(full_screen = TRUE, plotOutput("plot_org_female")),
  card(full_screen = TRUE, plotOutput("plot_org_children"))
)
```

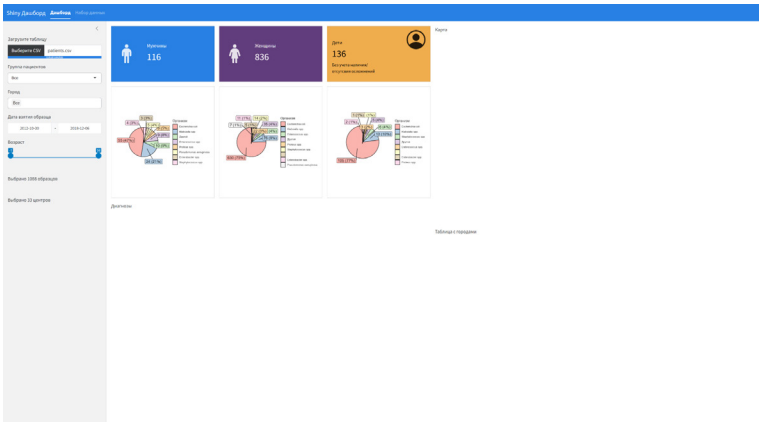

Рис. 8.188. Дашборд с карточками *card* и графиками.

Также можно добавить график распределения диагнозов по группам пациентов в блоке *server*.

```
output$plot_diag <- renderPlotly({
  # Формирование нужных наборов данных.
  diag <- data() %>% group_by(PAT_GROUP, mkb_name) %>%
    summarise(Count = n()) %>%
    ungroup() %>%
    pivot_wider(names_from = "PAT_GROUP", values_from = "Count", values_fill = 0)
%>%
    mutate(mkb_name = case_when(
      mkb_name == "Интерстициальный цистит (хронический)" ~ "Хронический
цистит",
      mkb_name == "Необструктивный хронический пиелонефрит, связанный с
рефлюксом" ~ "Хронический пиелонефрит",
      mkb_name == "Острый тубулоинтерстициальный нефрит" ~ "Острый нефрит",
      mkb_name == "Инфекция мочевыводящих путей без установленной локализации" ~
"Инфекция МВП",
      TRUE ~ mkb_name
    ))
  # Проверка наличия данных.
  validate(
    need(nrow(diag) > 0, "Данные отсутствуют")
  )
  # Определение заранее возможных категорий.
  categories <- c(
    'Дети, неосложненные', 'Дети, осложненные',
    'Женщины, неосложненные', 'Женщины, осложненные',
    'Мужчины, неосложненные', 'Мужчины, осложненные'
  )
  # Добавление недостающих столбцов для упрощенного построения графика.
  diag_plot <- diag
  for (i in seq_along(categories)) {
    category <- categories[i]
    if (!(category %in% colnames(diag_plot))) {
      diag_plot[[category]] <- 0
    }
  }
  # Построение графика.
  plot_ly(data = diag_plot, x = ~mkb_name, type = 'bar',
                  y = ~`Дети, неосложненные`, name = 'Дети, неосложненные',
                  marker = list(color = 'rgba(247,167,102, 0.8)')) %>%
    add_trace(y = ~`Дети, осложненные`, name = 'Дети, осложненные',
              marker = list(color = 'rgba(243,109,0, 0.8)')) %>%
    add_trace(y = ~`Женщины, неосложненные`, name = 'Женщины, неосложненные',
              marker = list(color = 'rgba(134,96,142, 0.8)')) %>%
    add_trace(y = ~`Женщины, осложненные`, name = 'Женщины, осложненные',
              marker = list(color = 'rgba(108,48,130, 0.8)')) %>%
    add_trace(y = ~`Мужчины, неосложненные`, name = 'Мужчины, неосложненные',
              marker = list(color = 'rgba(39,188,209, 0.8)')) %>%
    add_trace(y = ~`Мужчины, осложненные`, name = 'Мужчины, осложненные',
              marker = list(color = 'rgba(36,107,206, 0.8)')) %>%
```

```
    layout(yaxis = list(title = 'Кол-во'), barmode = 'stack') %>%
    layout(xaxis = list(title = '' )) %>%
    layout(legend = list(orientation = 'h'))
})
```

В этом случае карточку следует определить по другому — отдельно указать заголовок и само тело карточки. В данном случае в качестве заголовка карточки используется простой текст, но могут фигурировать и более сложные элементы, т. к. функции *card_header()* и *card_body()* задают контейнеры, которые могут заполняться различными элементами (рис. 8.189).

```
info_diag <- card(full_screen = TRUE,
  card_header("Структура диагнозов по группам пациентов"),
  card_body(plotlyOutput("plot_diag"))
)
```

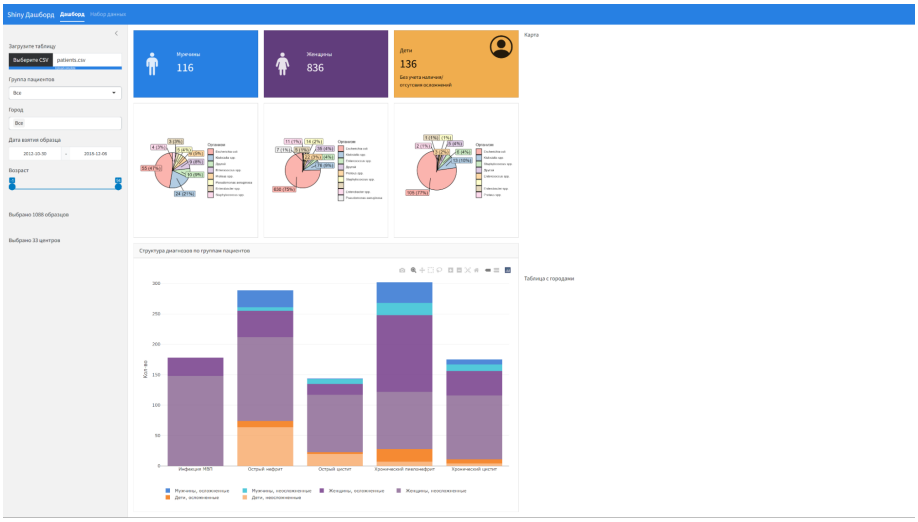

Рис. 8.189. Дашборд с карточками *card* графиком структуры диагнозов.

Схожим образом можно создать карточку для отображения карты. Для этого в блоке *server* необходимо описать создание соответствующего объекта.

```
output$map <- renderLeaflet({
  # Все пациенты.
  all <- data() %>% select(CITYNAME, LATITUDE, LONGITUDE) %>%
    group_by(CITYNAME, LATITUDE, LONGITUDE) %>%
    summarise(Count = n()) %>%
    ungroup()
  # Мужчины.
  men <- data() %>%
    filter(grepl("Мужчины",PAT_GROUP)) %>%
    select(CITYNAME, LATITUDE, LONGITUDE) %>%
    group_by(CITYNAME, LATITUDE, LONGITUDE) %>%
    summarise(CountMen = n()) %>%
    ungroup()
```

```
  # Женщины.
  women <- data() %>%
    filter(grepl("Женщины",PAT_GROUP)) %>%
    select(CITYNAME, LATITUDE, LONGITUDE) %>%
    group_by(CITYNAME, LATITUDE, LONGITUDE) %>%
    summarise(CountWoman = n()) %>%
    ungroup()
  # Дети.
  children <- data() %>%
    filter(grepl("Дети",PAT_GROUP)) %>%
    select(CITYNAME, LATITUDE, LONGITUDE) %>%
    group_by(CITYNAME, LATITUDE, LONGITUDE) %>%
    summarise(CountChild = n()) %>%
    ungroup()
  # Набор данных для отображения на карте.
  mapdata <- all %>% left_join(men) %>% left_join(women) %>% left_join(children)
%>%
    mutate_all(~replace(., is.na(.), 0))
  # Проверка наличия данных.
  validate(need(nrow(mapdata) > 0, "Данные отсутствуют"))
  # Создание карты.
  mapdata %>%
    leaflet() %>%
    addCircleMarkers(
      lng = ~ LONGITUDE,
      lat = ~ LATITUDE,
      stroke = FALSE,
      fillOpacity = 0.5,
      radius = ~ scales::rescale(sqrt(Count), c(1, 10)),
      label = ~ paste(
        "<strong>" , CITYNAME, ": ", Count,        "</strong>",
        "<br/>",
        "Мужчин:", CountMen, "<br/>",
        "Женщин:",  CountWoman, "<br/>",
        "Дети:",  CountChild
      ) %>% map(html),
      labelOptions = c(textsize = "15px")) %>%
    addTiles("http://services.arcgisonline.com/arcgis/rest/services/Canvas/
World_Light_Gray_Base/MapServer/tile/{z}/{y}/{x}")
})
```

На интерфейсе необходимо разместить соответствующую карточку (рис. 8.190).

```
info_map <- card(full_screen = TRUE,
  card_header("Количество пациентов по городам"),
  card_body(leafletOutput("map")))
```

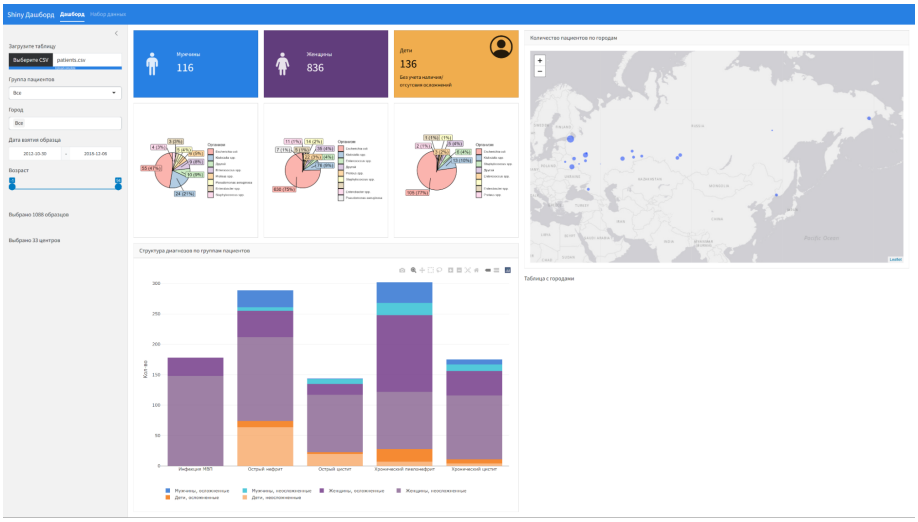

Рис. 8.190. Дашборд с интерактивной картой.

Виджет navset_card_tab

В карточках можно размещать несколько вкладок рядом с заголовком и в дальнейшем переключаться между их содержимым. Для этого используется функция *navset_card_tab()*, в которой размещаются навигационные панели *nav_panel()*. Параметры навигационных панелей во многом совпадают с параметрами карточек.

В этих вкладках будут размещены таблицы с подсчетом пациентов в различных городах. В разрабатываемом дашборде будут использованы три формата вывода таблиц: *kable*, *flextable* и *gt*.Сначала в блок *server* необходимо добавить код для создания объекта citypat, который содержит таблицу с необходимыми данными.

```
citypat <- reactive({
  citypat <- data() %>%
    group_by(CITYNAME, PAT_GROUP) %>% summarise(Count = n()) %>%
    ungroup() %>%
    pivot_wider(names_from = "PAT_GROUP", values_from = "Count", values_fill = 0)
%>%
    select(order(colnames(.)))

  colnames(citypat)[1] <- "Город"
  citypat
})
```

После этого можно создать непосредственно интерактивные таблицы на основе объекта *citypat*.

```
# Таблица gt
output$citypat_gt <- render_gt({
  # Проверка на отсутствие данных.
  validate(need(nrow(citypat()) > 0, "Данные отсутствуют"))
  # Создание таблицы.
  citypat() %>%
```

```
    gt() %>%
    tab_header(title = "Распределение пациентов по городам") %>%
    tab_spanner(label = "Дети", columns = starts_with("Дети")) %>%
    tab_spanner(label = "Женщины", columns = starts_with("Женщины")) %>%
    tab_spanner(label = "Мужчины", columns = starts_with("Мужчины")) %>%
    cols_label(
      ends_with(", неосложненные") ~ "Неосложненные",
      ends_with(", осложненные") ~ "Осложненные"
    ) %>%
    opt_row_striping()
})

# Таблица flextable.
output$citypat_ft <- renderUI({
  # Проверка на отсутствие данных.
  validate(need(nrow(citypat()) > 0, "Данные отсутствуют"))
  # Создание таблицы.
  citypat() %>%
    flextable() %>%
    separate_header(split = ", ") %>%
    labelizor(part = "header",
              labels = c("diagnosis" = "Диагноз",
                         "sex" = "Пол",
                         "n" = "Случаев",
                         "percent" = "Процент",
                         "diag" = "В группе",
                         "overall" = "Всего")) %>%
    set_caption("Распределение пациентов по городам") %>%
    theme_zebra() %>%
    htmltools_value()
})

# Таблица kable.
output$citypat_kb <- function(){
  # Проверка на отсутствие данных.
  validate(need(nrow(citypat()) > 0, "Данные отсутствуют"))
  # Создание таблицы.
  citypat() %>%
    kbl(caption = "Распределение пациентов по городам") %>%
    kable_styling(bootstrap_options = c("striped"))
}
```

Затем каждую таблицу необходимо вывести на отдельной вкладке в интерфейсе дашборда (рис. 8.191).

```
info_map_table <- navset_card_tab(
  full_screen = TRUE,
  title = "Распределение пациентов по городам",
  nav_panel("kable", tableOutput("citypat_kb")),
  nav_panel("gt", gt_output("citypat_gt")),
  nav_panel("flextable", tableOutput("citypat_ft"))
)
```

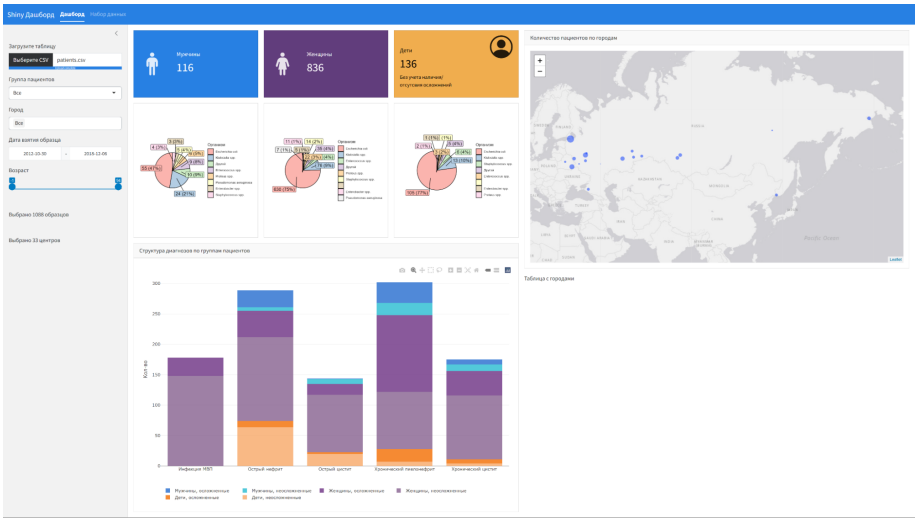

Рис. 8.191. Дашборд с добавленными таблицами.

На этом этапе оформление первой страницы дашборда завершено.

Размещение без компоновки

Создание сетки компоновки элементов требуется, когда необходимо разместить несколько элементов, обеспечить их масштабируемость, отзывчивость и так далее. В случае, если требуется разместить всего один элемент — такой подход можно не использовать. Далее на второй странице будет размещена таблица *reactable*, которая будет отображать отфильтрованные (отобранные) данные. Для этого в блоке server необходимо определить подписи кнопок таблицы на русском языке и определение самой таблицы.

```
# Локализованные подписи кнопок.
options(reactable.language = reactableLang(
  pageSizeOptions   = "показано {rows} значений",
  pageInfo          = "с {rowStart} по {rowEnd} из {rows} строк",
  pagePrevious      = "назад",
  pageNext          = "вперед",
  searchPlaceholder = "Поиск...",
  noData            = "Значения не найдены"
))
# Создание таблицы.
output$data_rt <- renderReactable({
  # Проверка на отсутствие данных.
  validate(need(nrow(data()) > 0, "Данные отсутствуют"))
  # Создание таблицы.
  data() %>%
    # Исключение ненужных столбцов.
    select(-c("LATITUDE", "LONGITUDE")) %>%
    # Выбор необходимых столбцов в нужном порядке.
    select(study_subject_id, PAT_GROUP, SEX, AGE, DATEBIRTH,
           STRAIN, DATESTRAIN,
           CENTER, COUNTRY, CITYNAME, DATEFILL,
```

```
            DIAG_ICD, mkb_name, COMPL) %>%
    # Создание реактивной таблицы.
    reactable(filterable = TRUE, searchable = TRUE, striped = TRUE,
              # Параметры отображения столбцов.
              columns = list(
                study_subject_id = colDef(name = "ID", width = 64, defaultSor-
tOrder = "asc"),
                PAT_GROUP = colDef(name = "Группа", width = 150),
                SEX = colDef(name = "Пол", width = 100),
                AGE = colDef(name = "Возраст", width = 90),
                DATEBIRTH = colDef(name = "Дата рожд.", width = 120),
                STRAIN = colDef(name = "Организм", width = 150),
                DATESTRAIN = colDef(name = "Дата получ.", width = 120),
                CENTER = colDef(name = "Центр", width = 70),
                COUNTRY = colDef(name = "Страна", width = 100),
                CITYNAME = colDef(name = "Город", width = 150),
                DATEFILL = colDef(name = "Дата заполн.", width = 120),
                DIAG_ICD = colDef(name = "МКБ-10", width = 80),
                mkb_name = colDef(name = "Диагноз"),
                COMPL = colDef(name = "Осложнения")
              ))
})
```

Таблица будет выведена непосредственно в nav_panel (рис. 8.192).

```
ui <- page_navbar(
  # ...
  # Вторая страница с таблицей.
  nav_panel("Набор данных", reactableOutput("data_rt"))
)
```

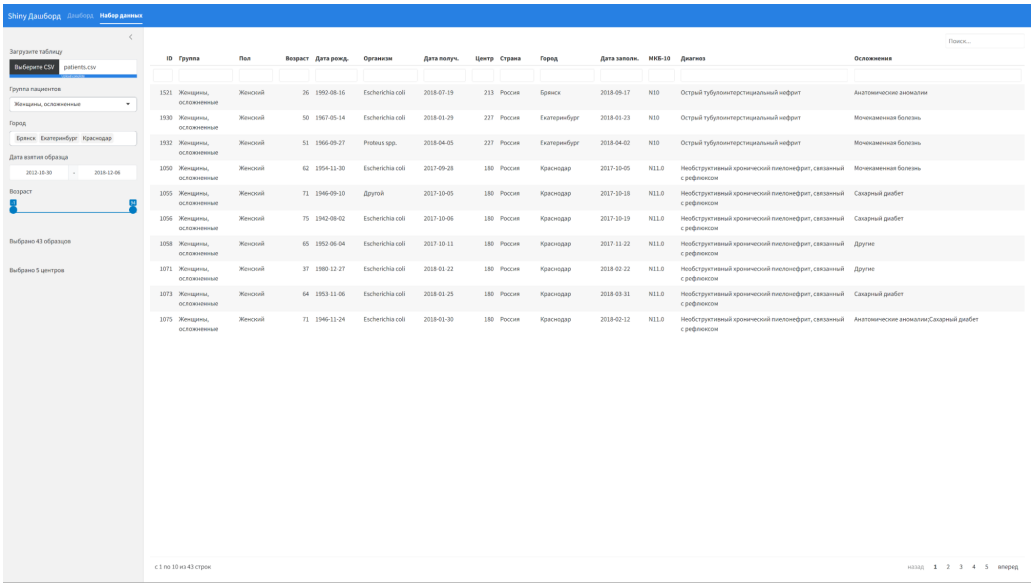

Рис. 8.192. Вторая страница дашборда с отфильтрованной таблицей.

Скачивание файла

Поскольку создается полноценное приложение, функционал не должен ограничиваться только загрузкой и анализом данных — необходимо предусмотреть сохранение полученных результатов в какой-либо форме. Для этих целей следует добавить кнопку, которая позволит скачать отфильтрованный набор данных. В Shiny данный функционал реализуется с использованием специального виджета *downloadButton()*. Он создает кнопку, которая будет генерировать событие скачивания. Кроме того, для этой кнопки можно задать подпись и иконку. Данная кнопка в разрабатываемом дашборде будет размещена в заголовке страницы. Для этого, после описания страниц приложения необходимо добавить виджет *nav_spacer* который будет заполнять свободное место в заголовке, а затем виджет *nav_item*, в котором будет размещена кнопка. В результате кнопка будет отображаться в правой части заголовка (рис. 8.193).

```
ui <- page_navbar(
  # ...
  # Вторая страница с таблицей.
  nav_panel("Набор данных", reactableOutput("data_rt")),
  # Заполняет свободное место.
  nav_spacer(),
  # Кнопка скачивания.
  nav_item(downloadButton(outputId = "downloadButton", label = "Скачать таблицу",
icon = icon("download"), class = "btn-primary"))
)
```

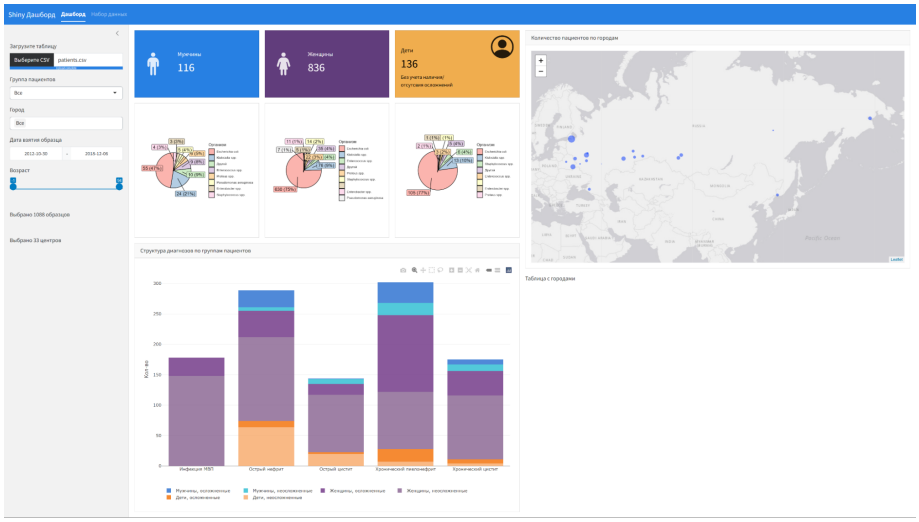

Рис. 8.193. Дашборд с добавленной кнопкой для скачивания данных в правой части заголовка страницы.

Визуально на данном этапе кнопка отображается неактивной, потому что для нее не назначен обработчик события. Чтобы обрабатывать нажатие кнопки в *server* нужно объявить обработчик события *downloadHandler()*. В этом обработчике потребуется определить имя скачиваемого файла и его содержимое. В качестве

файла может выступать любой объект, в данном случае это будет документ Excel с реактивным набором данных *data*. Чтобы сохранить файл для скачивания вместо пути сохранения нужно указать объект *file* — таким образом результат будет записан во временный файл и уже потом будет отдаваться на скачивание.

```
output$downloadButton <- downloadHandler(
  # Имя скачиваемого файла.
  filename = function() {
    "table.xlsx"
  },
  # Файл, который будет скачан.
  content = function(file) {
    write.xlsx(data(), file, asTable = TRUE)
  }
)
```

После этого кнопка станет активной. Однако если ее нажать, но перед этим не загрузить файл в приложение, то она будет выдавать пустую страницу. В связи с этим необходимо добавить дополнительную проверку на существование набора данных, который необходимо выгрузить. Для этих целей необходимо создать новую кнопку *actionButton()*, которая по нажатию будет вызывать отдельное событие с проверкой наличия данных. В коде ниже она подписана как «*Скачать таблицу с проверкой*».

```
ui <- page_navbar(
  # ...
  nav_spacer(),
  # Кнопка скачивания.
  nav_item(downloadButton(outputId = "downloadButton", label = "Скачать таблицу",
icon = icon("download"), class = "btn-primary")),
  # Кнопка скачивания с проверкой.
  nav_item(actionButton("downloadActionButton", "Скачать таблицу с проверкой"))
)
```

Итого сейчас код интерфейса для примера содержит две кнопки:

- *downloadButton* («*Скачать таблицу*»), которая скачивает таблицу без проверки на наличие данных;
- *actionButton* («*Скачать таблицу с проверкой*»), которая позволит вывести отдельное окно для скачивания таблицы после проверки на наличие данных.

Чтобы кнопка «Скачать таблицу с проверкой» выполнила планируемую функцию, необходимо в блоке *server* добавить отслеживание события нажатия данной кнопки с помощью функции *observeEvent()*. Данная функция будет обрабатывать нажатие кнопки *downloadActionButton*. Внутри обработчика *observeEvent()* также будет происходить проверка на наличие набора данных *data()*. Если в объекте *data()* содержатся данные, то будет показано диалоговое окно с кнопкой скачать, а если нет — диалоговое окно будет содержать текстовое сообщение.

```
observeEvent(input$downloadActionButton, {
  # Если файл загружен.
  if (!is.null(input$file1)) {
```

```
    # Показать диалог скачивания.
    showModal(modalDialog(
      # Заголовок диалога.
      title = "Скачать набор данных",
      # Сообщение внутри диалога.
      p(paste("Набор данных включает", nrow(data()), "образцов", sep = " ")),
      footer = list(
        # Кнопка скачивания.
        downloadButton(outputId = "downloadModalButton",
                       label = "Скачать"),
        # Кнопка закрытия диалога.
        modalButton("Закрыть")
      )
    ))
  } else {
    # Если данные отсутствуют, показать сообщение.
    showModal(modalDialog(
      # Заголовок диалога.
      title = "Данные отсутствуют",
      # Сообщение внутри диалога.
      p("Загрузите файл или выберите другие значения фильтров"),
      # Кнопка закрытия диалога
      footer = list(modalButton("Закрыть"))
    ))
  }
})
```

Следует обратить внимание, что теперь в приложении появилась новая кнопка скачивания с идентификатором *downloadModalButton*. Таким образом, при нажатии кнопки *downloadActionButton* происходит проверка на наличие данных, и в случае наличия данных открывается диалоговое окно. Данное диалоговое окно содержит новую кнопку — *downloadModalButton*, которая отвечает непосредственно за скачивание данных). Для данной кнопки необходимо создать свой обработчик скачивания данных.

```
output$downloadModalButton <- downloadHandler(
  # Имя скачиваемого файла.
  filename = function() {
    "table.xlsx"
  },
  # Файл, который будет скачан.
  content = function(file) {
    # Проверка на отсутствие данных.
    validate(need(nrow(data()) > 0, "Данные отсутствуют"))
    # Создание таблицы.
    write.xlsx(data(), file, asTable = TRUE)
  }
)
```

Сообщение, которое будет получено при нажатии «*Скачать таблицу с проверкой*» при отсутствии данных, представлено рис. 8.194.

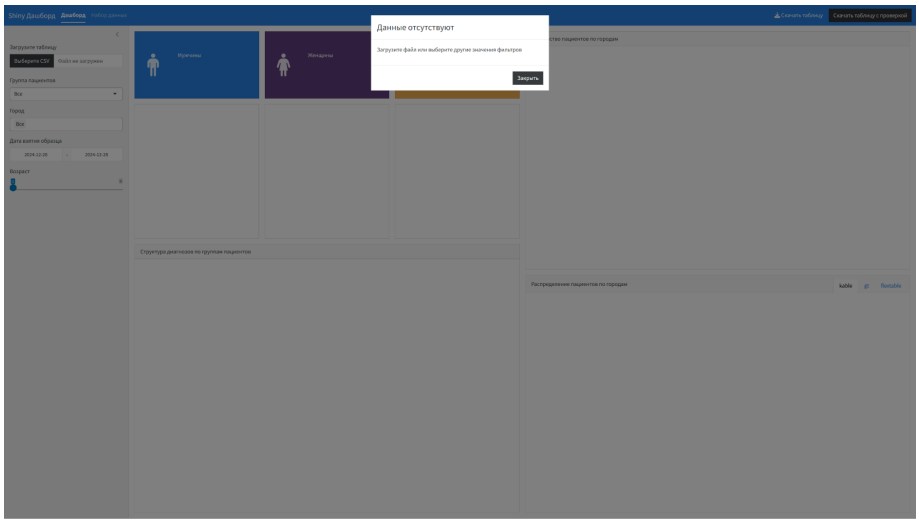

Рис. 8.194. Дашборд с предупреждением об отсутствии данных.

Если нажать кнопку «*Скачать таблицу с проверкой*» после загрузки файла в приложение, то файл будет доступен для скачивания (рис. 8.195).

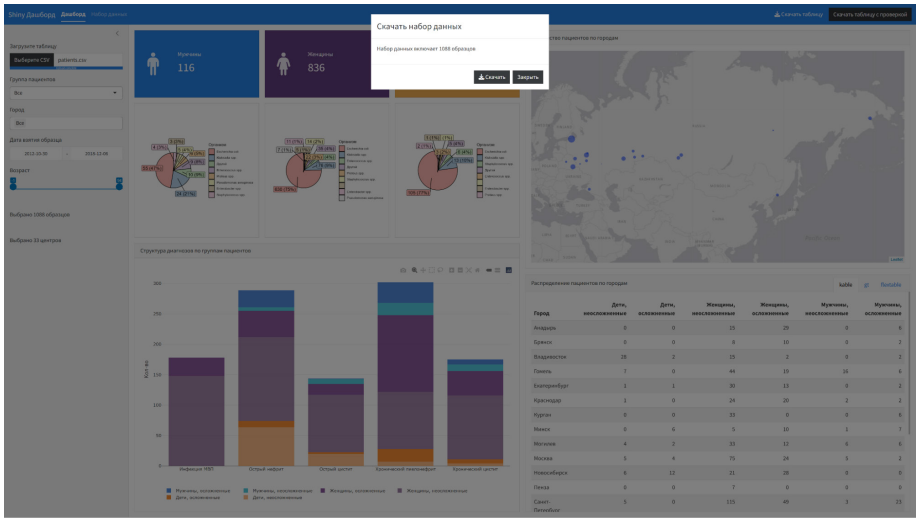

Рис. 8.195. Скачивание файла в дашборде после проверки на отсутствие данных.

Генерация отчета

Естественным желанием исследователя помимо сохранения отфильтрованного файла является получение полноценного отчета с таблицами, графиками и текстовыми комментариями. Данную задачу можно реализовать с помощью *downloadHandler()* несколькими способами: сформировать полноценный файл с помощью пакетов *openxlsx* или *officer* (см. раздел 8.2.4–8.2.6) или использовать отчеты в формате RMarkdown (см. раздел 8.2.7). Первый способ является более трудозатратным, однако более гибким. В тоже время RMarkdown удобен для соз-

дания типовых универсальных решений с разнообразными настройками формата экспорта. Далее будет представлен пример формирования отчета в Shiny-приложении с использованием RMarkdown.

Прежде всего необходимо создать шаблон отчета в файле *report.Rmd* прямо в папке проекта и описать в нем следующую преамбулу:

```
title: Отчет по исследованию
output:
  html_document:
    toc: true
    self_contained: true
params:
  data: NULL
```

Согласно описанной выше преамбуле отчет будет выводиться в формате HTML. Данный формат представляет широкие возможности для интерактивной визуализации данных. Следует обратить внимание, что по умолчанию все служебные файлы, которые потребуются для отрисовки отчета RMarkdown (таблицы стилей CSS, код JavaScript) сохраняются в отдельной папке рядом с результирующим файлом HTML и подгружаются только в момент открытия отчета. В создаваемом приложении необходимо обеспечить чтобы HTML-файл отчета был самодостаточным, т.е. содержал в себе все необходимые компоненты (а не генерировал их в отдельной папке, как это происходит по умолчанию при использовании RMarkdown). Поэтому необходимо указать параметр *self_contained: true*, чтобы при создании выходного документа все служебные файлы не сохранялись отдельно, а встраивались непосредственно в документ.

Кроме того, в отчете следует определить входные параметры, в частности набор данных *data*, который служит основой для его формирования. Эти данные передаются непосредственно из Shiny-приложения.

Для использования данных из входных параметров необходимо в отчет добавить следующий код:

```
```{r, include=FALSE}
data <- params$data
```
```

В остальном содержимое отчета может содержать любые элементы визуализации и отображения.

Для запуска процедуры формирования отчета необходимо добавить новую кнопку «*Создать отчет*».

```
ui <- page_navbar(
  # ...
  nav_spacer(),
  # ...
  # Кнопка генерации и скачивания отчета.
  nav_item(downloadButton(outputId = "generateButton", label = "Создать отчет")
)
```

Создание отчета может занять некоторое время, поэтому в обработчике клика новой кнопки необходимо добавить сообщение об индикации выполнения процесса. После завершения генерации отчета это сообщение следует удалить.

```
output$generateButton <-  downloadHandler(
  filename = "report.html",
  content = function(file) {
    # Отображение сообщения о выполняемых действиях.
    id <- showNotification("Подготовка отчета...",
                           duration = NULL, closeButton = FALSE)
    # Когда работа функции закончится, убрать сообщение.
    on.exit(removeNotification(id), add = TRUE)
    # Запуск процедуры.
    render("report.Rmd", output_file = file,
           params = list(data = data()),
           envir = new.env(parent = globalenv()))
  }
)
```

При запуске приложения следует указать параметр *output_file = file* в функции *render()* для загрузки сгенерированного документа. Также потребуется создать среду выполнения *envir = new.env(parent = globalenv())*, чтобы генератор имел доступ к необходимым пакетам. После нажатия кнопки «*Создать отчет*» появится сообщение в правом нижнем углу экрана «*Подготовка отчета*». Через некоторое время файл *report.html* будет загружен (рис. 8.196, 8.197).

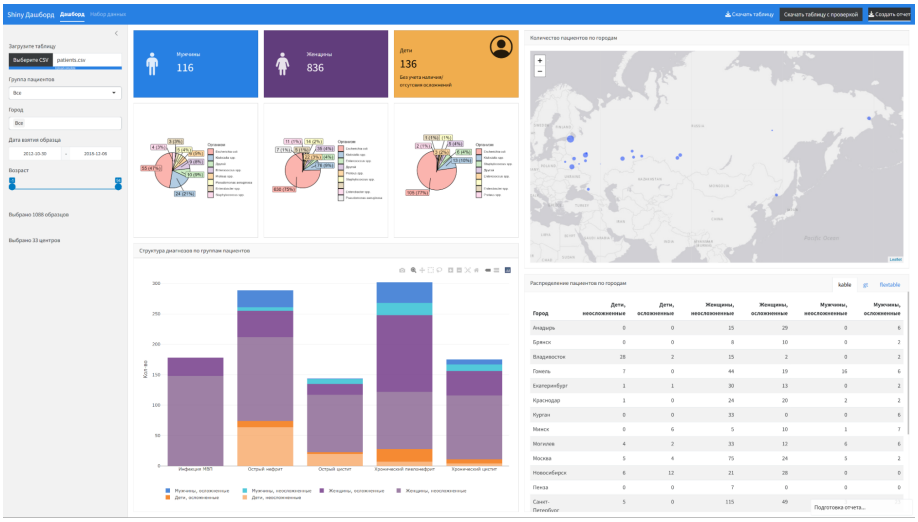

Рис. 8.196. Всплывающее сообщение в правом нижнем углу экрана при подготовке отчета.

Отчет по исследованию

Пациенты

В набор данных включено 1088 пациентов из 33 центров.

Распределение пациентов по полу:

- Мужчины: 116
- Женщины: 836
- Дети: 136

Структура выделенных организмов

Мужчины

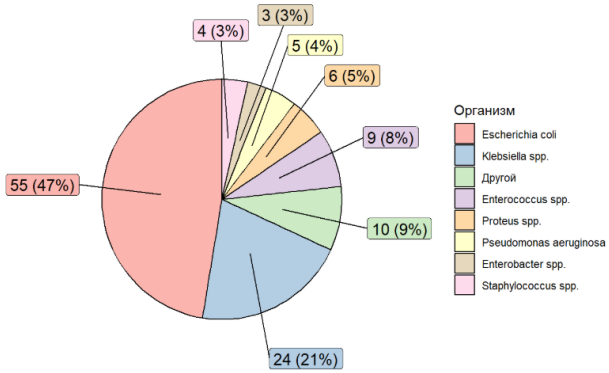

Рис. 8.197. Созданный отчет *report.html*.

8.4.4. Публикация приложения Shiny

Созданное приложение Shiny возможно опубликовать с помощью кнопки «Publish application» в RStudio (рис. 8.198).

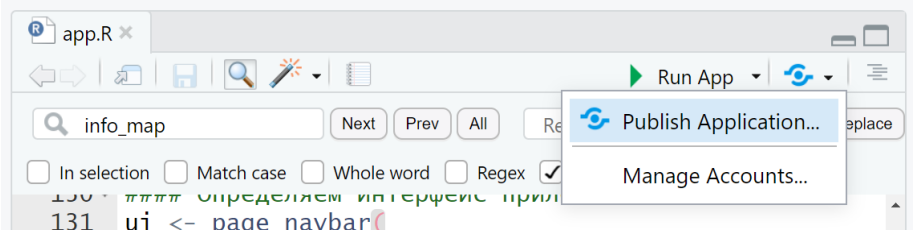

Рис. 8.198. Кнопка «Publish application» в RStudio.

По умолчанию в RStudio для публикации Shiny-приложения предлагаются следующие сервисы (рис. 8.199).

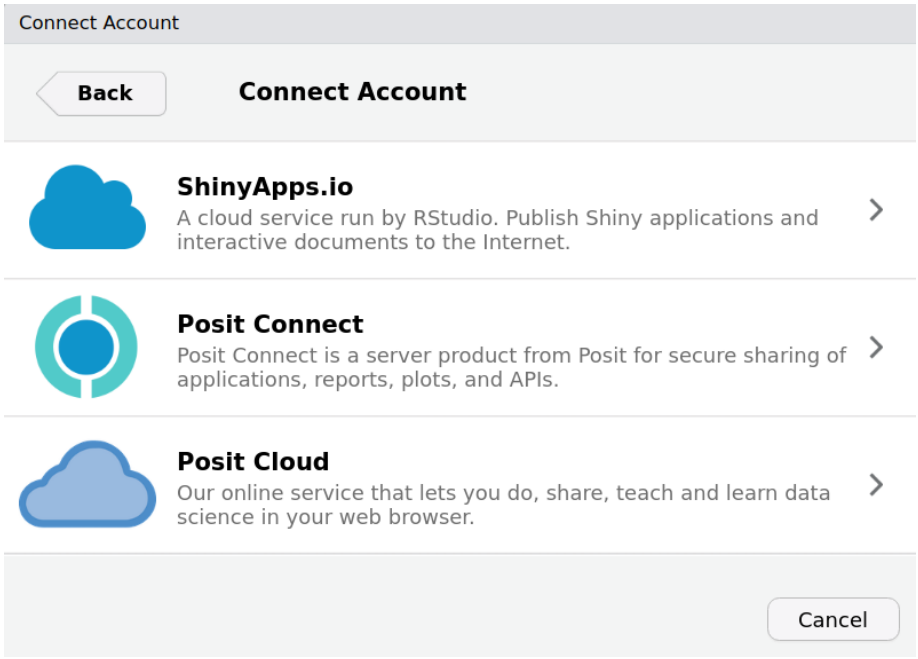

Рис. 8.199. Варианты публикации Shiny-приложения в Rstudio.

Все представленные варианты требуют предварительной регистрации учетной записи. Они предлагают как платные тарифы, так и бесплатные с ограничениями на количество публикуемых приложений и время их работы. Процесс публикации обычно автоматизированный и состоит из нескольких шагов с помощью мастера. Благодаря простому интерфейсу он не вызывает затруднений.

8.4.5. Запуск Shiny приложения с помощью технологий контейнеризации

Введение в Docker

После разработки полноценного приложения для работы с данными необходимо рассмотреть альтернативные варианты его развертывания и распространения, исключающие использование сервисов Posit. Такой подход полезен для размещения приложений на собственных серверах с неограниченным использованием, или для локального запуска на компьютерах без установленного R или RStudio.

Для решения этой задачи можно использовать технологию контейнеризации Docker (https://www.docker.com). Docker позволяет создавать, развертывать и запускать приложения в изолированных средах, называемых контейнерами. Популярность Docker обусловлена его переносимостью: для запуска контейнера требуется только Docker, а необходимые компоненты находятся в самом контейнере.

Docker позволяет создавать образы, представляющие неизменяемые шаблоны для создания контейнеров. Образы содержат базовый образ операционной системы, код приложения, библиотеки и необходимые файлы.

Контейнер представляет собой запущенный экземпляр образа. Docker позволяет одновременно запускать множество изолированных контейнеров на одном компьютере, не оказывая существенного влияния на систему.

Для работы с Docker можно использовать приложение Docker Desktop (https://www.docker.com/products/docker-desktop) или инструменты командной строки Docker CLI (https://www.docker.com/products/cli). Наиболее универсальным вариантом является Docker CLI.

Образы создаются на основе Dockerfile — кода, определяющего образ. Dockerfile обычно хранится в репозитории Git. Этот файл должен содержать набор инструкций, следуя которым Docker будет собирать образ контейнера. Этот файл содержит описание базового образа, который будет представлять собой исходный слой образа. Среди популярных официальных базовых образов можно отметить python, ubuntu, alpine.

Образы обычно хранятся в реестрах (registry), которые похожи на репозитории Git. Самый распространенный реестр для общедоступных контейнеров — Docker Hub, который позволяет размещать общедоступные и частные образы на бесплатных и платных условиях. Docker Hub включает официальные образы для операционных систем и языков программирования, а также множество контейнеров, созданных сообществом. Кроме того, возможность использования реестра контейнером для репозитория Git предоставляется и в сервисе Gitlab. Для публикации образов в реестрах необходима учетная запись в этих сервисах.

Процесс работы с Docker можно описать следующим образом (рис. 8.200):

1. С помощью команд, описанных в Dockerfile собирается образ — команда *docker build*;
2. Данный образ сохраняется локально на компьютере для последующего использования или публикуется в реестре (для возможности скачивания на другие компьютеры или серверы) — команда *docker push*;
3. При необходимости образ может быть скачан из реестра на компьютер с установленным Docker, где предполагается создать и запустить контейнер — команда *docker pull* (такой сценарий возможен, например, при необходимости распространения созданного приложения на другие компьютеры или серверы);
4. Запуск экземпляра контейнера из созданного образа — команда *docker run* (следует отметить, что из созданного образа можно запустить множество копий контейнера).

Таким образом, контейнер Docker можно рассматривать как отдельно функционирующее приложение, которое помимо кода приложения содержит все необходимое: начиная от библиотек и пакетов, и заканчивая операционной системой. В связи с этим у контейнеров есть одно из главных преимуществ: относительно полная независимость (нужен лишь Docker). Кроме того можно запускать несколько контейнеров с одним и тем же приложением. Они не будут конфликтовать друг с другом (стоит просто задать им разные имена). Образ контейнера следует рассматривать как условный «установочный файл приложения», из которого на одном и том же

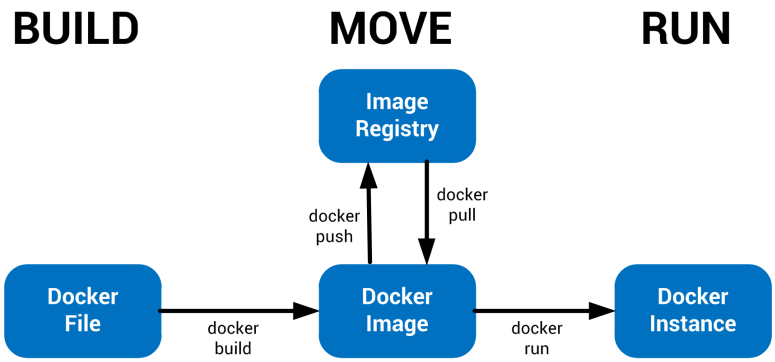

Рис. 8.200. Алгоритм работы с технологией контейнеризации Docker.

компьютере можно «установить» одно и тоже приложение несколько раз. При этом данные «установленные» приложения будут существовать одновременно, и будут изолированы друг от друга и не будут конфликтовать.

Необходимо отметить, что если при запуске контейнера командой *docker run* образ отсутствует на компьютере, то он будет автоматически загружен. Следовательно, команду *docker pull* для извлечения образа из реестра при запуске контейнера применять необязательно.

В представленном далее примере реестр контейнеров не используется: образ собирается и контейнер запускается локально. Поэтому команды *docker push* и *docker pull* не рассматриваются.

Следует уделить несколько слов наименованию и версионированию образов. Чтобы различать образы используются имена и теги. Имена обычно имеют вид *<user>/<image name>*, например *rocker/shiny* (https://hub.docker.com/r/rocker/shiny), но могут и иметь только *image name*, например *ubuntu* (https://hub.docker.com/_/ubuntu). Главное, чтобы имя образа было уникальным в рамках реестра, потому что оно должно однозначно определять тип используемого образа.

Теги указывают на версии и варианты образов и указываются после названия через символ «:», например *rocker/shiny:4.4.1* или *ubuntu:22.04*. Если не указан тег в названии контейнера, то используется тег по умолчанию — latest.

Команды Docker

Все основные команды Docker, которые чаще всего используются при работе с образами и контейнерами представлены ниже:

- *docker ps* — показать список активных контейнеров;
- *docker ps -a* – показать все контейнеры, включая остановленные;
- *docker stop <имя_контейнера>* — остановить контейнер;
- *docker rm <имя_контейнера>* — удалить контейнер;
- *docker images* — показать список всех локальных образов;
- *docker pull <имя_образа>* — загрузить образ из Docker Hub;
- *docker rmi <имя_образа>* — удалить локальный образ;
- *docker build -t <имя_образа>:<тег> <путь_к_Dockerfile>* — сборка образа на основе Dockerfile;

- *docker push <имя_реестра>/<имя_образа>:<тег>* — отправка образа в Docker Hub или другой реестр;
- *docker run -p <локальный_порт>:<контейнерный_порт> <имя_образа>* — запуск контейнера на основе указанного образа.

Команда *docker run <имя_образа>* запускает контейнер — отдельный экземпляр образа *<имя_образа>*. Однако для полноценного использования контейнера в большинстве случаев потребуется указать дополнительные параметры (флаги).

Флаг *-name <имя_контейнера>* задает имя контейнера, который будет запущен, чтобы его можно было отличить от других работающих контейнеров. Если не указать имя, при запуске каждый экземпляр контейнера получит случайный буквенно-цифровой идентификатор.

Флаг *-rm* автоматически удаляет контейнер после завершения его работы. Если не использовать флаг -rm, то контейнер будет оставаться на месте до тех пор, пока не удалить его вручную с помощью команды docker rm.

Флаг *-d* запустит контейнер в фоновом режиме и контейнер не будет блокировать командную строку и выводить в нее свои сообщения. Данный флаг обычно используется при развертывании на сервере или когда предполагается длительная автономная работа контейнера. При разработке не рекомендуется применять этот флаг, чтобы иметь возможность отслеживать происходящие внутри контейнера процессы и своевременно обнаруживать ошибки или сообщения.

Флаг *-p <локальный_порт>:<порт_контейнера>* позволяет сопоставить внутренние сетевые порты контейнера с сетевыми портами компьютера, на котором запущен контейнер. Это необходимо для того, чтобы обеспечить связь запущенного внутри контейнера приложения с внешним миром.

Создание образа из Dockerfile

Dockerfile — это набор текстовых инструкций для создания образа Docker. Обычно он содержит инструкции, которые предполагается выполнять в командной строке.

При создании Dockerfile необходимо помнить, что результирующий образ является неизменяемым. Следовательно, все включенные в образ компоненты остаются неизменными во времени. В контексте примера необходимо настроить версии R и установить системные пакеты в Dockerfile. В зависимости от назначения образа можно скопировать код, данные и/или пакеты R с локального компьютера в создаваемый образ или установить их непосредственно в операционной системе образа во время его создания.

Существует множество команд Dockerfile, но для создания образов в большинстве случаев используются следующие команды:

- *FROM* — указание на базовый образ, который будет использоваться при сборке. Обычно в первой строке файла Dockerfile.
- *RUN* — Выполнение команды в командной строке внутри собранного образа.
- *COPY* — Копирование файлов внутрь контейнера.
- *EXPOSE* — Открытие порта внутри контейнера для доступа из хоста (компьютера, на котором запускается контейнер).
- *CMD* — Команда, которую необходимо выполнить при старте контейнера. Обычно это последняя строка Dockerfile.

Обычно образы не собираются с нуля, а используют так называемые «базовые образы», которые обычно содержат операционную систему и определенный набор установленных пакетов.

Каждая команда в Dockerfile определяет новый слой, предыдущие слои при этом остаются неизменными. Преимущество Docker в том, что он перестраивает только те слои, в которых производились изменения.

Далее представлен пример простейшего Dockerfile:

```
FROM ubuntu:latest                  # Базовый образ
COPY my-data.csv /data/data.csv     # Копирование файла в образ в папку data
RUN ["head", "/data/data.csv"]      # Вывод первых строк файла
```

В данном файле определяются три слоя. Если потребуется изменить команду *head* на *tail* и выводить последние строки файла, то необходимо будет пересобрать только последний слой, потому что предыдущие строки Dockerfile остались неизменными.

После создания Dockerfile можно собрать образ с помощью команды *docker build -t <имя_образа> <путь_к_Dockerfile>*. Если не указать тег для образа, по умолчанию будет использоваться latest.

Затем можно отправить образ в DockerHub или другой реестр с помощью команды *docker push <имя_образа>* или запустить контейнер *docker run*.

Более подробно с основами Docker можно ознакомиться по ссылке (https://habr.com/ru/companies/ruvds/articles/438796).

Сборка образа и запуск контейнера с приложением Shiny

Далее рассмотрен процесс создания образа и запуска контейнера для разработанного приложения Shiny.

Прежде всего необходимо сохранить информацию об используемой версии R и установленных пакетах (и их версиях), чтобы обеспечить наилучшую воспроизводимость результатов. Для этого будет использован пакет *renv*. Если он уже установлен, необходимо инициализировать его с помощью команды *renv::init()*. Следует отметить, что если при создании проекта Shiny в RStudio была отмечена галочка «Use renv with this project», то это действие можно пропустить, так как инициализация уже была выполнена.

После инициализации пакета renv на локальном компьютере необходимо выполнить команду *renv::snapshot()*, которая создаст «снимок» всех пакетов (и их версий), используемых в проекте, а также сохранит используемую версию R. Все результаты будут записаны в файл *renv.lock*. Далее файл *renv.lock*, содержащий информацию об используемых пакетах и их версиях, будет скопирован «внутрь» создаваемого образа Docker и использован для загрузки и восстановления всех пакетов необходимых версий. Ниже представлен пример Dockerfile для разрабатываемого Shiny-приложения.

```
# Использование базового образа нужной версии согласно renv.lock.
# https://hub.docker.com/u/rocker/
FROM rocker/shiny:4.4.1
# Установка необходимых пакетов debian для корректной работы пакетов R.
RUN apt-get update -qq && apt-get -y --no-install-recommends install \
```

```
    libxml2-dev \
    libcairo2-dev \
    libsqlite3-dev \
    libpq-dev \
    libssh2-1-dev \
    unixodbc-dev \
    libcurl4-openssl-dev \
    libssl-dev \
    libharfbuzz-dev \
    libfribidi-dev \
    libfreetype6-dev \
    libpng-dev \
    libtiff5-dev \
    libjpeg-dev \
    libgdal-dev
# Обновление системных пакетов.
RUN apt-get update && \
    apt-get upgrade -y && \
    apt-get clean
# Копирование файла renv с описанием окружения.
COPY ./renv.lock ./renv.lock
# Копирование файла проекта в папку /app внутри образа.
COPY . ./app
# Устанавливаем renv.
RUN Rscript -e 'install.packages("renv")'
# Восстановление всех пакетов, используемых в проекте.
RUN Rscript -e 'renv::restore()'
# Открытие порта, через который будет доступно приложение.
EXPOSE 3838
# Запуск Shiny-приложения при старте контейнера.
CMD ["R", "-e", "shiny::runApp('/app', host = '0.0.0.0', port = 3838)"]
```

В первой строке представленного Dockerfile задается используемый базовый образ *rocker/shiny* — он основан на Ubuntu, в котором уже установлен язык R и пакеты, необходимые для запуска приложений Shiny. Данный подход устраняет необходимость ручной установки указанных пакетов.

Затем в систему Ubuntu устанавливаются специфические пакеты, необходимые для корректной работы R-пакетов, которые будут использоваться в конкретном приложении Shiny, таких как *openxlsx*, *rmarkdown* и т. д.

После этого копируются файлы приложения и восстанавливается окружение приложения согласно сохраненной конфигурации *renv*. В результате формируется приложение, полностью готовое к запуску.

В последних двух строках определяются порт 3838 и возможность доступа к приложению из компьютера-хоста, в котором запущен контейнер, через данный порт. Команда *EXPOSE* 3838 сообщает, что приложение, запущенное внутри контейнера, будет доступно извне через порт 3838, что позволяет открыть его в браузере.

В последней строке приводится пример команды для запуска приложения Shiny из командной строки.

После создания Dockerfile необходимо запустить процесс сборки образа. Образ будет назван my-shiny-app-image. Сборка запускается из командной строки, находящейся в каталоге проекта. Поэтому путь к Dockerfile можно указать с помощью точки («.»). Обычно, если файл для сборки образа называется Dockerfile, то его имя можно опустить. В случае, если имя отличается от стандартного (например, *Dockerfile.prod*), путь к нему задается как *./Dockerfile.prod*.

```
docker build -t my-shiny-app-image .
```

Первоначальная сборка образа может занимать достаточно длительное время, потому что необходимо собрать все слои образа, чтобы установить все необходимые пакеты. При последующих сборках потребуется пересобирать только слои, начиная со строки *COPY ./renv.lock ./renv.lock* и процесс будет занимать меньше времени.

Для запуска контейнера из собранного образа необходимо использовать команду:

```
docker run -d --rm --name my-shiny-app-container -p 3838:3838 my-shiny-app-image
```

В представленной команде используются следующие параметры:

- *-d* — приложение не будет выводить логи в командную строку;
- *--rm* — после остановки контейнера он будет удален (не будет оставаться в остановленном состоянии на компьютере и не будет занимать место на диске);
- *--name my-shiny-app-container* — указание произвольного имени контейнера;
- *-p 3838:3838* — сопоставление порта 3838 компьютера с портом 3838 внутри контейнера, чтобы приложение можно было открыть в браузере по адресу http://localhost:3838.

После запуска контейнера в командную строку будет выведен хеш-идентификатор контейнера.

Просмотреть список запущенных контейнеров можно с помощью команды *docker ps*.

```
CONTAINER ID   IMAGE              COMMAND                  CREATED              STATUS              PORTS                    NAMES
85d1bace4c55   my-shinyapp-image  "R -e 'shiny::runApp…"   About a minute ago   Up About a minute   0.0.0.0:3838->3838/tcp   my-shiny-app-container
```

Остановить контейнер можно с помощью команды docker stop my-shiny-app-container. Проверить, действительно ли контейнер был удален с компьютера можно с помощью команды docker *ps -a*.

```
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

В выводе командной строки не представлен созданный и затем удаленный контейнер с названием *my-shiny-app-container*.

8.4.6. Публикация Shiny приложения на Github Pages с помощью shinylive

Несмотря на то, что приложения Shiny предполагают использование серверной части, тем не менее существует возможность запускать созданные приложения целиком в браузере. Это стало возможно благодаря пакету *shinylive* (https://posit-dev.github.io/r-shinylive).

Этот пакет позволяет транслировать код R приложения Shiny в формат WebAssembly и сформировать статический сайт из HTML, CSS и JavaScript. Подобные статические сайты получались при компиляции документов с использованием Quarto. Такие сайты можно публиковать на GithubPages и других сервисах, т.к. они не требуют серверной части для запуска. Однако в реальной практике такая трансляция не всегда удается — некоторые пакеты содержат код, который невозможно транслировать таким образом. Второй проблемой может стать большой размер получаемых файлов. Если в случае использования «классического» серверного приложения на пользователя ложится только часть нагрузки и передается малая часть данных, нужная только для отображения страницы, а основная работа происходит на сервере, то в предлагаемом подходе пользователю придется загружать все файлы к себе и выполнять все вычисления на своем компьютере (в браузере).

Так, например, скомпилированная статическая версия разработанного учебного приложения занимает около 300 МБ. Браузеру придется сначала загрузить весь этот объем данных к себе в память, а затем обработать его и запустить. Даже при успешном запуске производительность такого большого приложения будет весьма сомнительной.

Однако для небольших и компактных Shiny-приложений такой подход может быть оправдан.

Для примера далее рассмотрена заготовка приложения, которая создается при инициализации нового проекта Shiny-application в RStudio.

```
library(shiny)
# Define UI for application that draws a histogram
ui <- fluidPage(
    # Application title
    titlePanel("Old Faithful Geyser Data"),
    # Sidebar with a slider input for number of bins
    sidebarLayout(
        sidebarPanel(
            sliderInput("bins", "Number of bins:", min = 1, max = 50, value =
30)
        ),
        # Show a plot of the generated distribution
        mainPanel(
           plotOutput("distPlot")
        )
    )
)
# Define server logic required to draw a histogram
server <- function(input, output) {
    output$distPlot <- renderPlot({
        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)
        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', border = 'white',
             xlab = 'Waiting time to next eruption (in mins)',
             main = 'Histogram of waiting times')
```

```
    })
}
# Run the application
shinyApp(ui = ui, server = server)
```

Для реализации предлагаемого подхода необходимо установить пакет *shinylive*.

```
install.packages("shinylive")
```

Для экспорта приложения необходимо выполнить команду *shinylive::export()*. Первым аргументом указывается папка, в которой находится Shiny-приложение, вторым аргументом — в какую папку сохранять скомпилированный сайт. Если запускать команду в командной строке из папки проекта и сайт требуется сохранить в папку *site*, команда будет выглядеть следующим образом:

```
shinylive::export(".", "site")
```

Компиляция займет некоторое время, а по завершении будет выведено сообщение:

```
√ Shinylive app export complete..
```

Открыть скомпилированный сайт в браузере можно с помощью пакета *httpuv* и функции *runStaticServer()*.

```
httpuv::runStaticServer("site/")
```

Если данное приложение хранится в репозитории GitHub, то можно добавить Workflow для автоматической сборки и публикации собранного статического сайта на GitHub Pages с помощью команды:

```
usethis::use_github_action(url="https://github.com/posit-dev/r-shinylive/blob/
actions-v1/examples/deploy-app.yaml")
```

Информация о подходах к непрерывному развертыванию также представлена в разделе 8.3.6. Подробная информация о подходах Workflow с использованием *shinylive* доступна по адресу (https://github.com/posit-dev/r-shinylive/tree/actions-v1/examples).