МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Уральский государственный лесотехнический университет»
(УГЛТУ)

Д. Д. Стратонов

Shiny.
Методы и инструменты для разработки интерактивных веб-приложений на языке программирования R

Учебное пособие

Екатеринбург
УГЛТУ
2025

УДК 004.774(075.8)
ББК 32.973.2я73
С83

Рецензенты:

ГАПОУ Свердловской области «Екатеринбургский монтажный колледж», зав. отделением автоматики и электромеханики *Н. А. Софьина*;

Г. А. Фролова, директор ООО «Прайм Регион»

Стратонов, Дмитрий Дмитриевич.

С83 Shiny. Методы и инструменты для разработки интерактивных веб-приложений на языке программирования R : учебное пособие / Д. Д. Стратонов ; Министерство науки и высшего образования Российской Федерации, Уральский государственный лесотехнический университет. – Екатеринбург : УГЛТУ, 2025. – 154 с.

ISBN 978-5-94984-942-2

Учебное пособие «Shiny. Методы и инструменты для разработки интерактивных веб-приложений на языке программирования R» является вспомогательным учебно-методическим обеспечением самостоятельной работы обучающихся по программе курса «Прикладная информатика». Пособие посвящено созданию интерактивных веб-приложений с помощью языка программирования R. В книге подробно рассмотрены основы языка R, принципы работы с библиотекой Shiny, а также продвинутые техники создания сложных и масштабируемых приложений. Книга будет полезна как начинающим разработчикам, так и опытным пользователям R, желающим расширить свои навыки в области создания интерактивной визуализаций данных.

Предназначено для обучающихся, осваивающих образовательные программы по направлению «Прикладная информатика» всех форм обучения.

Издается по решению редакционно-издательского совета Уральского государственного лесотехнического университета.

УДК 004.774(075.8)
ББК 32.973.2я73

ISBN 978-5-94984-942-2

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данное учебное пособие посвящено глубокому погружению в мир создания интерактивных веб-приложений с использованием мощной комбинации языка программирования R и библиотеки Shiny. Курс направлен на развитие у обучающихся не только технических навыков программирования, но и аналитического мышления, необходимого для эффективной работы с данными.

Обучаясь по этому курсу, вы освоите основы языка R, научитесь создавать интуитивно понятные и функциональные пользовательские интерфейсы, реализовывать сложные визуализации данных и разрабатывать динамические веб-приложения, которые реагируют на действия пользователя. Особое внимание будет уделено принципам реактивного программирования, характерным для Shiny, что позволит создавать приложения, в которых изменения в одних элементах интерфейса автоматически отражаются на других.

Помимо изучения фундаментальных концепций, курс охватит такие важные аспекты, как оптимизация производительности приложений, обеспечение их безопасности и масштабируемости. Вы научитесь интегрировать Shiny с другими инструментами и технологиями, что позволит создавать комплексные решения для анализа данных и визуализации результатов.

В результате прохождения курса вы получите не только теоретические знания, но и практические навыки, необходимые для создания профессиональных веб-приложений. Вы сможете применять полученные знания для решения широкого круга задач, от создания интерактивных дашбордов для бизнеса до разработки научных приложений. В современном мире данных визуализация и интерактивность играют ключевую роль в эффективной коммуникации результатов анализа. Традиционные статические отчеты все чаще уступают место динамичным веб-приложениям, позволяющим пользователям исследовать данные самостоятельно, изменять параметры и получать мгновенные результаты. Библиотека Shiny, интегрированная в экосистему R, предоставляет мощный инструмент для создания таких приложений, позволяя превращать статические R-скрипты в интерактивные веб-интерфейсы.

Глава 1
ЗНАКОМСТВО С SHINY

Shiny – это библиотека для языка программирования R, разработанная компанией RStudio. Она предоставляет возможность создавать интерактивные веб-приложения и динамические отчеты непосредственно на языке R. Данные приложения могут взаимодействовать с данными, отображать графики и таблицы, а также выполнять аналитические вычисления, все это с использованием синтаксиса R.

Shiny позволяет создавать интерфейсы для взаимодействия с данными и аналитическими моделями, не требуя знаний веб-разработки. Библиотека основана на концепции реактивного программирования, что позволяет приложениям Shiny автоматически обновляться при изменении входных данных.

Shiny является мощным инструментом для аналитиков данных и специалистов по анализу данных, позволяя им создавать интерактивные отчеты и приложения для демонстрации результатов исследований и анализа данных.

Основные принципы работы с Shiny:

– **user interface:** в Shiny пользовательский интерфейс определяется с помощью специального синтаксиса в R. Можно создавать различные элементы управления, такие как кнопки, текстовые поля, графики и таблицы, чтобы взаимодействовать с данными;

– **server function:** в Shiny каждое приложение имеет функцию сервера, которая содержит код R для обработки входных данных, выполнения анализа и отображения результатов на пользовательском интерфейсе. Функция реагирует на действия пользователя и обновляет приложение в реальном времени;

– **reactivity:** одной из ключевых концепций Shiny является реактивное программирование. Это означает, что приложение автоматически обновляется при изменении входных данных или параметров, без необходимости перезагрузки страницы. Это добавляет приложениям больше интерактивности и динамичности;

– **widgets:** Shiny предоставляет множество различных виджетов и элементов управления, помогающих взаимодействовать с данными, например, выпадающие списки, чекбоксы, слайдеры и многое другое. Приложения становятся более гибкими и удобными для пользователей;

– **Graphs and data visualization:** с помощью Shiny есть возможность легко интегрировать графики и визуализации данных, созданные

с помощью пакетов вроде ggplot2 или plotly. Это позволяет пользователям взаимодействовать с графиками, изменять параметры и видеть результаты в реальном времени.

Shiny открывает перед аналитиками данных и специалистами по анализу данных возможность создавать интерактивные приложения для демонстрации результатов своей работы, облегчая коммуникацию и понимание данных для широкой аудитории.

Использование библиотеки Shiny в R имеет несколько преимуществ и применений, приведем некоторые из них:

1. **Интерактивные визуализации данных:** Shiny позволяет создавать интерактивные графики и визуализации данных, которые пользователи могут настраивать, масштабировать, фильтровать и исследовать в реальном времени. Это делает процесс анализа данных более динамичным и позволяет пользователям получать более глубокое понимание данных;

2. **Интерактивные отчеты и дашборды:** Shiny дает возможность формировать интерактивные отчеты и дашборды, позволяющие пользователям взаимодействовать с данными, менять параметры, просматривать различные виды аналитики и видеть результаты в реальном времени. Процесс принятия решений становится более информативным и удобным;

3. **Пользовательские интерфейсы для аналитических моделей:** еще одно преимущество Shiny – создание пользовательских интерфейсов для аналитических моделей, что делает их доступными для широкой аудитории. Пользователи могут легко вводить данные, настраивать параметры моделей и видеть результаты, не обладая навыками программирования;

4. **Обучение и обучающие материалы:** Shiny можно использовать для разработки интерактивных обучающих материалов, учебных задач и упражнений. Благодаря этому обучение более увлекательное, интересное, эффективное, а студенты могут непосредственно взаимодействовать с данными и результатами анализа.

Несколько примеров различных областей, где использовалась библиотека Shiny для создания интерактивных веб-приложений на языке программирования R:

1. **Финансовый анализ:** компании и финансовые аналитики используют Shiny для создания интерактивных дашбордов и отчетов для анализа финансовых данных, прогнозирования трендов, моделирования портфеля и мониторинга рыночной активности;

2. **Медицинская статистика:** интерактивные приложения для анализа клинических данных, визуализации результатов исследований и моделирования заболеваний. Как результат, врачи и исследователи лучше понимают данные и принимают информированные решения;

3. **Образовательные проекты:** как уже упоминалось выше, Shiny может создавать интерактивные учебные материалы и учебные задачи. Студенты взаимодействуют с данными, проводят анализ и видят результаты в реальном времени, что делает процесс обучения более эффективным;

4. **Маркетинговый анализ:** маркетологи используют Shiny для анализа маркетинговых кампаний, изучения поведения потребителей, прогнозирования продаж и оптимизации рекламных стратегий;

5. **Биоинформатика и геномика:** в данной области Shiny применяется для анализа генетических данных, визуализации геномов, исследования биологических процессов и медицинской диагностики. Это позволяет исследователям и генетикам лучше понимать генетические данные и разрабатывать индивидуализированные подходы к лечению.

Это лишь несколько примеров, где библиотека Shiny была успешно применена. В каждом из них Shiny помогла улучшить анализ данных, визуализацию результатов и коммуникацию, что положительно повлияло на эффективность работы.

Глава 2
ОСНОВЫ ЯЗЫКА R ДЛЯ РАБОТЫ С SHINY

2.1. Введение в язык R

R – это интерпретируемый язык программирования, широко использующийся в статистике, анализе данных, научных исследованиях, машинном обучении и других областях. R предоставляет мощные инструменты для работы с данными, включая встроенные структуры данных (например, векторы, матрицы, списки), операторы и функции для выполнения различных операций. Кроме того, R обладает богатым набором пакетов, которые расширяют его функциональность и позволяют выполнять различные задачи, такие как, визуализация данных, статистический анализ, машинное обучение.

Основные концепции программирования на языке R:

1. **Векторизация:** один из ключевых принципов языка R – векторизация, заключающаяся в том, что операции в R могут быть применены ко всему вектору данных сразу, без необходимости использования циклов. Например, если имеется вектор чисел, можно применить математическую операцию ко всем элементам вектора одновременно;

2. **Функциональное программирование:** R поддерживает функциональное программирование, которое означает, что функции являются объектами первого класса, их можно передавать в качестве аргументов другим функциям, возвращать из функций и присваивать переменным. Это делает R гибким и мощным инструментом для работы с данными;

3. **Реактивное программирование:** как уже отмечалось, библиотека Shiny для R основана на концепции реактивного программирования, которое позволяет создавать интерактивные веб-приложения, автоматически обновляющиеся при изменении входных данных. Как следствие, приложения более динамичны и удобны для пользователей;

4. **Гибкость и мощность:** R предоставляет широкий спектр инструментов для взаимодействия с данными, в том числе встроенные структуры данных (векторы, матрицы, списки), возможности визуализации данных, статистические функции, аналитические возможности и многое другое. Благодаря этому R – популярный выбор для анализа данных и статистики;

5. **Сообщество и пакеты:** R имеет активное сообщество пользователей и разработчиков, которые создают и поддерживают тысячи пакетов для различных задач. Пакеты значительно расширяют

функциональность R и позволяют выполнять разнообразные задачи, начиная от визуализации данных и статистического анализа до машинного обучения и глубокого анализа данных.

Для установки и настройки библиотеки Shiny вам понадобится R и RStudio. Ниже представлена пошаговая инструкция по установке и настройке Shiny:

1. **Установка R:** сначала необходимо установить язык программирования R с официального сайта https://cran.r-project.org/ и следовать инструкциям для конкретной операционной системы;

2. **Установка RStudio:** после установки R необходимо установить RStudio с официального сайта https://www.rstudio.com/products/rstudio/download/. RStudio предоставляет удобную среду разработки для работы с R и Shiny;

3. **Установка пакета Shiny:** запустить RStudio и ввести следующую команду в консоли R для установки пакета Shiny: *R install.packages(“shiny”)*;

4. **Создание приложения Shiny:** создать новый скрипт в RStudio и ввести простейший пример приложения Shiny:

```
library(shiny)
ui <- fluidPage( titlePanel("Пример приложения Shiny"),
sidebarLayout( sidebarPanel( sliderInput("obs",
"Число наблюдений:", min = 1, max = 1000, value = 500) ),
mainPanel( plotOutput("plot") ) ) )
server <- function(input, output)
{ output$plot <- renderPlot({ hist(rnorm(input$obs)) }) }
shinyApp(ui = ui, server = server)
```

5. **Запуск приложения Shiny:** для запуска приложения необходимо нажать кнопку “Run App” в RStudio. Появится интерактивное приложение с гистограммой, которая меняется в зависимости от выбранного числа наблюдений;

6. **Настройка и доработка приложения:** есть возможность настроить и доработать приложение Shiny, добавляя новые элементы управления, графики, таблицы и другие компоненты для взаимодействия с данными.

Теперь, имея базовое представление об установке и настройке библиотеки Shiny в RStudio, можно начать создавать интерактивные веб-приложения на языке программирования R.

Базовое приложение Shiny

Код представляет собой простой пример приложения Shiny на языке программирования R. Разберем его по частям.

1. **Установка и загрузка библиотеки Shiny:**

```
library(shiny)
```

Данная команда загружает библиотеку Shiny, необходимую для создания интерактивных веб-приложений.

2. **Определение пользовательского интерфейса**

```
ui <- fluidPage( titlePanel("Пример приложения Shiny"),
sidebarLayout( sidebarPanel(
sliderInput("obs", "Число наблюдений:", min = 1,
max = 1000, value = 500) ),
mainPanel( plotOutput("plot")
) ) )
```

Этот код определяет пользовательский интерфейс приложения. В данном случае мы создаем страницу с заголовком и разделением на боковую панель и основную панель. На боковой панели есть слайдер для выбора числа наблюдений, на основе которого будет строиться гистограмма. Основная панель содержит график, отображающий гистограмму.

3. **Определение серверной функции: R**

```
server <- function(input, output) {
output$plot <- renderPlot({
hist(rnorm(input$obs))
}) }
```

Этот код определяет серверную функцию, которая будет обрабатывать входные данные и генерировать график. В данном случае мы используем функцию hist для построения гистограммы случайной выборки из нормального распределения с числом наблюдений, заданным пользователем через слайдер.

4. **Запуск приложения Shiny:**

```
shinyApp(ui = ui, server = server)
```

Данная команда запускает приложение Shiny, объединяя пользовательский интерфейс и серверную функцию. После запуска приложения увидим интерактивное веб-приложение с гистограммой,

меняющуюся в зависимости от выбранного числа наблюдений. Этот пример демонстрирует основные концепции Shiny: пользовательский интерфейс, реактивное программирование и интерактивные возможности.

RStudio – это популярная интегрированная среда разработки (IDE) для языка R, представляющая удобный и мощный интерфейс для работы с кодом R. RStudio состоит из нескольких основных панелей:

– **Console:** в этой панели можно выполнять команды R построчно и получать результаты;

– **Script Editor:** здесь можно писать и редактировать свои скрипты на R;

– **Environment/History:** в этой панели отображаются переменные вашего рабочего пространства и история выполненных команд;

– **Plots/Files/Packages/Help:** эти панели предоставляют доступ к графикам, файлам, пакетам и справочной информации, соответственно.

Базовые команды для работы в RStudio:

1. Для установки пакетов в RStudio используйте функцию install.packages(“package_name”);
2. Для загрузки пакетов в RStudio используйте функцию library(package_name);
3. Для запуска скрипта в RStudio нажмите кнопку “Run” или используйте сочетание клавиш “Ctrl + Enter”;
4. Для сохранения скрипта в RStudio используйте комбинацию клавиш “Ctrl + S” или выберите “File” → “Save As”;
5. Для управления рабочим пространством и пакетами в RStudio используйте соответствующие панели.

2.2. Переменные, векторы, списки

2.2.1. Переменные

Переменные в R используются для хранения данных и объектов. Они могут содержать различные типы данных, такие как числа, строки, логические значения и др.

Для создания переменной в R используется оператор присваивания `<-` или =. Например, чтобы создать переменную x и присвоить ей значение 10, можно написать следующий код: `x <- 10`. Также можно использовать оператор =, который имеет тот же смысл, что и `<-`.

В языке программирования R существует несколько видов переменных, содержащих различные типы данных. Основные виды переменных в R представлены ниже.

1. **Числовые переменные (numeric):** этот тип переменных используется для хранения числовых значений, как целых, так и с плавающей точкой. Например,

```
x <- 10 y <- 3.14
```

2. **Целочисленные переменные (integer):** в R есть отдельный тип для хранения именно целочисленных значений. Для создания целочисленной переменной используется функция as.integer(). Пример кода:

```
z <- as.integer(5)
```

3. **Логические переменные (logical):** этот тип переменных может принимать два значения: TRUE или FALSE. Код может выглядеть следующим образом:

```
is_true <- TRUE is_false <- FALSE
```

4. **Строковые переменные (character):** для хранения текстовой информации используются строковые переменные. Например,

```
name <- "Alice" message <- "Hello, World!"
```

5. **Факторы (factor):** факторы используются для представления категориальных данных, к примеру,

```
gender <- factor(c("Male", "Female", "Male", "Female"))
```

6. **Списки (list):** списки позволяют хранить различные типы данных в одном объекте. Один из вариантов использования данного вида переменной:

```
my_list <- list(number = 10, text = "Hello",
values = c(1, 2, 3))
```

Каждый тип переменной имеет свои особенности и применяется в различных сценариях программирования и анализа данных.

Создадим переменные a и b, присвоим им значения 5 и 7 и выведем их сумму на экран с помощью функции print():

```
a <- 5
b <- 7
sum <- a + b print(sum)
```

Подобным образом можно создавать переменные, присваивать им значения и выполнять различные операции с ними в языке программирования R.

2.2.2. Векторы

Работа с векторами является одним из важных аспектов языка программирования R. *Вектор* представляет собой упорядоченный набор значений одного типа данных, который может быть числовым, символьным, логическим и т. д. В R векторы могут быть одномерными (содержащими только одну строку или столбец) или многомерными (содержащими несколько строк и столбцов).

Основные характеристики векторов в R:

– все элементы вектора должны быть одного и того же типа данных;

– векторы могут быть созданы с помощью функции c() (combine) или оператора **:** для создания последовательностей чисел;

– для доступа к элементам вектора используются квадратные скобки [];

– векторы в R могут быть использованы для выполнения различных операций, таких как сложение, вычитание, умножение, деление и др.

Пример создания вектора чисел от 1 до 5 с использованием функции c():

```
numbers <- c(1, 2, 3, 4, 5)
```

Векторы широко используются в R для хранения и манипуляции данными, а также для выполнения различных операций, включая статистический анализ, визуализацию данных, машинное обучение и другие задачи.

Рассмотрим функцию c(), которая объединяет элементы вектора.

Пример 1. Создание вектора чисел:

```
numbers <- c(1, 2, 3, 4, 5) print(numbers)
```

Пример 2. Создание вектора строк:

```
fruits <- c("apple", "banana", "orange") print(fruits)
```

Пример 3. Создание вектора логических значений:

```
logical_values <- c(TRUE, FALSE, TRUE) print(logical_values)
```

Объяснение кода: в первом примере создается вектор чисел от 1 до 5. Во втором примере создается вектор строк с названиями фруктов. В третьем примере создается вектор логических значений TRUE и FALSE. Функция print() используется для вывода созданных векторов на экран.

После выполнения этого кода на экране будет выведено содержимое каждого созданного вектора. Этот пример поможет лучше понять процесс создания векторов с использованием функции c() в R.

Можно также создавать векторы с помощью оператора. Оператор **:** позволяет создавать последовательности чисел или символов в векторе. Синтаксис оператора **:** следующий: начальное_значение **:** конечное_значение. При этом будут созданы все числа или символы от начального до конечного значения включительно.

Пример 1. Создание вектора чисел от 1 до 5:

```
numbers <- 1:5 print(numbers)
```

Пример 2. Создание вектора четных чисел от 2 до 10:

```
even_numbers <- 2:10 print(even_numbers)
```

Пример 3. Создание вектора букв от A до E:

```
letters <- "A":"E" print(letters)
```

Пример 4. Создадим вектор чисел от 1 до 10 с шагом 2:

```
numbers_step <- 1:10:2 print(numbers_step)
```

Оператор **:** позволяет создавать последовательности чисел или символов в векторе без явного перечисления каждого элемента. Это удобно и эффективно при работе с большими последовательностями.

Индексация элементов вектора в языке программирования R позволяет получать доступ к конкретным элементам вектора по их позиции. Индексация в R начинается с 1. Для доступа к элементам используются квадратные скобки [].

Рассмотрим примеры индексации элементов вектора:

```
# Создание вектора
numbers <- c(10, 20, 30, 40, 50)
# Получение элемента по индексу 3
third_element <- numbers[3] print(third_element)
# Получение нескольких элементов по индексам 2 и 4
elements_2_4 <- numbers[c(2, 4)] print(elements_2_4)
# Получение всех элементов, кроме первого
all_except_first <- numbers[-1] print(all_except_first)
```

Объяснение кода: в этом примере создается вектор numbers с элементами от 10 до 50. numbers [3] возвращает третий элемент вектора, который равен 30. numbers [c(2, 4)] возвращает элементы вектора с индексами 2 и 4, то есть 20 и 40. numbers [-1] возвращает весь вектор, кроме первого элемента, то есть 20, 30, 40, 50.

Дополнительные операции с индексацией:

– можно использовать диапазоны индексов для выбора нескольких последововательных элементов. Например, numbers[2:4] вернет элементы с индексами от 2 до 4;

– можно использовать логическую индексацию для выбора элементов, удовлетворяющих определенному условию;

– индексы могут быть переменными или результатами других операций.

Индексация элементов вектора позволяет удобно выбирать и работать с конкретными значениями вектора в языке R.

Также в языке программирования R можно выполнять различные операции с векторами, такие как сложение, вычитание, умножение, деление и другие. Рассмотрим основные операции с векторами в R.

1. **Сложение векторов:**

```
vector1 <- c(1, 2, 3) vector2 <- c(4, 5, 6)
result <- vector1 + vector2 print(result)
```

Результатом выполнения этого кода будет вектор, в котором каждый элемент равен сумме соответствующих элементов из vector1 и vector2.

2. **Вычитание векторов:**

```
result <- vector1 - vector2 print(result)
```

В этом случае каждый элемент результирующего вектора будет равен разности соответствующих элементов из vector1 и vector2.

3. **Умножение векторов:**

```
result <- vector1 * vector2 print(result)
```

Каждый элемент результирующего вектора будет равен произведению соответствующих элементов из vector1 и vector2.

4. **Деление векторов:**

```
result <- vector1 / vector2 print(result)
```

Каждый элемент результирующего вектора будет равен частному соответствующих элементов из vector1 и vector2.

В R также можно выполнять другие операции с векторами, например, возведение в степень, взятие остатка от деления и другие математические операции.

Важно отметить, что для выполнения операций с векторами в R необходимо, чтобы векторы имели одинаковую длину; если же векторы имеют разную длину, R выполнит операцию по принципу recycling, то есть повторит элементы более короткого вектора для соответствия длине более длинного вектора; векторы в R могут содержать различные типы данных, и операции будут выполняться в зависимости от типов данных элементов.

Выполнение различных математических операций над элементами векторов очень удобно и эффективно при анализе данных и выполнении вычислений.

В языке программирования R существует множество встроенных функций, которые облегчают работу с векторами. Функции выполняют различные операции над векторами, позволяют проводить анализ данных, вычисления.

Некоторые из основных функций для работы с векторами в R:

1. **sum():** используется для вычисления суммы всех элементов вектора. Например,

```
numbers <- c(1, 2, 3, 4, 5) total <- sum(numbers) print(total)
```

2. **mean():** вычисляет среднее значение элементов вектора. К примеру,

```
average <- mean(numbers) print(average)
```

3. **length():** возвращает количество элементов в векторе.
Код может выглядеть следующим образом:

```
vector_length <- length(numbers) print(vector_length)
```

4. **min() и max():** используются для нахождения минимального и максимального значения в векторе, соответственно. Например,

```
min_value <- min(numbers) max_value <- max(numbers)
print(min_value) print(max_value)
```

5. **sort():** сортирует элементы вектора по возрастанию. Например,

```
sorted_vector <- sort(numbers) print(sorted_vector)
```

Это лишь несколько примеров функций для работы с векторами в R. Есть множество других функций, которые могут быть полезны при работе с данными и векторами. Понимание и использование этих функций очень важно для эффективной работы.

2.2.3. Списки

Список (list) в языке программирования – это структура данных, которая позволяет хранить коллекцию элементов различных типов внутри одного объекта. Элементы списка могут быть любого типа данных, включая числа, строки, логические значения, векторы, другие списки и даже функции.

В R список создается с помощью функции list(). Каждый элемент списка имеет уникальное имя (ключ), которое позволяет обращаться к нему. Список в R может содержать элементы различной длины и типов данных.

Пример создания списка в R:

```
# Создание списка
my_list <- list(name = "Alice", age = 30,
is_student = TRUE, grades = c(85, 90, 75))
# Вывод списка
print(my_list)
```

В этом примере создается список my_list, который содержит четыре элемента: строку “Alice”, число 30, логическое значение TRUE и вектор оценок.

Списки в R являются удобной структурой данных для хранения и организации разнородной информации. Они широко используются в анализе данных, программировании, статистике и других областях для структурирования данных, передачи информации между функциями, создания сложных структур.

Для доступа к элементам списка в языке программирования R используются различные методы, включая использование двойных квадратных скобок [[ ]], оператора $ и функции [[ ]]. Рассмотрим эти методы доступа подробнее.

1. **Двойные квадратные скобки [[ ]]:** используются для доступа к элементу списка по его индексу или имени. Если элемент списка имеет имя, его можно получить по имени, а если элементы нумерованы, то можно обратиться к элементу по индексу.

Пример:

```
# Создание списка
my_list <- list(name = "Alice", age = 30,
is_student = TRUE, grades = c(85, 90, 75))
# Доступ к элементу по имени
element_name <- my_list[["name"]]
# Доступ к элементу по индексу
element_age <- my_list[[2]]
print(element_name)
print(element_age)
```

2. **Оператор $:** позволяет получить доступ к элементам списка также по их именам. Метод удобен, когда элементы списка имеют имена.

Пример:

```
# Доступ к элементу по имени с использованием
оператора
$ element_name <- my_list$name
print(element_name)
```

3. **Функция [[ ]]:** также используется для доступа к элементам списка по их индексу или имени. Она может быть полезна в тех случаях, когда имя элемента хранится в переменной.

Пример:

```
# Использование функции [[ ]] для доступа
к элементу по имени
name_variable <- "name"
element_name <- my_list[[name_variable]]
print(element_name)
```

Понимание различных методов доступа к элементам списка в R позволит эффективно работать с данными в списках и извлекать нужную информацию для дальнейшей обработки.

Для добавления новых элементов в список в языке программирования R можно использовать различные методы, такие как присваивание по новому ключу или добавление элемента с помощью функции append(). Рассмотрим методы подробнее.

Присваивание нового элемента по новому ключу

Для добавления нового элемента в список по новому ключу можно использовать простое присваивание. Новый элемент будет добавлен в список с указанным ключом.

Пример:

```
# Создание списка
my_list <- list(name = "Alice", age = 30)
# Добавление нового элемента по новому ключу
my_list$city <- "New York" print(my_list)
```

В этом примере элемент “city” со значением “New York” добавляется в список my_list.

Использование функции append()

Функция append() позволяет добавлять новые элементы в список, она принимает список и новый элемент, который нужно добавить.

Пример:

```
# Создание списка
my_list <- list(name = "Alice", age = 30)
# Добавление нового элемента с использованием
функции append()
my_list <- append(my_list, list(city = "New York"))
print(my_list)
```

В этом примере функция append() используется для добавления нового элемента “city” со значением “New York” в список my_list.

При использовании append() необходимо присваивать результат обратно переменной списка.

Важно отметить, что при добавлении новых элементов в список необходимо убедиться, что переменная списка обновляется с учетом новых элементов; можно добавлять элементы различных типов данных в список, включая числа, строки, логические значения, векторы и другие списки; при добавлении элементов по новому ключу нужно удостовериться, что ключ уникален в списке.

Теперь в нашем арсенале два метода для добавления новых элементов в список в R, что позволит эффективно расширять и обновлять списки с данными по мере необходимости.

Для итерации по элементам списка в языке программирования R можно использовать циклы for или функцию lapply().

Итерация с помощью цикла for

Цикл for используют для перебора всех элементов списка. В каждой итерации цикла можно обращаться к текущему элементу списка. Пример итерации по элементам списка с помощью цикла for:

```
# Создание списка
my_list <- list(name = "Alice", age = 30,
is_student = TRUE, grades = c(85, 90, 75))
# Итерация по элементам списка с помощью цикла for
for (element in my_list) {
print(element)
}
```

В примере цикл for проходит по всем элементам списка my_list и выводит каждый элемент на экран.

Использование функции lapply()

Функция lapply() применяется к каждому элементу списка и возвращает результат в виде списка.

Пример использования lapply() для итерации по элементам списка:

```
# Создание функции для вывода элементов на экран
print_element <- function(element) {
print(element)
}
# Применение функции к каждому элементу списка
с помощью lapply()
result <- lapply(my_list, print_element)
```

Здесь функция print_element применяется к каждому элементу списка my_list с помощью lapply(), что позволяет вывести каждый элемент на экран.

Важно отметить, что при итерации по элементам списка необходимо убедиться, что обрабатывается каждый элемент соответствующим образом в зависимости от его типа данных; использовать циклы или функции для итерации по элементам списка в зависимости от ваших потребностей и предпочтений; итерация по элементам списка позволяет обрабатывать данные в списках, выполнять операции и анализировать информацию.

Работа со вложенными списками в языке программирования R позволяет структурировать данные более сложным образом, создавая иерархию элементов внутри списка. Вложенные списки могут содержать другие списки в качестве элементов, что делает их более гибкими для хранения и организации данных. Изучим работу со вложенными списками на примерах.

Создание вложенных списков происходит путем включения одного списка в качестве элемента другого списка.

Пример создания вложенного списка:

```
# Создание вложенного списка
nested_list <- list(person = list(name = "Alice",
age = 30),
grades = c(85, 90, 75))
print(nested_list)
```

Здесь формируется вложенный список nested_list, который содержит два элемента: элемент “person”, который сам является списком с элементами “name” и “age”, и элемент “grades”, являющийся вектором оценок.

Для доступа к элементам вложенных списков можно использовать комбинацию методов доступа к элементам списка, таких как двойные квадратные скобки [[ ]] и оператор $.

Пример доступа к элементам вложенного списка:

```
# Доступ к элементам вложенного списка
name <- nested_list$person$name
grade <- nested_list$grades[1]
print(name) print(grade)
```

В данном примере мы обращаемся к элементам вложенного списка nested_list по их ключам или индексам.

Можно использовать циклы или функции для итерации по элементам вложенных списков. Пример итерации по вложенным спискам:

```
# Итерация по вложенным спискам
for (element in nested_list) {
if (is.list(element))
{ for (sub_element in element)
{ print(sub_element) }
}
else {
print(element) }
}
```

В этом примере используется цикл for для итерации по элементам вложенного списка nested_list, позволяя обрабатывать как внутренние списки, так и другие элементы.

Работа со вложенными списками в R позволяет более гибко структурировать данные и организовывать информацию в сложных структурах.

В R можно применять различные встроенные функции к спискам для анализа данных, обработки информации и выполнения других операций. Списки в R представляют собой мощный инструмент для организации и хранения разнородных данных. Они широко используются в анализе данных, программировании, статистике и других областях.

2.3. Условные операторы

Условный оператор – это конструкция, позволяющая выполнять определенные действия в зависимости от соблюдения определенного условия. Условный оператор позволяет программе принимать решения и изменять ход выполнения кода на основе различных условий.

В языке программирования R условный оператор может быть реализован с помощью конструкций if-else, if-else if-else и switch. Изучим каждый из них.

1. **Оператор if-else:** данный оператор проверяет условие. Если условие истинно, выполняется блок кода внутри if, если ложно – блок кода внутри else.

2. **Оператор if-else if-else:** позволяет проверить несколько условий последовательно. Если первое условие истинно, выполняется соответствующий блок кода. Если ни одно из условий не истинно, выполняется блок кода в else.

3. **Оператор switch:** выбирает действие для выполнения на основе значения выражения. Он сравнивает значение выражения с различными вариантами и выполняет соответствующий блок кода.

Пример использования условного оператора if-else в R:

```
age <- 25
if (age >= 18) {
  print("Вам можно голосовать")
} else {
  print("Вы не можете голосовать")
}
```

Здесь, если возраст age больше или равен 18, будет выведено сообщение «Вам можно голосовать», иначе – «Вы не можете голосовать».

Пример использования условного оператора if-else if-else в R:

```
score <- 85
if (score > 90) {
  print("Отлично")
} else if (score > 80) {
  print("Хорошо")
} else {
  print("Нужно подтянуть знания")
}
```

В данном примере: если оценка score больше 90, будет выведено сообщение «Отлично»; если оценка находится в диапазоне от 80 до 90 (не включая 90), будет выведено сообщение «Хорошо»; если оценка находится в диапазоне от 70 до 80 (не включая 80), будет выведено сообщение «Удовлетворительно»; если ни одно из условий не выполняется, будет выведено сообщение «Нужно подтянуть знания».

Условный оператор if-else if-else позволяет проверять несколько условий последовательно и выполнять соответствующий блок кода для первого истинного условия. Это удобно для создания более сложных логических проверок в коде.

В следующем примере будем использовать оператор switch для определения дня недели на основе его номера. Это может выглядеть так:

```
day_number <- 3
day <- switch(day_number, "Понедельник", "Вторник", "Среда",
"Четверг", "Пятница", "Суббота", "Воскресенье")
print(day)
```

– переменная day_number содержит номер дня недели;

– оператор switch сравнивает day_number с различными вариантами и возвращает соответствующий день недели;

– если day_number равен 3, будет возвращено «Среда»;

– если day_number не соответствует ни одному из вариантов, оператор switch вернет значение по умолчанию.

Условный оператор switch в R позволяет выбирать действие для выполнения на основе значения выражения. Он удобен в случаях, когда необходимо выбирать действие из нескольких вариантов на основе значения одной переменной.

Условные операторы являются важным инструментом в программировании, поскольку позволяют создавать гибкие и адаптивные программы, которые могут принимать решения на основе входных данных. Понимание и использование условных операторов помогает программистам писать более эффективный и функциональный код.

2.4. Циклы

Циклы в программировании – это конструкции, выполняющие повторяющиеся действия или операции несколько раз. Циклы позволяют автоматизировать выполнение однотипных задач без необходимости повторного написания кода. В языке программирования R существуют различные типы циклов, такие как цикл for, цикл while и цикл repeat. Познакомимся с каждым из них.

1. **Цикл for** выполняет определенное количество итераций, перебирая элементы в последовательности или векторе. Синтаксис цикла for в R:

```
for (переменная in последовательность) {
  # Блок кода, который будет выполнен на каждой
итерации
}
Пример использования цикла for:

# Пример использования цикла for для вывода чисел
от 1 до 5
for (i in 1:5) {
  print(i)
}
```

2. **Цикл while** выполняется до тех пор, пока условие истинно. Синтаксис цикла while в R:

```
while (условие) {
  # Блок кода, который будет выполняться, пока
условие истинно
}
Пример использования цикла while:

# Пример использования цикла while для вывода чисел
от 1 до 5
i <- 1
while (i <= 5) {
  print(i)
  i <- i + 1
}
```

3. **Цикл repeat** выполняет блок кода до тех пор, пока не будет явно прерван с помощью оператора break. Синтаксис цикла repeat в R:

```
repeat {
  # Блок кода, который будет выполняться повторно
  if (условие_выхода) {
    break
  }
}
Пример использования цикла repeat:

# Пример использования цикла repeat для вывода чисел
от 1 до 5
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 5) {
    break
  }
}
```

Циклы в программировании позволяют эффективно обрабатывать повторяющиеся задачи, обходить элементы в структурах данных, итерироваться по данным и многие другие операции. Понимание и использование циклов помогает программистам писать более компактный код.

2.5. Функции

Функции в программировании – это блоки кода, которые могут быть вызваны для выполнения определенной задачи или операции. Функции позволяют структурировать код, изолировать логику выполнения определенных действий и повторно использовать код. В языке программирования R функции могут быть встроенными (встроенными в язык) или пользовательскими (определяемыми пользователем).

Параметры функций – это входные данные, которые функция принимает для выполнения операций. В R параметры функций указываются в скобках после имени функции. Функция может иметь любое количество параметров.

Возвращаемое значение – это результат, который функция возвращает после выполнения операций. В R используется ключевое слово return для возврата значения из функции. Если return не указан, функция вернет последнее вычисленное значение.

Изучим основные аспекты функций в R.

Функция в R определяется с помощью ключевого слова function и может принимать аргументы (входные параметры) и возвращать результат (выходное значение).

Пример определения функции в R:

```
# Определение функции в R
my_function <- function(x, y) {
result <- x + y return(result)
}
```

В указанном примере функция my_function принимает два аргумента x и y, складывает их и возвращает результат.

После определения функции ее можно вызвать для выполнения заданной задачи.

Пример вызова функции в R:

```
# Вызов функции и вывод результата
result <- my_function(3, 5) print(result)
```

Здесь функция my_function вызывается с аргументами 3 и 5, результат складывания которых будет сохранен в переменной result и выведен на экран.

В R также существуют встроенные функции, которые предоставляются в стандартной библиотеке языка и позволяют выполнять различные операции.

Пример использования встроенной функции mean для вычисления среднего значения:

```
# Пример использования встроенной функции
mean() values <- c(2, 4, 6, 8, 10)
average <- mean(values)
print(average)
```

Фунция mean принимает вектор values в качестве аргумента и возвращает среднее значение этого вектора.

Преимущества использования функций:

– структурирование кода и изоляция логики выполнения задач;

– использование кода без необходимости его повторного написания;

– улучшение читаемости кода и облегчение его отладки и сопровождения.

Примеры использования пользовательских функций.

Пример 1. Функция для вычисления среднего значения списка чисел:

```
# Создание пользовательской функции для вычисления
среднего значения
mean_function <- function(numbers) {
  total <- sum(numbers)
  count <- length(numbers)
  mean_value <- total / count
  return(mean_value)
}

# Вызов функции с передачей списка чисел
values <- c(10, 15, 20, 25, 30)
result <- mean_function(values)
print(result) # Выведет: 20
```

Пример 2. Функция для определения четности числа с использованием условного оператора if-else:

```
# Создание пользовательской функции для определения
четности числа
check_even_odd <- function(number) {
  if (number %% 2 == 0) {
    return("Число четное")
  } else {
    return("Число нечетное")
  }
}

# Вызов функции с передачей числа
result <- check_even_odd(7)
print(result) # Выведет: "Число нечетное"
```

Пример 3. Функция для объединения двух строк:

```
# Создание пользовательской функции для объединения
строк
combine_strings <- function(str1, str2) {
  combined <- paste(str1, str2, sep = " ")
  return(combined)
}

# Вызов функции с передачей двух строк
result <- combine_strings("Hello", "World")
print(result) # Выведет: "Hello World"
```

Создание пользовательских функций, работа с параметрами и возвращаемыми значениями являются важными аспектами программирования в R. Они позволяют структурировать код, повторно использовать логику и улучшить организацию программы. Функции помогают создавать более эффективные и организованные программы.

Глава 3
СОЗДАНИЕ ПРОСТОГО ПРИЛОЖЕНИЯ SHINY

UI (User Interface) в библиотеке Shiny в R представляет собой компоненты, определяющие внешний вид и взаимодействие пользовательского интерфейса в веб-приложении. Элементы UI позволяют создавать различные виджеты, кнопки, текстовые поля, графики и другие элементы, которые пользователь может видеть и с которыми может взаимодействовать.

Некоторые распространенные UI элементы, которые можно использовать при создании веб-приложений с помощью Shiny в R:

– **titlePanel:** панель заголовка, отображающая заголовок приложения. Пример: titlePanel (“Мое первое приложение Shiny”);

– **sidebarLayout:** макет с боковой панелью и основной панелью. Позволяет размещать элементы в боковой и основной панелях.

Пример:

```
sidebarLayout(
  sidebarPanel(
    # Здесь размещаются элементы для боковой панели
  ),
  mainPanel(
    # Здесь размещаются элементы для основной панели
  )
)
```

– **actionButton:** кнопка, которую пользователь может нажать для выполнения определенного действия. Пример: actionButton (“my_button”, “Нажми меня!”);

– **textInput:** текстовое поле, в которое пользователь может вводить текст. Пример: textInput (“my_text”, “Введите текст:”);

– **numericInput:** числовое поле, в которое пользователь может вводить числовое значение. Пример: numericInput (“my_number”, “Введите число:”, value = 0);

– **plotOutput:** элемент для вывода графиков или графических изображений. Пример: plotOutput (“my_plot”);

– **textOutput:** элемент для вывода текста или результатов вычислений. Пример: textOutput (“my_text_output”).

Выше представлен лишь небольшой набор UI элементов, которые можно использовать для создания интерактивных веб-приложений

с помощью библиотеки Shiny. Комбинируя различные UI элементы, можно создавать разнообразные пользовательские интерфейсы с возможностью взаимодействия пользователя с приложением.

Server функция в библиотеке Shiny в R отвечает за обработку входных данных от пользовательского интерфейса и управление логикой приложения. Server функция взаимодействует с UI элементами, обрабатывает пользовательский ввод, выполняет вычисления, обновляет данные и выводит результаты на UI.

Пример Server функции для простого приложения Shiny, которое отображает приветственное сообщение и счетчик нажатий кнопки:

```
# Определение Server функции
server <- function(input, output) {
output$message <- renderText(
{ "Добро пожаловать в мое первое приложение Shiny!" })
clickCount <- reactiveVal(0) observeEvent
(input$button,
{ clickCount(clickCount() + 1) })
output$counter <- renderText({
paste("Количество нажатий кнопки:", clickCount())
})
}
```

В данном примере:

– **server** – это функция, которая принимает два аргумента input и output;

– **output$message** – это элемент UI, который отображает приветственное сообщение. Функция renderText используется для обновления текста на UI;

– **clickCount** – это реактивное значение, которое хранит количество нажатий кнопки;

– **observeEvent(input$button, {...})** – это блок кода, реагирующий на событие нажатия кнопки. При каждом нажатии кнопки счетчик увеличивается на 1;

– **output$counter** – это элемент UI, который отображает счетчик нажатий кнопки. Функция renderText используется для обновления текста на UI.

Server функция в Shiny играет ключевую роль в обработке пользовательского ввода, выполнении вычислений и обновлении данных на UI. Она позволяет создавать интерактивные и динамические веб-приложения, которые реагируют на действия пользователя.

Для создания простого приложения с использованием библиотеки Shiny в R нам понадобится определить UI элементы и Server функцию. Создадим приложение, которое будет отображать приветственное сообщение и счетчик нажатий кнопки. Пример кода для такого приложения:

```
# Установка и загрузка пакета Shiny
install.packages("shiny")
library(shiny)

# Определение UI (User Interface) элементов
ui <- fluidPage(
  titlePanel("Простое приложение Shiny"),
  sidebarLayout(
    sidebarPanel(
      actionButton("button", "Нажми меня!")
    ),
    mainPanel(
      textOutput("message"),
      textOutput("counter")
    )
  )
)

# Определение Server функции
server <- function(input, output) {
  output$message <- renderText({
    "Добро пожаловать в мое первое приложение Shiny!"
  })

  clickCount <- reactiveVal(0)

  observeEvent(input$button, {
    clickCount(clickCount() + 1)
  })

  output$counter <- renderText({
    paste("Количество нажатий кнопки:", clickCount())
  })
}

# Запуск приложения
shinyApp(ui = ui, server = server)
```

В этом приложении UI элементы определяются с помощью функции fluidPage, где создаем заголовок, боковую панель с кнопкой «Нажми меня!» и главную панель для отображения сообщения и счетчика.

Server функция определяется с помощью функции server, где устанавливаем сообщение приветствия и реагируем на нажатие кнопки, увеличивая счетчик.

Затем запускаем приложение с помощью функции shinyApp, передавая UI и Server функции.

Чтобы запустить это приложение, необходимо сохранить в файле с расширением .R и выполнить его в R-студии или другой среде R. После этого появится простое приложение Shiny с приветственным сообщением и счетчиком нажатий кнопки.

Глава 4
РАБОТА С РЕАКТИВНЫМИ ОБЪЕКТАМИ

Реактивные объекты в библиотеке Shiny в R представляют собой специальные объекты, которые автоматически пересчитываются при изменении их зависимостей. Реактивные объекты могут быть использованы для создания интерактивных приложений, в которых результаты автоматически обновляются при изменениях входных данных или пользовательского ввода.

В Shiny существует несколько типов реактивных объектов, таких как реактивные значения (reactive values), реактивные выражения (reactive expressions) и реактивные функции (reactive functions).

Общие принципы работы с реактивными объектами в библиотеке Shiny в R включают в себя понимание создания, использования и обновления реактивных объектов для создания интерактивных веб-приложений.

Основные принципы работы с реактивными объектами в Shiny:

1. **Создание реактивных объектов:**

– для создания реактивных значений используются функции reactiveVal(), reactiveValues(), для реактивных выражений и реактивных функций – reactive();

– реактивные объекты могут хранить данные, выражения или функции, которые автоматически пересчитываются при изменении их зависимостей.

2. **Использование реактивных объектов:**

– в Server функции для обработки пользовательского ввода, выполнения вычислений и обновления данных на пользовательском интерфейсе;

– создание динамических приложений, реагирующих на изменения входных данных или пользовательского ввода.

3. **Зависимости реактивных объектов:** Shiny отслеживает зависимости реактивных объектов и автоматически пересчитывает их при изменениях значений, от которых они зависят. Данный принцип позволяет обновлять результаты вычислений и данные в реальном времени в соответствии с изменениями входных данных.

4. **Обновление реактивных объектов:** реактивные объекты могут быть обновлены явно с помощью функции invalidate() для пересчета значения или выражения вручную. Это может быть полезно в случаях, когда требуется контролировать пересчет реактивных объектов или обновлять их по запросу пользователя.

Рассмотрим основные типы реактивных объектов в Shiny: *реактивные выражения*, *реактивные функции* и *реактивные значения*.

Реактивные значения – это простейший тип реактивных объектов, которые используются для хранения отдельных реактивных переменных:

– **reactiveVal** – используется для хранения одного реактивного значения;

– **reactiveValues** – используется для хранения нескольких реактивных значений, похожих на список или словарь.

Пример с reactiveVal:

```
rv <- reactiveVal(10)
```

В другом месте приложения может использоваться rv(), чтобы получить значение, и rv(newValue), чтобы установить новое значение.

Пример с reactiveValues:

```
values <- reactiveValues(score = 0,
userName = "Guest")
```

Чтобы получить значение, используется values$score. Чтобы установить значение – values$score <- 10.

Реактивные выражения – это выражения, которые автоматически пересчитывают свой результат, когда переменные, от которых они зависят, меняются.

Пример:

```
reactiveSum <- reactive({
  input$num1 + input$num2
})
```

У реактивного выражения есть вызываемый метод для получения его текущего значения, то есть reactiveSum() вернет сумму num1 и num2.

Реактивные функции – это функции, срабатывающие в ответ на изменения в реактивных зависимостях.

Observe используется для непосредственного наблюдения за реактивными значениями и создания побочных эффектов при их изменении. observeEvent работает аналогично, но реагирует только на специфические события вместо всех изменений реактивных объектов.

Пример с observe:

```
observe({
  # Этот код выполнится каждый раз, когда input$num1
или input$num2 изменится.
  output$sum <- renderText({
    paste("The sum is", input$num1 + input$num2)
  })
})
```

Пример с observeEvent:

```
# Этот код выполнится только тогда, когда
пользователь нажимает кнопку (input$goButton)
observeEvent(input$goButton, {
  output$sum <- renderText({
    paste("The sum is", input$num1 + input$num2)
  })
})
```

Рассмотрим отличия между реактивными объектами:

– **реактивные значения** – это переменные, хранящие данные, которые могут изменяться и вызывать реактивные обновления в приложении;

– **реактивные выражения** служат для определения логики или вычислений, которые должны автоматически обновляться при изменении их зависимостей. Реактивные выражения имеют возвращаемое значение, которое можно использовать в других частях Shiny-приложения;

– **реактивные функции** (observe и observeEvent) не возвращают значение, но выполняют действия (например, изменение UI или других сторонних эффектов), когда их реактивные зависимости изменяются.

Пример реактивного приложения, которое принимает два числа от пользователя и выводит их сумму:

```
library(shiny)
 # Определение UI
 ui <- fluidPage( sidebarLayout( sidebarPanel(
 numericInput("number1", "Введите первое число:",
value = 0),
 numericInput("number2", "Введите второе число:",
value = 0) ),
 mainPanel( textOutput("result") ) ) )
 # Определение Server функции
 server <- function(input, output) {
```

```
  # Создание реактивного выражения для вычисления суммы
  sum_result <- reactive({ sum <- input$number1 +
  input$number2 return(sum) })
  # Вывод результата на UI
  output$result <- renderText({ paste("Сумма чисел:",
  sum_result()) }) }
  # Запуск приложения
  shinyApp(ui = ui, server = server)
```

В этом примере, демонстрирующем создание реактивного выражения для вычисления суммы двух чисел, создаем два числовых ввода (numericInput), в которые пользователь может ввести два числа. Далее, создаем реактивное выражение sum_result, которое вычисляет сумму двух чисел, введенных пользователем. Используем renderText для отображения результата суммы на пользовательском интерфейсе. При изменении значений в числовых вводах, реактивное выражение автоматически пересчитывается, обновляя сумму на UI.

Реактивные объекты в библиотеке Shiny в R часто используются в Server функции для обработки пользовательского ввода, выполнения вычислений и обновления данных на пользовательском интерфейсе. Далее приведен пример использования реактивного выражения в Server функции для создания простого приложения, которое вычисляет квадрат введенного числа:

```
R library(shiny)
# Определение UI
ui <- fluidPage( sidebarLayout( sidebarPanel
( numericInput("number", "Введите число:", value = 0) ),
mainPanel( textOutput("result") ) ) )
# Определение Server функции
server <- function(input, output) {
# Создание реактивного выражения для вычисления
квадрата введенного числа
square_result <- reactive({ square <- input$number^2
return(square) })
# Вывод результата на UI
output$result <- renderText({ paste("Квадрат числа:",
square_result()) }) }
# Запуск приложения
shinyApp(ui = ui, server = server)
```

В приведенном примере создаем числовой ввод (numericInput), в который пользователь может ввести число. Далее, создаем реактивное выражение square_result, которое вычисляет квадрат введенного числа. Используем renderText для отображения результата квадрата на пользовательском интерфейсе. При изменении значения в числовом вводе, реактивное выражение автоматически пересчитывается, обновляя квадрат на UI.

Функция invalidate() в библиотеке Shiny в R используется для явного вызова пересчета реактивного выражения. Это действительно может быть полезно, когда требуется обновить результат вычислений в реактивном выражении вручную, а не дожидаться автоматического пересчета при изменении зависимостей.

Пример использования функции invalidate() в контексте реактивного выражения:

```
R library(shiny)
# Определение UI
ui <- fluidPage( sidebarLayout( sidebarPanel
( numericInput("number", "Введите число:", value = 0) ),
mainPanel( textOutput("result"), actionButton
("update_button", "Обновить результат") ) ) )
# Определение Server функции
server <- function(input, output, session) {
square_result <- reactive({ input$number^2
})
output$result <- renderText({ square_result() })
observeEvent(input$update_button, {
# Явный вызов пересчета реактивного выражения
invalidate(square_result) })
}
# Запуск приложения
shinyApp(ui = ui, server = server)
```

Объяснение кода:

– создаем числовой ввод, реактивное выражение для вычисления квадрата введенного числа и текстовый вывод результата;

– добавляем кнопку «Обновить результат», которая при нажатии вызывает функцию invalidate(), пересчитывая реактивное выражение square_result;

– это позволяет явно обновить результат вычислений, даже если значение числового ввода не изменилось.

Теперь создадим приложение Shiny, которое будет обновлять случайное число каждую секунду в реальном времени, при этом будем использовать реактивные объекты и функции для обновления данных динамически.

```
# Определение UI
ui <- fluidPage(
  titlePanel("Обновление данных в реальном времени"),
  mainPanel(
    textOutput("random_number")
  )
)

# Определение Server функции
server <- function(input, output, session) {
  # Создание реактивного значения для хранения
случайного числа
  random_value <- reactiveVal(runif(1))

  # Функция для обновления случайного числа каждую
секунду
  update_random_number <- function() {
    random_value(runif(1))
  }

  # Функция для обновления данных каждую секунду
  observe({
    invalidateLater(1000, session)
    update_random_number()
  })

  # Отображение случайного числа на UI
  output$random_number <- renderText({
    random_value()
  })
}

# Запуск приложения
shinyApp(ui = ui, server = server)
```

В примере создаем UI, которое содержит один текстовый вывод для отображения случайного числа. В Server функции создаем реактивное значение random_value, хранящее случайное число и функцию update_random_number(), которая обновляет случайное число. Далее, используем функцию invalidateLater() внутри observe(), чтобы

обновлять случайное число каждую секунду. Используем renderText() для отображения случайного числа на пользовательском интерфейсе.

При запуске приложения можно увидеть, как случайное число обновляется каждую секунду в реальном времени на пользовательском интерфейсе. Это отличная демонстрация того, как использовать реактивные объекты и функции в библиотеке Shiny для обновления данных в реальном времени.

Реактивные выражения являются мощным инструментом в библиотеке Shiny для разработки интерактивных и динамических веб-приложений, реагирующих на изменения пользовательского ввода и данных. Понимание работы с реактивными выражениями позволяет создавать более сложные и интересные приложения.

Глава 5
ГРАФИКИ И ВИЗУАЛИЗАЦИЯ ДАННЫХ В SHINY

5.1. Введение в ggplot2

В библиотеке Shiny в R можно легко создавать интерактивные графики и визуализации данных, которые автоматически обновляются при изменении пользовательского ввода или данных.

Использование пакета ggplot2 для создания графиков в библиотеке Shiny в R позволяет формировать красивые и высококачественные визуализации данных с возможностью интерактивного взаимодействия. Рассмотрим основные шаги по использованию ggplot2 в Shiny для создания графиков.

1. **Установка и подключение пакета ggplot2:**

– убедиться, что пакет ggplot2 установлен в вашем R-окружении; иначе необходимо выполнить install.packages(“ggplot2”);

– подключить пакет ggplot2 в скрипте Shiny с помощью library(ggplot2).

2. **Создание графика с ggplot2 в Server функции:**

– в Server функции необходимо использовать функции ggplot2 для создания графика на основе данных и пользовательского ввода;

– использовать функцию renderPlot() для отображения графика на пользовательском интерфейсе.

Пакет ggplot2 в R предоставляет широкий набор функций для создания различных типов графиков и визуализаций данных. Ниже представлены некоторые из основных функций ggplot2, которые могут быть использованы для создания графиков.

ggplot(): основная функция для создания объекта ggplot.

```
library(ggplot2)
data <- data.frame(x = c(1, 2, 3, 4, 5), y = c(2, 4, 6,
8, 10))
ggplot(data, aes(x = x, y = y)) + geom_point()
```

Свойства функции ggplot2

1. **aes():** определение эстетических атрибутов графика.

```
ggplot(data, aes(x = x, y = y, color = y)) + geom_point()
```

2. **geom_point():** добавление точек на график.

```
ggplot(data, aes(x = x, y = y)) + geom_point()
```

3. **geom_line():** добавление линий на график.

```
ggplot(data, aes(x = x, y = y)) + geom_line()
```

4. **geom_bar():** создание столбчатой диаграммы.

```
data <- data.frame(x = c("A", "B", "C", "D"),
y = c(10, 20, 15, 25))
ggplot(data, aes(x = x, y = y)) + geom_bar(
stat = "identity")
```

5. **geom_histogram():** создание гистограммы.

```
data <- data.frame(values = rnorm(100))
ggplot(data, aes(x = values)) + geom_histogram()
```

6. **geom_smooth():** добавление сглаженной линии на график.

```
ggplot(data, aes(x = x, y = y)) + geom_point() +
geom_smooth()
```

7. **geom_text():** добавление текстовых меток на график.

```
data <- data.frame(x = c(1, 2, 3), y = c(2, 4, 6),
label = c("A", "B", "C"))
ggplot(data, aes(x = x, y = y, label = label)) +
geom_point() + geom_text()
```

Это лишь небольшой набор примеров использования функций ggplot2 для создания различных типов графиков и визуализаций данных. Пакет ggplot2 предоставляет множество других функций и возможностей для настройки графиков.

Рассмотрим простое приложение Shiny, которое использует ggplot2 для построения графика синусоиды в зависимости от пользовательского ввода:

```
library(ggplot2)

# Определение UI
ui <- fluidPage(
  titlePanel("График синусоиды с ggplot2"),
  sidebarLayout(
    sidebarPanel(
```

```
      sliderInput("freq", "Выберите частоту:",
min = 1, max = 10, value = 5)
    ),
    mainPanel(
      plotOutput("sin_plot")
    )
  )
)

# Определение Server функции
server <- function(input, output) {

  output$sin_plot <- renderPlot({
    freq <- input$freq
    x <- seq(0, 2*pi, length.out = 100)
    y <- sin(freq * x)

    df <- data.frame(x = x, y = y)

    ggplot(df, aes(x = x, y = y)) +
      geom_line(color = "blue") +
      labs(x = "X", y = "sin(X)", title = "График
синусоиды") +
      theme_minimal()
  })
}

# Запуск приложения
shinyApp(ui = ui, server = server)
```

Выше приведен процесс создания простого приложения Shiny с интерфейсом, содержащим слайдер для выбора частоты синусоиды. В Server функции используется ggplot2 для построения графика синусоиды в зависимости от выбранной частоты. График отображается на пользовательском интерфейсе с помощью plotOutput(“sin_plot”). При запуске появится интерактивный график синусоиды, который будет обновляться в зависимости от выбранной частоты на слайдере.

5.2. plotly

Пакет plotly – популярный пакет в языке программирования R, который предоставляет возможность создания интерактивных графиков и визуализаций данных. Приведем несколько причин, почему plotly может быть полезен и зачем его нужно использовать.

1. **Интерактивные возможности:** plotly позволяет создавать интерактивные графики, которые пользователи могут масштабировать,

наводить для получения дополнительной информации, выделять и фильтровать данные, а также экспортировать графики в различные форматы.

2. **Простота использования:** пакет plotly предоставляет простой и интуитивно понятный способ создания интерактивных графиков в R. Синтаксис plotly похож на синтаксис пакета ggplot2, что упрощает его освоение для пользователей, знакомых с ggplot2.

3. **Широкие возможности:** plotly поддерживает различные типы графиков, включая диаграммы рассеяния, линейные графики, гистограммы, ящики с усами, тепловые карты и многое другое. Таким образом можно создавать графики для различных типов данных и задач.

4. **Интерактивные дашборды:** с помощью plotly есть возможность разрабатывать интерактивные дашборды и веб-приложения с несколькими графиками, таблицами и элементами управления для взаимодействия пользователя с данными.

5. **Интеграция с библиотекой Shiny:** plotly хорошо интегрируется с библиотекой Shiny.

Благодаря plotly получаются красивые, информативные и интерактивные графики, а визуализация данных – более привлекательной и эффективной. Если необходимо создать визуализации данных, которые пользователи могут исследовать и взаимодействовать с ними, то plotly может быть отличным выбором для этой цели.

Ниже приведен полный код для трех расширенных примеров интерактивных графиков с использованием пакета plotly в библиотеке Shiny. Можно смело использовать этот код в вашей среде R для запуска приложения Shiny с интерактивными графиками:

```
library(shiny)
library(plotly)

# Пример 1: Интерактивная диаграмма рассеяния
ui <- fluidPage(
  titlePanel("Интерактивная диаграмма рассеяния"),
  mainPanel(
    plotlyOutput("scatter_plot")
  )
)

server <- function(input, output) {
  output$scatter_plot <- renderPlotly({
    plot_ly(x = rnorm(100), y = rnorm(100),
mode = "markers")
  })
}
```

```
# Пример 2: Интерактивный временной ряд
ui <- fluidPage(
  titlePanel("Интерактивный временной ряд"),
  mainPanel(
    plotlyOutput("time_series_plot")
  )
)

server <- function(input, output) {
  output$time_series_plot <- renderPlotly({
    x <- seq(as.Date("2022-01-01"), by = "day",
length.out = 100)
    y <- rnorm(100)
    plot_ly(x = x, y = y, type = "scatter", mode =
"lines")
  })
}

# Пример 3: Интерактивная гистограмма
ui <- fluidPage(
  titlePanel("Интерактивная гистограмма"),
  mainPanel(
    plotlyOutput("histogram_plot")
  )
)

server <- function(input, output) {
  output$histogram_plot <- renderPlotly({
    x <- rnorm(100)
    plot_ly(x = x, type = "histogram")
  })
}

# Запуск приложения Shiny
shinyApp(ui = ui, server = server)
```

Итак, интерактивные графики представляют собой мощный инструмент визуализации данных, который позволяет пользователям взаимодействовать с графиками, изменять параметры, получать дополнительную информацию и углублять свое понимание данных. Благодаря таким графикам визуализация становится более привлекательной, информативной и эффективной. А пользователи, в свою очередь, могут лучше понять данные, искать взаимосвязи и принимать информированные решения на основе визуализаций.

Глава 6
ВИДЖЕТЫ И ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Виджеты и *элементы управления* – это интерактивные компоненты, которые добавляют функциональность к интерактивным графикам и визуализациям данных. Они позволяют пользователям взаимодействовать с графиками, изменять параметры, фильтровать данные и получать дополнительную информацию. В библиотеке Shiny в R виджеты и элементы управления могут быть легко интегрированы в веб-приложения для получения интерактивных пользовательских интерфейсов.

Некоторые примеры виджетов и элементов управления, которые могут быть использованы в Shiny для взаимодействия с графиками, включают в себя: слайдеры (sliderInput), выбор из списка (selectInput), флажки (checkboxInput), текстовые поля (textInput), диапазон дат (dateRangeInput), кнопки (actionButton), текстовые метки (textOutput), таблицы (tableOutput).

6.1. sliderInput

Слайдеры (sliderInput) – это виджеты в библиотеке Shiny, позволяющие пользователям выбирать числовые значения из заданного диапазона, перемещая ползунок вдоль шкалы. Слайдеры могут быть использованы для выбора параметров, настройки параметров модели, фильтрации данных и других целей. Ниже приведен пример использования слайдера в Shiny:

```
library(shiny)

# Определение пользовательского интерфейса (UI)
ui <- fluidPage(
  titlePanel("Пример слайдера в Shiny"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("slider", "Выберите значение:",
min = 1, max = 100, value = 50),
      textOutput("slider_value")
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)
```

```
# Определение серверной части (Server)
server <- function(input, output) {
  output$slider_value <- renderText({
    paste("Выбранное значение слайдера:", input$slider)
  })

  output$plot <- renderPlot({
    x <- seq(1, input$slider)
    y <- x^2
    plot(x, y, type = "l", col = "blue", xlab = "X",
ylab = "Y")
  })
}

# Запуск веб-приложения Shiny
shinyApp(ui = ui, server = server
```

Здесь создается веб-приложение Shiny с использованием слайдера. Пользователь может выбирать значение от 1 до 100, перемещая ползунок; выбранное значение отображается на странице. Также строится график квадратичной функции, где X – это значения от 1 до выбранного числа на слайдере, а Y – это квадраты этих значений.

Слайдеры представляют собой удобный и интуитивно понятный способ для пользователей выбирать числовые значения из заданного диапазона. Они могут быть использованы для настройки параметров, фильтрации данных, интерактивной настройки графиков и многих других целей в веб-приложениях Shiny.

6.2. selectInput

Выбор из списка (selectInput) позволяет пользователям выбирать одно или несколько значений из выпадающего списка.

Пример:

```
library(shiny)

# Определение пользовательского интерфейса (UI)
ui <- fluidPage(
  titlePanel("Пример выбора из списка в Shiny"),
  sidebarLayout(
    sidebarPanel(
      selectInput("choice", "Выберите элемент:",
choices = c("A", "B", "C")),
      textOutput("selected_value")
    ),
```

```
    mainPanel(
      plotOutput("plot")
    )
  )
)

# Определение серверной части (Server)
server <- function(input, output) {
  output$selected_value <- renderText({
    paste("Выбранный элемент из списка:",
input$choice)
  })

  output$plot <- renderPlot({
    if(input$choice == "A") {
      plot(1:10, type = "l", col = "blue", xlab = "X",
ylab = "Y")
    } else if(input$choice == "B") {
      plot(10:1, type = "l", col = "red", xlab = "X",
ylab = "Y")
    } else {
      plot(1:5, type = "l", col = "green", xlab = "X",
ylab = "Y")
    }
  })
}

# Запуск веб-приложения Shiny
shinyApp(ui = ui, server = server)
```

В данной программе создается веб-приложение Shiny с использованием элемента управления «Выбор из списка». Пользователь может выбирать один из элементов списка “A”, “B” или “C”. Выбранный элемент отображается на странице, а также строится соответствующий график в зависимости от выбора пользователя.

Очень полезным инструментом при написании программ может являться выпадающий список. Код на языке R, демонстрирующий использование элемента управления «Выпадающий список» (selectInput) в библиотеке Shiny для создания интерактивного веб-приложения:

```
library(shiny)

# Определение пользовательского интерфейса (UI)
ui <- fluidPage(
  titlePanel("Пример выпадающего списка в Shiny"),
  sidebarLayout(
```

```
    sidebarPanel(
      selectInput("fruit", "Выберите фрукт:",
choices = c("Яблоко", "Груша", "Банан")),
      textOutput("selected_fruit")
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)

# Определение серверной части (Server)
server <- function(input, output) {
  output$selected_fruit <- renderText({
    paste("Выбранный фрукт:", input$fruit)
  })

  output$plot <- renderPlot({
    if(input$fruit == "Яблоко") {
      plot(1:10, type = "l", col = "green",
xlab = "X", ylab = "Y")
    } else if(input$fruit == "Груша") {
      plot(10:1, type = "l", col = "orange",
xlab = "X", ylab = "Y")
    } else {
      plot(1:5, type = "l", col = "yellow",
xlab = "X", ylab = "Y")
    }
  })
}

# Запуск веб-приложения Shiny
shinyApp(ui = ui, server = server)
```

Код создает веб-приложение Shiny с использованием элемента управления «Выпадающий список». Пользователь может выбирать один из предложенных вариантов фруктов: «Яблоко», «Груша» или «Банан». Выбранный фрукт отображается на странице, а также строится соответствующий график в зависимости от выбора пользователя.

6.3. checkboxInput & radioButtons

Флажки (*checkboxInput*) и *радиокнопки* (*radioButtons*) – это элементы управления, которые позволяют пользователям выбирать определенные опции или значения в приложении.

Характеристики флажков (checkboxInput):

– представляют собой переключатели, которые позволяют пользователям выбирать одно или несколько значений из списка;

– пользователь может установить флажок (выбрать значение) или снять флажок (сбросить значение);

– обычно используются для фильтрации данных, выбора нескольких параметров или опций.

Характеристики радиокнопок (radioButtons):

– представляют собой группу взаимосвязанных кнопок, среди которых пользователь может выбрать только одну опцию;

– обычно используются для выбора одного параметра из нескольких взаимоисключающих опций.

Пример использования флажков и радиокнопок в Shiny:

```
library(shiny)

ui <- fluidPage(
  titlePanel("Пример флажков и радиокнопок в Shiny"),
  sidebarLayout(
    sidebarPanel(
      checkboxInput("show_plot", "Показать график",
value = TRUE),
      radioButtons("color", "Выберите цвет:",
choices = c("Красный", "Синий", "Зеленый"),
selected = "Красный")
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)

server <- function(input, output) {
  output$plot <- renderPlot({
    if(input$show_plot) {
      color <- switch(input$color, "Красный" = "red",
"Синий" = "blue", "Зеленый" = "green")
      plot(1:10, col = color, type = "l", xlab = "X",
ylab = "Y")
    }
  })
}

shinyApp(ui = ui, server = server)
```

В данном примере пользователь может выбирать, показывать ли график с помощью флажка «Показать график» и выбирать цвет графика с помощью радиокнопок «Выберите цвет». В зависимости от выбора пользователя, на странице отображается график с выбранным цветом.

6.4. textInput

Текстовые поля (*textInput*) – это элемент управления в библиотеке Shiny, который дает возможность пользователям вводить текстовую информацию в интерактивном веб-приложении. Пользователь может вводить текст, числа, даты и другие данные с клавиатуры в текстовое поле. Текстовые поля обычно используются для ввода информации, параметров, запросов или фильтрации данных.

Пример использования текстового поля (textInput) в Shiny:

```
library(shiny)

ui <- fluidPage(
  titlePanel("Пример текстового поля в Shiny"),
  sidebarLayout(
    sidebarPanel(
      textInput("name", "Введите ваше имя:",
value = ""),
      textOutput("greeting")
    ),
    mainPanel(
      verbatimTextOutput("text_output")
    )
  )
)

server <- function(input, output) {
  output$greeting <- renderText({
    paste("Привет, ", input$name, "!")
  })

  output$text_output <- renderPrint({
    input$name
  })
}

shinyApp(ui = ui, server = server)
```

Данный код демонстрирует, как пользователь может ввести свое имя в текстовое поле «Введите ваше имя». При вводе имени на странице отобразится приветственное сообщение «Привет, [введенное имя]!». Также будет отображаться введенное имя в текстовом формате.

Текстовые поля являются удобным способом для пользователей взаимодействовать с веб-приложением, вводя текстовую информацию. Они могут быть использованы для различных целей, включая ввод параметров, фильтрацию данных, поиск и многое другое.

6.5. dateRangeInput

Диапазон дат (*dateRangeInput*) – это элемент управления в библиотеке Shiny, который позволяет пользователям выбирать диапазон дат с помощью календаря в интерактивном веб-приложении. Пользователь может выбрать начальную и конечную дату в заданном диапазоне, что удобно для фильтрации данных по временному периоду. Диапазоны дат часто используются для анализа временных данных, построения временных графиков или отчетов. Ниже представлен пример использования диапазона дат (dateRangeInput) в Shiny:

```
library(shiny)

# Определение пользовательского интерфейса (UI)
ui <- fluidPage(
  titlePanel("Пример диапазона дат в Shiny"),
  sidebarLayout(
    sidebarPanel(
      dateRangeInput("dates", "Выберите диапазон
дат:", start = "2022-01-01", end = "2022-12-31"),
      textOutput("selected_dates")
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)

# Определение серверной части (Server)
server <- function(input, output) {
  output$selected_dates <- renderText({
    paste("Выбранный диапазон дат:", input$dates[1],
"до", input$dates[2])
  })
```

```
   output$plot <- renderPlot({
     start_date <- as.Date(input$dates[1])
     end_date <- as.Date(input$dates[2])
     dates <- seq(start_date, end_date, by = "day")
     values <- rnorm(length(dates))
     plot(dates, values, type = "l", col = "blue",
 xlab = "Дата", ylab = "Значение")
   })
 }

 # Запуск веб-приложения Shiny
 shinyApp(ui = ui, server = server)
```

В этом коде создается веб-приложение Shiny с использованием элемента управления «Диапазон дат». Пользователь может выбирать начальную и конечную дату в заданном диапазоне с помощью календаря. Выбранный диапазон дат отображается на странице, а также строится график со случайными значениями в выбранном временном периоде.

6.6. buttons

Кнопки (*actionButton*) – это элемент управления в библиотеке Shiny, позволяющий пользователям запускать определенные действия или операции в интерактивном веб-приложении. Пользователь может нажать на кнопку для выполнения определенного действия, например, загрузки данных, обновления графика или запуска анализа. Кнопки обычно используются для управления функциональностью приложения и запуска определенных действий по запросу пользователя. Приведем пример использования кнопки (actionButton) в Shiny:

```
library(shiny)

# Определение пользовательского интерфейса (UI)
ui <- fluidPage(
  titlePanel("Пример кнопки в Shiny"),
  sidebarLayout(
    sidebarPanel(
      actionButton("btn", "Нажмите для обновления
графика"),
      textOutput("click_message")
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)
```

```
# Определение серверной части (Server)
server <- function(input, output, session) {
  click_count <- reactiveVal(0)

  observeEvent(input$btn, {
    click_count(click_count() + 1)
  })

  output$click_message <- renderText({
    paste("Количество нажатий кнопки:", click_count())
  })

  output$plot <- renderPlot({
    plot(1:10, type = "l", col = "blue", xlab = "X",
ylab = "Y")
  })
}

# Запуск веб-приложения Shiny
shinyApp(ui = ui, server = server)
```

В данном коде создается веб-приложение Shiny с использованием элемента управления «Кнопка». Пользователь может нажать на кнопку «Нажмите для обновления графика», чтобы обновить график и увеличить счетчик нажатий кнопки. При каждом нажатии кнопки увеличивается счетчик, отображающийся на странице. Также на странице отображается график, который можно обновлять по запросу пользователя.

6.7. textOutput

Элемент управления «Текстовая метка» (textOutput) в библиотеке Shiny используется для динамического отображения текстовой информации или результатов вычислений на веб-странице в реальном времени. Этот элемент позволяет выводить текстовую информацию, которая может быть обновлена или изменена в зависимости от действий пользователя или других событий в приложении.

Рассмотрим пример использования элемента управления «Текстовая метка» (textOutput) в Shiny:

```
library(shiny)

ui <- fluidPage(
  titlePanel("Пример текстовой метки в Shiny"),
  sidebarLayout(
    sidebarPanel(
```

```
      numericInput("number", "Введите число:",
value = 0),
      actionButton("update", "Обновить текст")
    ),
    mainPanel(
      textOutput("result_text")
    )
  )
)

server <- function(input, output) {
  output$result_text <- renderText({
    if(input$update > 0) {
      paste("Вы ввели число:", input$number)
    } else {
      "Пожалуйста, нажмите кнопку 'Обновить текст'"
    }
  })
}

shinyApp(ui = ui, server = server)
```

В примере пользователь может ввести число в числовое поле и нажать кнопку «Обновить текст». Текстовая метка «result_text» будет динамически обновляться в зависимости от введенного числа и действий пользователя.

Элемент управления «Текстовая метка» (textOutput) позволяет создавать интерактивные и динамические приложения, где текстовая информация может быть обновлена в реальном времени в соответствии с действиями пользователя или изменениями данных.

6.8. tableOutput

Элемент управления «Таблица» (tableOutput) в библиотеке Shiny применяется для отображения данных в виде таблицы на веб-странице в интерактивном веб-приложении. Таблица позволяет выводить структурированные данные в виде таблицы, что обеспечивает удобный способ представления информации для пользователя. Пользователь может видеть данные в удобном формате, прокручивать таблицу, сортировать столбцы и выполнять другие действия для работы с данными. Код для использования элемента управления «Таблица» (tableOutput) в Shiny может выглядеть так:

```
library(shiny)

ui <- fluidPage(
  titlePanel("Пример таблицы в Shiny"),
  sidebarLayout(
    sidebarPanel(
      selectInput("dataset", "Выберите набор
данных:", choices = c("iris", "mtcars")),
      actionButton("show_table", "Показать таблицу")
    ),
    mainPanel(
      tableOutput("data_table")
    )
  )
)

server <- function(input, output) {
  output$data_table <- renderTable({
    switch(input$dataset,
           "iris" = iris,
           "mtcars" = mtcars)
  })
}

shinyApp(ui = ui, server = server)
```

В приведенном примере кода создается веб-приложение Shiny, в котором пользователь может выбрать набор данных (iris или mtcars) из выпадающего списка, нажать кнопку «Показать таблицу», и выбранные данные будут отображены в виде таблицы на веб-странице.

Элемент управления «Таблица» (tableOutput) позволяет удобно представлять структурированные данные в виде таблицы, что облегчает визуализацию и анализ данных для пользователя.

Подводя итог изученным виджетам и элементам управления в Shiny, рассмотрим следующий пример кода:

```
library(shiny)

ui <- fluidPage(
  titlePanel("Пример использования виджетов
и элементов управления"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("slider", "Выберите значение:",
min = 1, max = 100, value = 50),
```

```
      selectInput("select", "Выберите элемент:",
choices = c("A", "B", "C")),
      actionButton("button", "Нажмите кнопку"),
      textInput("text", "Введите текст:")
    ),
    mainPanel(
      textOutput("text_output"),
      tableOutput("table_output")
    )
  )
)

server <- function(input, output) {
  output$text_output <- renderText({
    paste("Выбранное значение слайдера:",
input$slider)
  })

  output$table_output <- renderTable({
    data.frame(A = 1:3, B = c("X", "Y", "Z"))
  })
}

shinyApp(ui = ui, server = server)
```

Пример демонстрирует использование различных виджетов и элементов управления в Shiny, таких как слайдер, выбор из списка, кнопка, текстовое поле, текстовая метка и таблица. Пользователь может взаимодействовать с виджетами и увидеть результаты в реальном времени на веб-странице.

Использование виджетов и элементов управления в Shiny позволяет создавать более интерактивные и функциональные веб-приложения для анализа данных и визуализаций.

Глава 7
РАСШИРЕННЫЕ ВОЗМОЖНОСТИ SHINY

7.1. Shiny модули

Shiny модули – это функциональность в пакете Shiny для R, позволяющая разработчикам структурировать и организовывать сложные Shiny-приложения. Модули помогают управлять повторно используемыми Shiny-компонентами, соблюдать принципы DRY, сокращать дублирование кода и повышать его читаемость.

Каждый модуль Shiny состоит из двух основных частей:

1. **UI function:** функция для создания пользовательского интерфейса модуля, которая возвращает UI код для вставки в общий интерфейс приложения;

2. **Server function:** функция, содержащая серверную логику модуля, то есть реактивные выражения и обработку событий, связанные с данным модулем.

Модули помогают в решении проблем, связанных с масштабированием приложения, поскольку они позволяют:

– разрабатывать и тестировать отдельные компоненты: модули можно разрабатывать в изоляции, что упрощает тестирование и отладку;

– избегать конфликтов в пространствах имен: модули имеют свои собственные пространства имен, что позволяет избежать конфликтов имен в больших приложениях;

– повторно использовать код: если определенные элементы UI и части серверной логики повторяются, их можно вынести в модули и повторно использовать в разных частях приложения;

– создавать четкую структуру приложения: крупные приложения часто могут стать сложными для понимания и поддержки, при этом модули позволяют декомпозировать их на управляемые, легко понимаемые блоки.

Чтобы использовать модуль в Shiny-приложении, разработчик должен:

– **определить модульные функции:** создать UI и серверные функции модуля;

– **вызвать UI функцию модуля:** вставить вызов UI функции в главный UI код приложения;

– **вызвать серверную функцию модуля:** вставить вызов серверной функции модуля с помощью callModule в серверной части

главного приложения, передавая ей соответствующие реактивные переменные и входные параметры.

Пример использования Shiny модулей:

```
# Определение UI функции модуля
myModuleUI <- function(id) {
  ns <- NS(id)
  tagList(
    selectInput(ns("input"), "Choose a number",
choices = 1:10),
    textOutput(ns("output"))
  )
}

# Определение серверной функции модуля
myModuleServer <- function(input, output, session) {
  output$output <- renderText({
    paste("You selected", input$input)
  })
}

# Использование модуля в приложении
ui <- fluidPage(
  myModuleUI("module1")
)

server <- function(input, output, session) {
  callModule(myModuleServer, "module1")
}

shinyApp(ui, server)
```

В данном примере myModuleUI и myModuleServer определяются как UI и серверные компоненты модуля. Затем они используются в основном приложении, где модуль “module1” инициализируется и вызывается. Это обеспечивает инкапсуляцию и возможность повторного использования модулей.

7.2. Интеграция Shiny с JavaScript

Интеграция Shiny с JavaScript представляет собой набор техник, обеспечивающих для разработчиков взаимодействие между серверным кодом R и клиентским JavaScript. Это дает возможность расширения стандартных функций Shiny и создания более интерактивных и динамичных пользовательских интерфейсов.

Примеры использования интеграции JavaScript в Shiny перечислены ниже.

1. **Манипуляция с DOM:** изменение элементов DOM (Document Object Model) веб-страницы на стороне клиента.

2. **Обработка событий:** отслеживание пользовательских действий, таких как клики мыши, перемещение мыши, нажатия клавиш и многое другое.

3. **Асинхронное взаимодействие:** загрузка данных без перезагрузки страницы (AJAX).

4. **Анимации и эффекты:** создание анимаций и других визуальных эффектов.

5. **Интеграция с внешними библиотеками:** использование мощных библиотек JavaScript, таких как D3.js для визуализации данных, Leaflet для картографии и др.

Возможности интеграции с JavaScript в Shiny включают:

– **вставку JavaScript кода напрямую в UI:** можно вставлять собственный JavaScript код непосредственно в приложение Shiny с помощью функции tags$script;

– **использование Shiny.onInputChange:** это функция JavaScript Shiny, которая позволяет отправлять сообщения из JavaScript обратно в серверный R код. Сообщения могут быть любыми данными, на которые серверный код может реагировать;

– **использование Shiny.addCustomMessageHandler:** функция JavaScript для определения пользовательских обработчиков сообщений на стороне клиента, которые способны обрабатывать данные, отправленные сервером R;

– **создание пользовательских виджетов Shiny:** можно создать собственные HTML виджеты, которые полностью интегрируются с Shiny и R.

Пример использования JavaScript в Shiny:

R код (Server):

```
# Server-Side R Code
shinyServer(function(input, output, session) {

  # Send a message to the JavaScript client
  observe({
    input$someAction
    session$sendCustomMessage(type = 'my_message',
message = 'Hello from R!')
  })
})
```

JavaScript код (UI):

```
// Client-Side JavaScript Code
$(document).on('shiny:connected', function(event) {

  // Define custom handler for custom messages of
type 'my_message'
  Shiny.addCustomMessageHandler('my_message',
    function(message) {
      alert(message); // Do something with the
message from R
    }
  );
});
```

Этот код демонстрирует, как сервер Shiny в R может отправить сообщение, которое JavaScript затем перехватывает и обрабатывает на стороне клиента. Когда происходит определенное действие на стороне сервера (someAction), R отправляет сообщение my_message, которое JavaScript обрабатывает, выводя сообщение через alert.

Данная интеграция открывает путь для очень богатого интерактивного взаимодействия в приложениях Shiny и делает возможной реализацию сложных клиентских логик и визуализаций.

7.3. Кастомизация UI с использованием HTML/CSS

В контексте Shiny «кастомизация UI с использованием HTML/CSS» означает использование пользовательского HTML и таблиц стилей CSS для создания уникального внешнего вида и поведения веб-приложения. Это возможность, позволяющая отходить от стандартной схемы виджетов и макетов, которые предлагает Shiny, и кастомизировать приложение в соответствии с уникальными требованиями проекта.

HTML: пользователь по своему усмотрению может создать HTML-разметку, которая впоследствии будет вставлена в интерфейс приложения Shiny. Это может включать в себя:

– прямое использование HTML-тегов в UI с помощью функций, таких как tags$, HTML() и tagList();

– интеграцию пользовательских HTML-шаблонов и использование разметки Bootstrap для создания сложных макетов.

CSS: CSS используется для добавления стилей и визуальных эффектов к элементам HTML интерфейса пользователя. С его помощью можно:

– изменять цвета, размеры шрифтов, отступы, рамки и прочие аспекты внешнего вида виджетов и контейнеров UI;

– применять CSS-фреймворки и препроцессоры (типа SASS или LESS) для более удобной разработки стилей;

– создавать адаптивные и мобильно-оптимизированные веб-приложения с использованием медиа-запросов.

Для добавления пользовательского HTML и CSS в Shiny, можно использовать эти методы:

```
# Для добавления пользовательских стилей с помощью
инлайн-CSS или внешнего файла:
tags$style(HTML("
  .class {
    property: value;
  }
")),
includeCSS("www/style.css"),

# Для добавления пользовательского HTML:
tags$div(
  class = "custom-class",
  "Some custom content"
),

# Использование HTML() функции для вставки
неэкранированного HTML:
HTML('<div class="custom-div">Custom HTML
content</div>')
```

Использование дополнительного HTML и CSS в Shiny дает возможность разработчикам создавать приложения, которые могут соответствовать корпоративному стилю, быть более привлекательными для конечных пользователей и обеспечивать более богатый пользовательский опыт по сравнению с базовыми Shiny виджетами и макетами.

7.4. Информационные панели Shiny

Shiny Dashboard – это специализированный пакет в R, расширяющий возможности Shiny для создания информационных панелей (дэшбордов). *Дэшборды* – это веб-приложения, обычно используемые для отображения метрик, ключевых показателей эффективности

(KPIs), графиков и других данных, которые нужно мониторить в реальном времени.

Пакет shinydashboard предлагает набор функций и UI компонентов, которые помогают разрабатывать интерактивные приборные панели с профессиональным дизайном, без необходимости глубоких знаний в веб-разработке. Перечислим некоторые из его основных компонентов.

1. **dashboardPage:** основная функция, которая создает страницу дэшборда и является контейнером для всех остальных компонентов.

2. **dashboardHeader:** создает шапку дэшборда, которая обычно содержит название приложения, меню выбора и другие элементы навигации.

3. **dashboardSidebar:** создает боковую панель, предлагающую различные элементы управления, такие как вкладки, меню и другие входные компоненты. Это место для взаимодействия пользователя с дэшбордом.

4. **dashboardBody:** главный контент дэшборда размещается здесь; это может включать в себя графики, таблицы, и другие Shiny модули.

shinydashboard также предлагает дополнительные компоненты для улучшения визуального восприятия и функциональности дэшборда:

– **valueBox:** для отображения главных показателей или кратких статистических данных;

– **infoBox:** для представления дополнительной информации, с возможностью добавления иконок;

– **box:** универсальное поле для отображения содержимого, может содержать графики, таблицы, или другой контент.

Shiny Dashboard позволяет быстро и эффективно создавать адаптивные приборные панели для интерактивного отображения данных. С помощью такого пакета можно за считанные минуты создать полноценное приложение, пригодное для презентации широкой аудитории.

Пример создания простого Shiny Dashboard:

```
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(title = "My Dashboard"),
  dashboardSidebar(
    sidebarMenu(
      menuItem("Dashboard", tabName = "dashboard",
icon = icon("dashboard")),
```

```
      menuItem("Widgets", tabName = "widgets",
  icon = icon("th"))
    )
  ),
  dashboardBody(
    tabItems(
      tabItem(tabName = "dashboard",
        h2("Dashboard content")
      ),
      tabItem(tabName = "widgets",
        h2("Widgets content")
      )
    )
  )
)

server <- function(input, output) { }

shinyApp(ui, server)
```

В этом примере создается дэшборд с двумя вкладками: Дашборд и Виджеты. Скелет приложения содержит шапку, боковую навигационную панель и тело дэшборда. Функциональная часть в данном случае остается пустой, так как это лишь шаблон.

7.5. Расширенные входные и выходные данные

“Advanced Shiny Inputs/Outputs” относится к расширенным методам ввода и вывода данных в приложениях Shiny, которые могут предоставить больше контроля и интерактивности, чем базовые виджеты ввода и вывода.

Advanced Inputs (расширенные входные данные) играют ключевую роль в интерактивности приложения Shiny, позволяя пользователям отправлять данные на сервер. Это может содержать выбор определенного значения из списка, ввод текста, выбор даты в календаре и т. д. Более сложные варианты включают использование слайдеров для выбора диапазона значений, выбор цвета или файла, выбор точек на карте и т. д. Advanced Inputs являются специализированными виджетами ввода, которые обеспечивают улучшенный пользовательский интерфейс для работы с более сложными данными или действиями, такими как:

– **дополненные текстовые поля** для автодополнения, тегирования и форматирования;

– **специализированные слайдеры:** позволяют выбирать диапазоны, даты и т. д.;

– **выбор цвета:** для взаимодействия с палитрой цветов;

– **загрузка файлов и папок:** расширенная интеграция с файловой системой пользователя;

– **ползунки для выбора времени:** позволяют пользователю устанавливать время;

– **интерактивные карты:** возможность выбора местоположений или регионов;

– **специализированные виджеты** для более сложных вариантов ввода (например, выбор из иерархического списка).

Advanced Outputs (расширенные выходные данные) улучшают способы представления и визуализации данных, включая:

– **интерактивные таблицы:** с использованием пакетов DT или reactable, которые предлагают возможности сортировки, фильтрации и пагинации;

– **интерактивные графики и диаграммы:** визуализация данных с помощью plotly, highcharter или других HTML-виджетов;

– **сложные визуализации**, такие как сетевые графики, 3D модели и тепловые карты;

– **динамическое отображение растровых изображений:** работа с изображениями высокого разрешения;

– **виджеты документов:** для интеграции и отображения файлов, например, PDF-документов;

– **кастомизированные панели (dashboards):** с использованием shinydashboard, для отображения показателей и метрик;

– **видео и аудио плееры:** интеграция мультимедийного контента непосредственно в приложение.

Пример использования расширенных входов и выходов в приложении Shiny:

```
library(shiny)
library(plotly)

ui <- fluidPage(
    # Advanced Input
    plotlyOutput("advanced_plot"),
    DT::dataTableOutput("interactive_table")
)

server <- function(input, output, session) {
```

```
    # Advanced Output: Interactive plot
    output$advanced_plot <- renderPlotly({
        # Place code for plotly plot here
        plotly::plot_ly(data = mtcars, x = ~mpg,
y = ~wt, type = 'scatter', mode = 'markers')
    })

    # Advanced Output: Interactive table
    output$interactive_table <- DT::renderDataTable({
        # Place code for data table here
        DT::datatable(data = mtcars,
options = list(pageLength = 5))
    })
}

shinyApp(ui, server)
```

Данный код демонстирует, как сложный график и интерактивная таблица могут быть добавлены в Shiny. Эти компоненты предоставляют пользователю возможность взаимодействия с данными и получения информации в реальном времени, добавляя значительную ценность аналитическому и визуализационному потенциалу приложения.

7.6. Интегрирование баз данных

Интеграция баз данных в Shiny приложения позволяет управлять данными и анализировать данные, хранящиеся за пределами локальной среды R, что делает приложения более мощными и гибкими при работе с большими объемами данных или при необходимости обеспечить многопользовательский доступ к данным.

Приведем несколько ключевых моментов и функций, связанных с интеграцией баз данных в Shiny.

1. **Подключение к Базам Данных (БД)**

Есть возможность подключаться к различным видам баз данных, включая SQL (MySQL, PostgreSQL), NoSQL (MongoDB), или к более специализированным, например, TimescaleDB или Google BigQuery.

2. **Использование пакетов R для работы с БД**

Для работы с базами данных в R существуют специализированные пакеты, такие как DBI, RMariaDB, RPostgres и RSQLite, которые предоставляют интерфейс к реляционным системам управления базами данных (РСУБД). Есть и другие пакеты, вроде mongolite для MongoDB и bigrquery для работы с Google BigQuery.

3. Работа с пулами соединения

Вместо того, чтобы открывать новое соединение с базой данных каждый раз, когда пользователь выполняет запрос, можно использовать пакет pool для создания пула соединений и повышения производительности приложения.

4. Интерактивные запросы

С помощью реактивных выражений в Shiny сервер может выполнять запросы к базе данных на основе входных данных пользователя и возвращать результаты для отображения в приложении.

5. Безопасность

При подключении к базе данных из Shiny приложения важно уделять внимание безопасности, используя, например, шифрованные соединения и безопасное хранение учетных данных.

Пример интеграции базы данных в Shiny приложения:

```
library(shiny)
library(DBI)

# Определите глобальные переменные здесь
pool <- dbPool(RPostgres::Postgres(),
               dbname = "your_dbname",
               host = "host",
               user = "user",
               password = "password",
               port = 5432)

ui <- fluidPage(
    # UI элементы ...
)

server <- function(input, output, session) {
    # Серверная логика, включающая запросы
к базе данных
    data_reactive <- reactive({
        query <- "SELECT * FROM your_table WHERE
some_condition;"
        dbGetQuery(pool, query)
    })

    output$table <- DT::renderDataTable({
        data_reactive()
    })
}
shinyApp(ui, server)
```

Здесь используется пул соединений для эффективного управления подключениями к базе данных PostgreSQL. Серверная логика Shiny запрашивает данные из базы данных на основе определенных критериев и отображает их в интерактивной таблице.

Интеграция баз данных с приложениями Shiny значительно повышает потенциал обработки данных и создания интерактивных пользовательских интерфейсов для аналитических платформ, отчетности и других предприятий, где требуется динамичный доступ к данным.

7.7. Аутентификация пользователей

В контексте веб-приложений, включая Shiny, *аутентификация пользователей* (*User Authentication*) – это процесс проверки личности пользователя, запросившего доступ к каким-либо ресурсам. Веб-приложения используют системы аутентификации для установления того, допустимы ли пользователю доступ к определенным данным или выполнение определенных действий.

В Shiny по умолчанию не включен механизм аутентификации, однако его можно реализовать различными способами, перечисленными ниже.

1. **Shiny Server Pro:** коммерческая версия Shiny Server включает в себя функционал аутентификации пользователей с поддержкой LDAP, Google Auth и другие варианты.

2. **RStudio Connect:** платформа RStudio Connect предоставляет более продвинутые возможности для размещения Shiny приложений, включая встроенные инструменты аутентификации.

3. **Custom Authentication:** можно создать собственную систему аутентификации, используя Shiny модули и дополнительные R пакеты, такие как shinyauthr, shinymanager, или использовать Shiny в сочетании с другими рамками и базами данных, такими как Firebase, для обработки аутентификации и авторизации.

4. **ShinyProxy:** предоставляет удобную функциональность для развертывания Shiny приложений в контейнерах Docker с поддержкой аутентификации и авторизации.

5. **HTTP Authentication:** можно использовать HTTP аутентификацию через веб-сервер, такой как Apache или Nginx, который будет стоять перед Shiny приложением и управлять доступом.

Функции аутентификации обычно включают в себя элементы ввода логина и пароля, возможность регистрировать новых пользователей, восстанавливать пароли и управлять пользовательскими

сессиями. Это, несомненно, важный аспект разработки приложения, так как аутентификация напрямую связана с безопасностью и разграничением доступа к важным бизнес-данным и функционалу.

Пример создания системы аутентификации в Shiny с использованием пакета shinyauthr:

```
library(shiny)
library(shinyauthr)

# данные для аутентификации
user_base <- data.frame(
  user = c("user1", "user2"),
  password = c("pass1", "pass2"),
  stringsAsFactors = FALSE
)

ui <- fluidPage(
  # функция пользовательского интерфейса для
аутентификации
  shinyauthr::loginUI(id = "login"),
  # содержимое приложения, доступное после входа
  uiOutput("app_content")
)

server <- function(input, output, session) {
  # вызов аутентификации сервера
  login_status <- shinyauthr::loginServer(
    id = "login",
    data = user_base,
    user_col = user,
    pwd_col = password
  )

  output$app_content <- renderUI({
    # только показываем содержимое, если пользователь
прошел аутентификацию
    req(login_status()$success)
    # далее содержимое вашего приложения...
  })
}

shinyApp(ui, server)
```

В этом приложении используется shinyauthr для создания простой системы входа в систему, которая защищает UI элементы, делая их доступными только после успешного входа пользователя в систему.

7.8. Масштабирование Shiny

Масштабирование приложений, разработанных в Shiny, является важным аспектом, особенно когда нужно обслуживать большое количество пользователей одновременно. Shiny Server Open Source Edition не включает в себя масштабируемые возможности или контроль доступа пользователей, что ограничивает его использование в больших организациях или приложениях с высокой нагрузкой. Однако существуют два решения от RStudio, которые предлагают дополнительные возможности для масштабирования и управления приложениями Shiny: Shiny Server Pro и RStudio Connect.

7.8.1. Shiny Server Pro

Shiny Server Pro представляет собой коммерческую версию Shiny Server и предлагает дополнительные функции, отсутствующие в открытом варианте:

– **аутентификация пользователей:** возможность входа в систему с использованием распространенных схем аутентификации, таких как LDAP/AD, Google Auth и других;

– **управление доступом:** позволяет ограничить доступ к приложениям на основе определенных правил или групп пользователей;

– **балансировка нагрузки сессий:** обрабатывает несколько пользовательских сессий и распределяет нагрузку между различными экземплярами приложения;

– **мониторинг и управление приложениями:** панель администрирования для мониторинга активности пользователей, просмотра логов и анализа использования ресурсов;

– **масштабируемость:** поддержка высокой доступности и развертываемости для обеспечения стабильной работы приложения при большом количестве одновременных пользователей.

7.8.2. RStudio Connect

RStudio Connect – это платформа для развертывания, управления и масштабирования приложений R и Shiny, отчетов R Markdown, документов Jupyter и других ресурсов в продакшн окружении. Основные возможности RStudio Connect:

– **интегрированная аутентификация:** поддержка множества провайдеров аутентификации;

– **управление доступом к контенту:** можно тонко настроить, кто способен видеть и взаимодействовать с опубликованным контентом;

– **простое масштабирование:** Connect может автоматически масштабировать приложение в зависимости от нагрузки, создавая дополнительные экземпляры приложения при возрастании количества пользователей;

– **непрерывная интеграция:** поддержка процессов непрерывной интеграции и непрерывного развертывания для автоматизированного обновления приложений.

Обе платформы поддерживают высокую доступность (High Availability, HA) и отказоустойчивость (failover), позволяя создать надежную инфраструктуру для критически важных приложений, требующих круглосуточного доступа.

Использование Shiny Server Pro или RStudio Connect позволяет организациям масштабировать свои аналитические веб-приложения до уровня, требуемого для обслуживания большого количества пользователей, обеспечивая при этом необходимый уровень качества и безопасности.

7.9. Условное отображение UI

Conditional UI означает условное отображение интерфейса пользователя: определенные элементы UI могут появляться, исчезать или изменяться в зависимости от входных данных или действий пользователя.

С использованием реактивных выражений в Shiny приложении разработчики могут создать динамический пользовательский интерфейс, который изменяется в реальном времени, что позволяет более точно соответствовать потребностям и вводимой информации пользователей, создавая интуитивно понятный и адаптивный опыт.

Примеры Conditional UI в Shiny:

1. Использование conditionalPanel в UI:

```
    conditionalPanel(
      condition = "input.someinput > 5",
      sliderInput("moreInput", "Input:", min = 1,
 max = 100, value = 50)
    )
```

В этом примере sliderInput будет отображаться только в том случае, если значение someinput больше 5.

2. Использование uiOutput и renderUI на стороне сервера:

```
    uiOutput("conditionalUI")

    output$conditionalUI <- renderUI({
      if (input$someinput > 5) {
        sliderInput("moreInput", "Input:", min = 1,
 max = 100, value = 50)
      }
    })
```

Здесь на основе входных данных someinput сервер решает, выводить ли sliderInput в UI.

3. Прямое использование реактивных значений для включения/отключения интерфейса:

```
    output$someOutput <- renderText({
      if (is.null(input$someInput)) {
        "Please enter a value"
      } else {
        paste("You entered:", input$someInput)
      }
    })
```

Сообщение будет меняться в зависимости от того, введено ли значение в someInput.

Conditional UI улучшает пользовательский опыт, облегчает навигацию по приложению и помогает пользователям избегать ненужных взаимодействий или перегруженного интерфейса, особенно когда доступно множество опций или настройки.

7.10. Фоновая обработка данных

Обработка данных в фоновом режиме (Background Data Processing) в контексте приложений Shiny – это выполнение задач обработки данных в отдельном процессе или сессии, чтобы основное приложение оставалось отзывчивым и не блокировалось на время выполнения этих задач.

Обычно, если в Shiny приложении запускается длительная операция (например, загрузка большого набора данных или выполнение сложных вычислений), то интерфейс пользователя может «зависать» или становиться недоступным на время выполнения этой операции.

При обработке данных в фоновом режиме эти задачи выполняются параллельно основному процессу приложения, что позволяет пользовательскому интерфейсу оставаться отзывчивым.

Несколько способов, как осуществить обработку данных в фоне в Shiny, представлены ниже.

Библиотека future

Библиотека future в R позволяет планировать выполнение кода в отдельном сеансе R, что обеспечивает его выполнение в фоновом режиме. Комбинируя future с Shiny, можно запускать вычисления асинхронно, чтобы они не влияли на производительность интерфейса пользователя.

Пример использования future с Shiny:

```
library(shiny)
library(future)

plan(multisession)

ui <- fluidPage(
    actionButton("go", "Go"),
    textOutput("result")
)

server <- function(input, output) {
    values <- reactiveValues(data = NULL)

    observeEvent(input$go, {
        future({
            # Длительный расчет
            Sys.sleep(5)
            rnorm(200)
        }) %...>% {
            values$data <- .
        }
    })

    output$result <- renderText({
        if (is.null(values$data)) "Not calculated yet"
        else paste("Result is", mean(values$data))
    })
}

shinyApp(ui, server)
```

В этом примере использование %...>% начинает длительную операцию в новой сессии R.

Промисы (Promises)

Промисы в R, реализованные в пакете promises, дают возможность работать с асинхронным кодом, подобно JavaScript. Промисы можно делать цепочками, что позволяет последовательно выполнять асинхронные операции.

Пример использования промисов в Shiny:

```
library(shiny)
library(promises)

ui <- fluidPage(
    actionButton("go", "Go"),
    verbatimTextOutput("result")
)

server <- function(input, output) {
    output$result <- renderText({
        paste("The result is", input$go)
    })

    observeEvent(input$go, {
        promise <- future({
            # Выполнение длительной операции
            Sys.sleep(5)
            rnorm(1)
        })
        promise %>%
            then(~{
                output$result <- renderText({
                    paste("Async result is", .)
                })
            })
    })
}

shinyApp(ui, server)
```

В представленном коде совершаются следующие действия:

– функция renderText связана с текстовым выводом и реагирует на изменения переменной input$go, которая представляет собой счетчик нажатий кнопки “Go”;

– в обработчике событий кнопки (нажатие на “Go”) запускается обещание (promise), которое выполняет длительную операцию (имитируемую паузой Sys.sleep(5)) и возвращает случайное число из нормального распределения (функция rnorm(1));

– функция then, которая определяет, что делать после выполнения promise. Она изменяет текстовый вывод, отображая асинхронный результат (случайное число).

Библиотека shiny.background

Пакет shiny.background позволяет запускать длительные процессы в фоновых работах и автоматически обновлять интерфейс пользователя при их завершении.

Пример использования shiny.background похож на примеры с использованием future или promises, но реализован с использованием специфических функций пакета.

Использование фоновой обработки данных улучшает пользовательский опыт, позволяя интерфейсу оставаться отзывчивым, показывать прогресс выполнения задачи и предотвращая непредвиденные блокировки интерфейса. Обработка в фоновом режиме особенно полезна для Shiny приложений, предполагающих тяжелые вычислительные операции или большой объем данных.

7.11. Расширенные возможности визуализации

В Shiny приложениях термин Advanced “Visualizations” относится к более сложным и интерактивным способам визуализации данных, выходящих за рамки базовых статических графиков, сгенерированных с помощью базовых пакетов R, таких как ggplot2 и plot.

Рассмотрим примеры расширенной визуализации.

1. **Interactive Plots:** используя пакеты, например, plotly, пользователи могут взаимодействовать с графиками, например, наводить курсор на точки данных для отображения подробной информации, увеличивать и перемещать график.

2. **3D-графика:** пакеты, такие как rgl и threejs, позволяют создавать интерактивные 3D-графики.

3. **Time Series Analysis:** для анализа временных рядов используются такие пакеты, как dygraphs, обеспечивающие интерактивный просмотр больших временных рядов.

4. **Геопространственные данные:** интеграция с пакетами leaflet для карт и mapview, которые дают возможность создавать интерактивное картографическое представление данных.

5. **Network Graphs:** визуализация сетевых данных с помощью networkD3 или visNetwork.

6. **Heatmaps:** использование heatmaply или plotly для создания интерактивных тепловых карт.

7. **Data Tables:** разработка интерактивных таблиц с большим количеством функций (сортировка, фильтрация, пагинация), с использованием пакета DT.

8. **Custom JavaScript Visualization:** возможность интеграции пользовательских визуализаций, созданных с прямым использованием JavaScript или через библиотеки типа D3.js.

Пример использования расширенных визуализаций в Shiny:

```
library(shiny)
library(plotly)

ui <- fluidPage(
    plotlyOutput("interactivePlot")
)

server <- function(input, output, session) {

    # Advanced Output: Interactive plot with 'plotly'
    output$interactivePlot <- renderPlotly({
        data <- mtcars
        p <- ggplot(data, aes(x = wt, y = mpg)) +
geom_point()
        ggplotly(p) # Превращаем ggplot граф
в интерактивную визуализацию plotly
    })
}

shinyApp(ui, server)
```

Здесь ggplotly из пакета plotly используется для добавления интерактивности приложению с использованием Shiny. Пользователь может наводить курсор на точки данных, чтобы увидеть дополнительную информацию, а также воспользоваться функциями увеличения и перемещения графика.

Расширенные визуализации значительно повышают степень представления данных в приложениях Shiny, делая аналитику более интерактивной и информативной для конечных пользователей. Представленные расширенные особенности и возможности делают Shiny мощным инструментом для создания богатых и динамичных веб-приложений для визуализации данных и проведения аналитики в режиме онлайн.

Глава 8
ОПТИМИЗАЦИЯ ПРОИЗВОДИТЕЛЬНОСТИ ПРИЛОЖЕНИЙ

Оптимизация производительности приложений Shiny – это процесс улучшения времени отклика приложения и его способности обрабатывать большее количество пользователей и/или данных. Оптимизация производительности приложений Shiny – ключевой компонент успешного взаимодействия пользователя с приложением. Это достигается не только через улучшение времени отклика приложения, но и через повышение его способности эффективно обрабатывать большой объем пользовательских запросов и данных. Для достижения этих целей используются различные методы, включая кэширование результатов, асинхронные операции, предварительную загрузку данных и оптимизацию кода. Эти подходы помогают снизить задержку и повысить масштабируемость приложений Shiny.

8.1. Кэширование вычислений

Кэширование вычислений в контексте компьютерных программ и веб-приложений, в том числе и созданных с помощью Shiny, – это процесс временного сохранения результатов дорогостоящих (вычислительно затратных) операций для их быстрого повторного использования. Целью кэширования является снижение нагрузки на систему и уменьшение времени ожидания пользователем повторно запрашиваемых данных или результатов выполнения кода.

Кэширование является важным аспектом работы, поскольку вычисления могут быть «дорогими» по ряду причин:

- они требуют значительной процессорной мощности;
- заставляют пользователя ждать окончания операции;
- осуществляют множественные запросы к внешним ресурсам, таким как API или базы данных.

Рассмотрим, как работает кэширование вычислений.

1. **Первичные вычисления:** при первом запросе выполняется полное вычисление.

2. **Сохранение результата:** результат вычисления сохраняется в кэше с идентификатором, основанным на входных параметрах.

3. **Повторный доступ:** при повторном запросе с теми же параметрами, вместо нового вычисления, результат извлекается из кэша и возвращается пользователю.

Варианты реализации в R и Shiny:

– **кэширование в глобальной области:** используя `global.R` файл в Shiny, можно кэшировать результаты, которые будут общими для всех сессий пользователя;

– **реактивные выражения:** реактивные выражения в Shiny автоматически кэшируют свои значения, и если их зависимости не изменяются, то они не будут пересчитаны;

– **memoise пакет:** данный пакет может быть использован для кэширования обычных R функций, что очень удобно для офлайн-вычислений в R;

– **кэширование в пользовательской сессии:** переменные и значения `reactiveValues` или `reactiveVal` могут быть использованы для кэширования посчитанных данных в контексте одной сессии Shiny.

Пример кода с memoise в Shiny:

```
library(shiny)
library(memoise)

# Допустим, у нас есть дорогостоящая функция
для вычисления
expensiveComputation <- function(input) {
  # Здесь могут быть сложные вычисления
  Sys.sleep(2) # имитация дорогостоящего вычисления
  return(input * 2)
}

# Используем memoise для кэширования результатов этой
функции
cachedComputation <- memoise(expensiveComputation)

ui <- fluidPage(
  numericInput('num', 'Choose a number', 5),
  actionButton("go", "Calculate"),
  textOutput("result")
)

server <- function(input, output, session) {
  result <- eventReactive(input$go, {
    # используем кэшированную версию для вычисления
    cachedComputation(input$num)
  })


  output$result <- renderText({
    result()
  })
}

shinyApp(ui, server)
```

В этом примере функция `expensiveComputation` кэшируется с помощью `memoise`. Когда пользователь вводит значение и нажимает кнопку “Calculate”, вместо выполнения обычной функции `expensiveComputation`, вызывается кэшированная версия `cachedComputation`. Если пользователь введет то же число, что и раньше, то результат будет мгновенно получен из кеша, вместо повторного выполнения затратных вычислений.

Виды кэширования в R и Shiny.

1. **Кэширование на уровне сессии**

Этот вид кэширования актуален для кэширования данных в пределах одной сессии Shiny. В этом случае, когда пользователь начинает новый сеанс Shiny, данные не переносятся из предыдущих сессий. Используются функции reactiveValues(), reactiveVal().

2. **Кэширование на уровне приложения**

Данные сохраняются на протяжении всех сессий. Данный вид кэширования полезен для данных, которые являются общими для всех пользователей и которые не нуждаются в обновлении с каждой новой сессией. Значение хранится в глобальном окружении и доступно в каждой новой сессии.

3. **Кэширование на стороне пользователя**

Это кэширование, которое запоминает результаты вычислений в браузере пользователя, например, в LocalStorage или с помощью cookies. Это может быть полезно для индивидуальных пользовательских настроек или данных, специфичных для конкретного браузера, чтобы предоставить пользователям персонализированный опыт.

4. **Кэширование с использованием внешних хранилищ данных**

Иногда можно использовать внешние системы кэширования, такие как Redis или Memcached, для управления кэшем на уровне инфраструктуры. Эти системы предлагают продвинутые возможности для сохранения и быстрого извлечения данных и могут быть использованы в распределенных системах, где множество экземпляров Shiny работают параллельно.

Общие рекомендации для кэширования:

– **валидация кэша:** удостоверьтесь, что кэш обновляется, когда исходные данные или параметры изменяются;

– **срок действия кэша (TTL):** определите время жизни кэшированных данных, после которого они будут считаться устаревшими;

– **размер кэша:** мониторьте и ограничивайте объем памяти, используемой кэшем, чтобы избежать переполнения;

– **безопасность:** при кэшировании личных данных пользователей обратите внимание на соответствующие меры безопасности и приватности.

Применяя эти стратегии, можно значительно повысить производительность приложений Shiny и обеспечить лучший пользовательский опыт благодаря сокращению времени ожидания ответа от сервера.

8.2. Асинхронные операции

Асинхронные операции – это такие задачи или процессы, выполнение которых происходит независимо от основного потока выполнения программы. В отличие от синхронного выполнения, где задачи выполняются одна за другой, асинхронное выполнение позволяет компьютеру перейти к следующей задаче, не дожидаясь окончания предыдущей. Это особенно важно в сетевых запросах, вводе/выводе данных и других операциях, которые могут занять значительное время.

Преимущества асинхронных операций:

– **улучшение отзывчивости:** веб-приложения и интерфейсы остаются отзывчивыми, пока осуществляется фоновая загрузка или обработка данных;

– **параллелизм:** можно одновременно запускать несколько операций, что эффективно использует системные ресурсы;

– **не блокирующее поведение:** главный поток программы не блокируется ожиданием, например, окончания длительного запроса к базе данных или завершения сетевого запроса.

Применение асинхронности в R и Shiny

R и Shiny классически известны своим синхронным выполнением кода, но это можно обойти с помощью асинхронных библиотек и пакетов в R.

Пример асинхронного выполнения с использованием пакета promises в Shiny:

```
library(shiny)
library(promises)
library(future)

plan(multisession) # Указываем, что будем
использовать асинхронные вычисления

ui <- fluidPage(
  actionButton("load", "Load Data"),
  tableOutput("data")
)
```

```
  server <- function(input, output) {
    reactData <- eventReactive(input$load, {
      future({
        # Длительная асинхронная операция
        Sys.sleep(5) # Имитация задержки
        mtcars # Возвращаем данные
      }) %>% then(~{
        # Результат асинхронной операции
        .
      })
    })

    output$data <- renderTable({
      reactData()
    })
  }

  shinyApp(ui = ui, server = server)
```

В этом примере при нажатии на кнопку “Load Data” запускается асинхронная операция, которая имитирует долгий процесс загрузки данных. Вместо того, чтобы блокировать UI, операция выполняется в отдельном процессе, и данные становятся доступны после их загрузки.

Пример использования пакета future:

```
library(future)

# Запланировать задачу для асинхронного выполнения
plan(multisession)


# Запуск длительной операции в фоновом режиме
aFuture <- future({
  # Длительный расчет или загрузка данных
  Sys.sleep(10)
  return(42)
})

# Затем, можно продолжить выполнять другой код
result <- value(aFuture)
```

Такой подход позволяет серверу Shiny продолжить обрабатывать другие запросы пользователей или обновлять UI, в то время как фоновая задача выполняется отдельно. Это улучшает масштабируемость и пользователям не приходится ждать завершения одной операции для выполнения следующей.

Асинхронные операции являются мощным способом повышения производительности в вычислительно интенсивных приложениях или там, где важно поддерживать интерактивность интерфейса во время обработки данных.

Глава 9
АДАПТИВНЫЙ ДИЗАЙН И СТИЛИЗАЦИЯ

Данный раздел предназначен для тех, кто хочет научиться применять принципы адаптивного дизайна и стилизации при создании приложений Shiny. Есть возможность научиться использовать CSS и интегрировать Bootstrap для создания отзывчивых и эстетически притягательных веб-интерфейсов.

9.1. Основы CSS в Shiny

CSS является центральным для стилизации и является неотъемлемой частью разработки веб-приложений. Shiny-приложения не являются исключением, поскольку они в конечном счете выполняются в браузере как обычные веб-страницы.

Shiny позволяет интегрировать CSS несколькими способами.

1. **Инлайн CSS:** стилизация прямо внутри UI компонентов Shiny. Например,

```
actionButton("action", "Do Action", style = "color: red;")
```

Однако инлайн стили могут быть трудны в поддержке и масштабировании, поэтому их лучше использовать для быстрых исправлений или тестирования.

2. **Встроенные стили (Inner CSS):** стилизация внутри тега tags$head в основной функции UI. Пример кода:

```
ui <- fluidPage(
  tags$head(
    tags$style(HTML("
      .class-name {
        background-color: blue;
        color: white;
      }
    "))
  ),
  # ... другие UI элементы
)
```

3. **Внешние стилевые листы:** отдельный файл .css, который подключается к Shiny приложению. Это предпочтительный способ управления стилями, особенно для больших приложений.

```
ui <- fluidPage(
  includeCSS("www/style.css"),
  # ... другие UI элементы
)
```

В этом случае файл style.css должен быть помещен в папку www в каталоге Shiny-приложения.

Селекторы, свойства и значения

Чтобы создать файл CSS, нужно разобраться в синтаксисе, который состоит из селекторов, свойств и их значений. Селекторы определяют, к какому элементу страницы применяются стили.

Типы селекторов:

- div, p, h1 ... – селекторы типа элемента;
- class-name – селекторы класса;
- #element-id – селекторы идентификатора;
- [attribute="value"] – селекторы атрибута.

Свойства и значения описывают, как именно стилизовать выбранный элемент.

Примеры свойств:

- color – цвет текста;
- background-color – цвет фона;
- margin, padding – внешние и внутренние отступы;
- font-size – размер шрифта;
- border – граница.

Примеры стилей в файле style.css:

```
/* Стилизация всех элементов <h1> на странице */
h1 {
  color: navy;
  font-size: 24px;
}

/* Стилизация элементов с классом 'highlight' */
.highlight {
  background-color: yellow;
}
```

```
/* Стилизация элемента с идентификатором
'main-button' */
#main-button {
  padding: 10px 20px;
  border: none;
  background-color: green;
  color: white;
}
```

Приоритет стилей

Когда один и тот же элемент стилизуется несколькими правилами, браузер применяет правила в следующей последовательности приоритета:

1. Инлайн стили (самый высокий приоритет);
2. Внутренние стилевые листы;
3. Внешние стилевые листы;
4. Стили браузера по умолчанию (самый низкий приоритет).

Если в разных стилях задано одно и то же свойство для одного элемента, будет применено правило с самым высоким приоритетом. Кроме того, CSS следует принципу «каскада», где более специфические селекторы (например, идентификаторы) переопределяют менее специфические (например, типы элементов).

Эти базовые понятия в сочетании с практикой и тестированием позволят широко применять CSS для стилизации ваших Shiny приложений. Следующий шаг будет включать применение этих знаний для стилизации конкретных компонентов приложения Shiny.

Перейдем к настройке стиля отдельных элементов интерфейса. Ниже приведены некоторые примеры того, как можно использовать CSS для управления внешним видом различных элементов Shiny.

1. **Изменение вида кнопок**

CSS позволяет вам контролировать такие параметры кнопок, как цвет фона, цвет текста, радиус границы и т. д.

```
/ Стилизация всех кнопок действия /
.action-button {
  background-color: blue;
  color: white;
  border-radius: 10px;
}

/ Стили в состоянии hover /
.action-button:hover {
  background-color: lightblue;
}
```

```
/ Стилизация конкретной кнопки по id /
далее
#main-button {
  padding: 10px 15px;
  border: 2px solid #4CAF50;
  background-color: #4CAF50;
}

#main-button:hover {
  background-color: #45a049;
}
```

В приведенном примере добавляются пользовательские стили для кнопки с идентификатором main-button. У кнопки будут округлые углы, увеличенные внутренние отступы и зеленый фон, который становится темнее при наведении курсора.

2. **Настройка текстовых полей**

Текстовые поля (input) могут быть подвергнуты различной стилизации: изменению ширины поля, стилей границ, фокусных стилей и другого.

```
/ Стилизация всех текстовых полей /
.shiny-input-container inputtype="text" {
  border: 1px solid #ccc;
  box-shadow: inset 0 1px 3px #ddd;
  border-radius: 4px;
}

/ Стили для состояния focus /
.shiny-input-container inputtype="text":focus {
  border-color: #66afe9;
  outline: 0;
  box-shadow: inset 0 1px 1px #ddd, 0 0 8px rgba
(102, 175, 233, .6);
}
```

Здесь целенаправленно стилизуем текстовые поля, добавленные в Shiny приложение. Добавляем внутреннюю тень для большей глубины, а также меняем стили границ и тени при фокусе, чтобы пользователь четко видел, на каком элементе он сейчас находится.

3. **Управление разметкой сетки**

С помощью CSS можно управлять расположением и выравниванием элементов в Shiny, которые по умолчанию располагаются в столбец или строку.

```
/ Стилизация контейнеров fluidRow /
.row {
  margin-bottom: 20px;
}

/ Управление стилями для отдельных columns /
.col-sm-4 {
  padding: 0 10px;
}
```

В данном примере определенные стили для классов .row и .col-sm-4 улучшают визуальное восприятие сеточной разметки, добавляя отступы и изменяя их размер.

4. **Адаптивные дизайнерские элементы**

Для создания отзывчивости можно использовать медиа-запросы для изменения стилей в зависимости от размера экрана.

```
/ Стили для экранов среднего размера /
@media (min-width: 768px) {
  .sidebar {
    width: 200px;
    float: left;
  }

  .main-content {
    margin-left: 220px;
  }
}

/ Стили для мобильных устройств /
@media (max-width: 767px) {
  .sidebar {
    display: none;
  }

  .main-content {
    margin-left: 0;
  }
}
```

Пример выше показывает, как можно адаптировать макет страницы в зависимости от размера экрана, убирая боковую панель для мобильных устройств и возвращая его на экраны среднего размера.

Все эти примеры стоит подключить к своему Shiny приложению, используя ссылку на внешний CSS-файл или определения в tags$style

внутри ui.R файла. Создавая пользовательские стили, необходимо стремиться поддерживать разделение сущностей и чистоту кода, чтобы облегчить его будущее обслуживание и обновление.

9.2. Адаптивный дизайн

Адаптивный дизайн – это подход в веб-разработке, при котором дизайн и разметка веб-страницы оптимально подстраиваются под размер экрана устройства пользователя. Главная цель адаптивного дизайна – обеспечить удобство просмотра и взаимодействия с сайтом, независимо от того, пользуется ли пользователь настольным компьютером, планшетом или смартфоном.

В контексте Shiny приложений адаптивный дизайн важен для обеспечения лучшего пользовательского опыта. Shiny приложения, как и любые другие веб-приложения, можно просматривать на любом устройстве с доступом к интернету. Адаптивный дизайн позволяет убедиться, что приложение выглядит и функционирует корректно на всех типах устройств.

9.2.1. Ключевые аспекты адаптивного дизайна

1. **Медиа-запросы (Media Queries)** в CSS позволяют применять различные стили в зависимости от условий, таких как ширина экрана, высота, разрешение и др. Это ядро технологии адаптивного дизайна.

```
@media screen and (max-width: 600px) {
  .sidebar {
    display: none;
  }
}
```

В этом примере .sidebar скрывается, когда ширина экрана меньше 600 пикселей.

2. **Сетки и фреймворки (Grids and frameworks):** существует множество CSS-фреймворков, включая Bootstrap, Foundation и другие, которые предоставляют готовые к использованию адаптивные сетки. Эти сетки используются для создания сложных макетов, которые автоматически адаптируются под разные размеры экранов.

3. **Гибкие изображения и сетки (Flexible Images and Grids):** адаптивный дизайн часто включает «резиновые» изображения и сетки. Это означает, что размеры элементов задаются в относительных

единицах (%, vh, vw), что позволяет им эластично масштабироваться в зависимости от размера контейнера или экрана.

4. **Вьюпорты (Viewports):** тег <meta name="viewport"> в HTML говорит браузеру, как контролировать масштаб и размеры страницы. Установка viewport необходима для корректного адаптивного дизайна, особенно для мобильных устройств.

5. **Гибкие макеты (Flexible layouts)** позволяют создавать макеты, которые могут легко адаптироваться к изменениям размеров экрана без необходимости изменения HTML структуры.

9.2.2. Адаптивность в Shiny

В Shiny приложениях принципы адаптивного дизайна используются для оптимизации интерактивной работы с приложением. Shiny предлагает ряд встроенных функций и аргументов для улучшения адаптивности, такие как fluidPage, fluidRow и column, которые автоматически масштабируют содержимое приложения под размер экрана.

```
shinyUI(fluidPage(
  titlePanel("My Shiny App"),
  fluidRow(
    column(4, wellPanel(...)),
    column(8, wellPanel(...))
  )
))
```

В этом макете fluidRow создает ряд, который занимает всю доступную ширину экрана, а column автоматически определяет рамер колонок в зависимости от ширины экрана.

Лучшие практики для адаптивного дизайна в Shiny:

– **тест приложения:** необходимо регулярно проверять, как ваше приложение отображается и работает на различных устройствах и размерах экранов;

– **использование встроенных стилей Shiny:** нужно обратить внимание на классы и идентификаторы, которые предоставляет Shiny, и использовать их для стилизации;

– **использование медиа-запросов:** применять медиа-запросы для мелкой настройки отображения приложения на различных устройствах;

– **Mobile-First:** необходимо рассматривать мобильные устройства как первостепенную платформу, на которой будет использоваться

приложение. Это поможет сфокусироваться на самом важном контенте и функционале.

Взяв за основу эти принципы и техники, можно будет разработать Shiny приложения, которые будут оптимально выглядеть и работать на любых устройствах, предоставляя пользователям высококачественный интерфейс, независимо от их используемого устройства.

9.3. Интеграция Bootstrap

Bootstrap – это один из самых популярных фронтенд фреймворков для разработки адаптивных и мобильных проектов. Он обеспечивает широкий набор инструментов для стилизации веб-страниц, включая систему сеток, компоненты интерфейса и JavaScript плагины. Использование Bootstrap в приложениях Shiny позволяет легко и быстро создавать адаптивные интерфейсы.

Приложения Shiny по умолчанию уже имеют встроенную поддержку Bootstrap, поэтому для большинства случаев нет необходимости в дополнительной интеграции. Однако если вы хотите изменить стандартный Bootstrap шаблон или использовать конкретную тему Bootstrap, то понадобится более точечная настройка.

Сетка Bootstrap

Bootstrap использует 12-колоночную сетку для размещения контента на странице, которую можно настраивать под конкретные размеры экрана. В Shiny можно использовать fluidRow и column для работы со сеткой Bootstrap:

```
ui <- fluidPage(
  fluidRow(
    column(3, "Колонка 1"),
    column(6, "Колонка 2"),
    column(3, "Колонка 3")
  )
)
```

9.3.1. Компоненты Bootstrap

Компоненты Bootstrap – это набор предварительно стилизованных элементов интерфейса, которые могут быть легко интегрированы в веб-приложения. В контексте Shiny это означает возможность использования разнообразных UI элементов, таких как кнопки, карты (cards), навигационные панели (navbars), выпадающие списки

(dropdowns), всплывающие подсказки (tooltips), модальные окна (modals) и многое другое, для создания интерактивного и насыщенного пользовательского интерфейса.

Bootstrap интегрируется в Shiny по умолчанию, и многие стандартные компоненты Shiny автоматически получают стили Bootstrap. Однако для более продвинутых компонентов Bootstrap, таких как модальные окна или вкладки, нужно использовать специальные Shiny-функции или HTML-разметку с классами Bootstrap.

1. **Создание кнопок Bootstrap**

Кнопки в Shiny можно легко стилизовать с помощью классов Bootstrap:

```
    # Пример кнопки Bootstrap в Shiny
    actionButton("someAction", "Take Action", class =
    "btn-primary")
```

2. **Использование навигационных панелей (Navbars)**

```
    navbarPage("Название приложения",
      tabPanel("Главная", "Содержимое..."),
      tabPanel("О приложении", "Содержимое...")
    )
```

3. **Работа с модальными окнами**

Модальные окна – это всплывающие окна, которые можно использовать для дополнительных уведомлений или для выполнения действий без перехода на другую страницу.

```
# Элемент UI для модального окна
ui <- fluidPage(
  actionButton("showModal", "Показать модальное окно")
)

# Логика сервера для отображения модального окна
server <- function(input, output) {
  observeEvent(input$showModal, {
    showModal(modalDialog(
      title = "Важное сообщение",
      "Это текст внутри модального окна.",
      footer = modalButton("Закрыть")
    ))
  })
}
```

4. **Карточки (Cards)**

Карточки представляют собой гибкие и экстенсивные контейнеры для контента, с опциями для заголовка, футера и широким спектром содержимого.

```
# Создание карточки в Shiny с HTML и классами
Bootstrap
ui <- fluidPage(
  tags$div(class = "card",
    tags$div(class = "card-body",
      tags$h5(class = "card-title", "Заголовок
карточки"),
      tags$p(class = "card-text", "Текст внутри
карточки.")
    )
  )
)
```

5. **Вкладки и Пиллы (Tabs and Pills)**

Bootstrap позволяет создавать компоненты навигации, такие как вкладки и пиллы, для организации связанного контента.

```
# Использование вкладок Bootstrap в Shiny
ui <- navbarPage(
  "Название приложения",
  tabPanel("Вкладка 1", "Контент..."),
  navbarMenu("Дополнительно",
    tabPanel("Вкладка 2", "Контент..."),
    tabPanel("Вкладка 3", "Контент...")
  )
)
```

6. **Использование выпадающих списков и группы кнопок (Dropdowns и Button Groups)**

Выпадающие списки и группы кнопок могут быть полезны для представления родственных действий или опций.

```
# Создание Dropdown в Shiny
ui <- fluidPage(
  dropdownButton(
    label = "Опции",
    icon = icon("cog"),
    status = "primary",
    circle = TRUE,
    # Содержимое Dropdown
    actionButton("action1", "Действие 1"),
    actionButton("action2", "Действие 2")
  )
)
```

7. **Меню навигации и боковая панель (Sidebar)**

```
# Построение боковой панели в Shiny
sidebar <- dashboardSidebar(
  sidebarMenu(
    menuItem("Главная", tabName = "dashboard",
icon = icon("dashboard")),
    menuItem("Отчеты", tabName = "reports",
icon = icon("file"))
  )
)

# Использование sidebar в UI
ui <- dashboardPage(
  dashboardHeader(),
  sidebar,
  dashboardBody()
)
```

Эти компоненты могут быть тонко настроены и адаптированы под потребности вашего Shiny приложения благодаря универсальности классов Bootstrap и гибкости Shiny в их применении. Важно помнить, что, хотя Shiny включает Bootstrap, все же может понадобиться прямое взаимодействие с HTML и классами Bootstrap, чтобы полностью использовать весь доступный потенциал фреймворка.

9.3.2. Пользовательские темы Bootstrap

Для использования пользовательской темы Bootstrap необходимо подключить ее в tags$head вашего Shiny UI. Можно выбрать одну из множества доступных тем Bootstrap, которые предоставляются различными ресурсами.

Пользовательские темы Bootstrap позволяют разработчикам Shiny значительно ускорить и упростить процесс стилизации приложений, придавая им уникальный внешний вид с минимальными усилиями. Вместо написания большого количества собственного CSS кода, разработчики могут воспользоваться темами, предложенными Bootstrap, или создать свою тему с помощью инструментов для кастомизации.

Использование готовых тем Bootstrap в Shiny

Shiny по умолчанию использует Bootstrap, что позволяет с легкостью включить готовые темы из экосистемы Bootstrap. Далее представлена инструкция по их использованию.

1. Необходимо выбрать тему на сайтах, предлагающих бесплатные или платные темы Bootstrap, таких как Bootswatch.

2. Скачать файлы CSS темы и поместить их в каталог /www вашего Shiny приложения.

3. Использовать includeCSS() в ui.R или fluidPage для подключения темы к вашему приложению.

```
ui <- fluidPage(
  includeCSS("www/имя_вашей_темы.css"),
  # остальная часть вашего UI...
)
```

Также есть возможность использовать пакет shinythemes, который предлагает набор стандартных Bootstrap тем, готовых к использованию прямо в R.

```
library(shiny)
library(shinythemes)

ui <- fluidPage(
  theme = shinytheme("flatly"),  # Пример
использования темы 'flatly'
  # остальная часть вашего UI...
)
```

Создание пользовательских тем Bootstrap

Для создания собственной темы Bootstrap можно использовать официальный инструмент Bootstrap Theme Customizer, который позволяет настроить цвета, размеры шрифтов и компонентов, а также другие CSS-переменные, и затем скачать свою настройку в виде CSS.

Кастомизированные темы позволяют следующее:

- поддерживать фирменный стиль компании;
- выделять важные UI элементы приложения;
- улучшать визуальную гармонию и пользовательский опыт.

Примеры использования кастомизированной темы Bootstrap

Допустим, был создан CSS файл my-theme.css со следующими стилями:

```
/* my-theme.css */
body {
  background-color: #f4f4f4;
}
```

```
.navbar {
  background-color: #333;
  border-bottom: 4px solid #3ea6ff;
}

.btn-primary {
  background-color: #3ea6ff;
  border-color: #3a9ae2;
}

.btn-primary:hover {
  background-color: #3592cc;
}
```

Этот файл можно подключить к Shiny следующим образом:

```
ui <- fluidPage(
  tags$head(
    # Собственные стили
    tags$link(rel="stylesheet", type="text/css",
href="my-theme.css")
  ),
  # ... UI код ...
)

server <- function(input, output) {
  # ... server код ...
}

shinyApp(ui = ui, server = server)
```

Важно помнить, что использование сторонних тем или кастомизированных тем Bootstrap может потребовать дополнительной проверки совместимости со стандартным стилем элементов Shiny, чтобы убедиться, что все компоненты будут корректно отображаться и функционировать.

9.3.3. Кастомизация CSS Bootstrap

Кастомизация CSS Bootstrap в Shiny позволяет улучшить визуальную составляющую приложений, изменяя стандартные стили и создавая уникальные темы. Вы можете настроить почти все аспекты внешнего вида: от цветов и шрифтов до размеров элементов и их поведения при взаимодействиях.

Кастомизация на уровне CSS классов Bootstrap

Один из самых простых способов кастомизации – это переопределение стандартных классов Bootstrap через собственные стили CSS.

```
/* Custom Bootstrap styling */
.navbar-custom {
  background-color: #555;
  color: white;
}

.btn-custom {
  background-color: #556B2F;
  color: white;
}

.form-control-custom {
  background-color: #f3f3f3;
  border-color: #dcdcdc;
}
```

Можно подключить эти стили, используя классы .navbar-custom, .btn-custom, .form-control-custom напрямую в своих UI компонентах Shiny.

Использование SASS для более глубокой кастомизации

Bootstrap использует препроцессор SASS, который способствует более тонкой настройке темы через переменные, миксины и функции. Если необходимо изменить основные цветовые схемы или типографику Bootstrap, то можно настроить исходные файлы SASS и скомпилировать их в CSS.

Пользовательские переменные позволяют сменить тематические константы Bootstrap, влияющие на весь проект.

Пример файла SASS со сменой переменных:

```
// Custom variable overrides
$theme-colors: (
  "primary": #4a8ada,
  "success": #52ad5b,
  "info": #14dae2,
  "warning": #f0ad4e,
  "danger": #dc3545,
  "light": #f2f5f8,
  "dark": #343a40
);

// Import Bootstrap source files
@import "bootstrap";
```

Внесение изменений в конкретные компоненты

Вариант стилизации отдельных компонентов Bootstrap, с использованием их классов:

```
.navbar {
  // изменить внешний вид навбара
  background-color: darken($primary, 10%);
}

.btn {
  // добавить стили для всех кнопок
  font-weight: bold;
  &:hover {
    box-shadow: 0 0 10px lighten($primary, 20%);
  }
}
```

При задействовании SASS кастомизация становится удобной и модульной, так как обеспечивается настройка тонких аспектов дизайна с сохранением чистоты и читаемости кода.

Подключение кастомизированных стилей к Shiny

После того как создан пользовательский файл CSS или скомпилирован SASS в CSS, необходимо подключить свои стили в UI приложения Shiny:

```
ui <- fluidPage(
  tags$head(
    tags$link(rel="stylesheet", type="text/css",
href="custom.css")
  ),
  # ... остальной UI ...
)
```

Рекомендации для кастомизации CSS Bootstrap в Shiny

1. **Настройка Bootstrap должна быть продуманной:** нет необходимости менять стили ради стилей. Убедитесь, что ваши изменения улучшают пользовательский опыт и делают интерфейс более понятным.

2. **Тестируйте на различных устройствах:** кастомизация должна сохранять адаптивность интерфейса и его корректное отображение на всех видах устройств и разрешениях экранов.

3. **Соблюдайте консистентность интерфейса:** кастомные стили должны гармонично вписываться в общий дизайн и логику интерфейса.

4. **Используйте переменные SASS для глобальных настроек:** переменные делают код легче для понимания и уменьшают вероятность ошибок при изменении стилей.

5. **Используйте контроль версий для отслеживания изменений:** ведение истории изменений позволит легко откатываться к предыдущим версиям и понять последствия внесенных изменений.

Внедряя эти практики, можно эффективно использовать Bootstrap для создания профессионального и привлекательного внешнего вида ваших Shiny приложений, обеспечивая одновременно высокую функциональность и удобство для конечных пользователей.

Мощность Bootstrap вливается в Shiny, давая ему возможность быть легко стилизованным и современным, что экономит время и усилия в процессе разработки пользовательского интерфейса.

Глава 10
ИНТЕРАКТИВНЫЕ КАРТЫ В SHINY

Библиотека Leaflet – это JavaScript библиотека для создания интерактивных карт на веб-страницах. Leaflet является одной из самых популярных и простых в использовании библиотек для работы с картами веб-приложений. Она предоставляет широкий набор функций для создания красивых и интерактивных карт с возможностью добавления маркеров, полигонов, линий, информационных окон, элементов управления и других элементов.

Leaflet легко интегрируется с различными картографическими провайдерами, такими как OpenStreetMap, Mapbox, Google Maps, Bing Maps и другими. Она также поддерживает мобильные устройства и адаптивный дизайн, что делает ее отличным выбором для создания кросс-платформенных интерактивных карт.

В R библиотека leaflet также доступна через пакет с одноименным названием, предоставляющий возможность создания интерактивных карт на основе Leaflet прямо из среды R. Пакет leaflet позволяет легко интегрировать карты в интерактивные веб-приложения, такие как Shiny.

10.1. Установка и подключение пакета leaflet

Шаг 1. Установка пакета leaflet

Открыть среду R (например, RStudio) на компьютере.

Ввести следующий код для установки пакета leaflet с использованием функции install.packages():

```
install.packages("leaflet")
```

Нажать “Enter”, чтобы запустить установку пакета. R начнет загружать и устанавливать пакет leaflet из репозитория CRAN. Дождаться завершения установки.

Шаг 2. Подключение пакета leaflet

После установки подключить пакет leaflet в вашем скрипте R с помощью функции library():

```
library(leaflet)
```

Запустить этот код, чтобы пакет leaflet был успешно подключен и готов к использованию.

Шаг 3. Пример использования пакета leaflet

После установки и подключения пакета leaflet можно создать простую интерактивную карту с маркером, например:

```
library(shiny)
library(leaflet)

ui <- fluidPage(
  titlePanel("Интерактивная карта с Leaflet в Shiny"),
  leafletOutput("map")
)

server <- function(input, output, session) {
  output$map <- renderLeaflet({
    leaflet() %>%
      setView(lng = 37.617635, lat = 55.755826,
zoom = 13) %>%
      addTiles() %>%
      addMarkers(lng = 37.617635, lat = 55.755826,
popup = "Красная площадь, Москва")
  })
}

shinyApp(ui = ui, server = server)
```

Этот код создает веб-приложение с использованием библиотек Shiny и Leaflet, которое отображает интерактивную карту с маркером в Москве.

Пошаговое объяснение, что делает каждая часть кода:

– **library(shiny):** эта строка подключает библиотеку Shiny;

– **library(leaflet):** эта строка подключает библиотеку Leaflet, которая позволяет создавать интерактивные карты на веб-страницах;

– **ui <- fluidPage(...):** в этой части определяется пользовательский интерфейс веб-приложения. В данном случае, создается страница с заголовком «Интерактивная карта с Leaflet в Shiny» и элементом leafletOutput(“map”), который будет отображать карту;

– **server <- function(...) {...}:** здесь определяется серверная часть приложения. Внутри функции renderLeaflet() создается интерактивная карта с помощью библиотеки Leaflet. Карта центрируется на координатах Москвы (долгота 37.617635, широта 55.755826) с масштабом зума 13. На карте добавляется маркер с информационным окном, указывающим на Красную площадь в Москве;

– **shinyApp(ui = ui, server = server):** эта строка запускает веб-приложение Shiny с определенным пользовательским интерфейсом и серверной логикой.

Таким образом, код создает веб-приложение, отображающее интерактивную карту с маркером на Красной площади в Москве.Теперь у вас установлен и подключен пакет leaflet в вашей среде R, и вы готовы создавать интерактивные карты.

10.2. Создание базовой интерактивной карты

Для создания базовой интерактивной карты любой сложности с использованием пакетов leaflet и Shiny в среде R нужно выполнить следующий алгоритм:

1. **Подготовка данных:** подготовить данные для отображения на карте. Это могут быть координаты точек, границы полигонов, маршруты и т. д.

2. **Создание пользовательского интерфейса (UI) веб-приложения Shiny:**
   - использовать функцию fluidPage() для создания основного макета веб-приложения;
   - добавить элемент leafletOutput(“map”), который будет отображать интерактивную карту.

3. **Написание серверной части приложения:**
   - использовать функцию renderLeaflet() для создания карты внутри серверной части;
   - внутри функции renderLeaflet() определить настройки карты, добавить необходимые слои (маркеры, линии, полигоны) и другие элементы.

4. **Запуск веб-приложения Shiny:** использовать функцию shinyApp() для запуска веб-приложения с определенным пользовательским интерфейсом и серверной логикой.

Примерный код для создания базовой интерактивной карты с использованием пакетов leaflet и Shiny:

```
library(shiny)
library(leaflet)

# Подготовка данных с координатами российских городов
data <- data.frame(city = c("Москва",
"Санкт-Петербург", "Екатеринбург"),
                   lon = c(37.617635, 30.315868,
60.605702),
                   lat = c(55.755826, 59.939095,
56.838011))
```

```
ui <- fluidPage(
  titlePanel("Интерактивная карта с Leaflet и Shiny"),
  leafletOutput("map")
)

server <- function(input, output, session) {
  output$map <- renderLeaflet({
    leaflet() %>%
      setView(lng = 60, lat = 55, zoom = 4) %>%
      addTiles() %>%
      addMarkers(data = data, popup = ~city)
  })
}

shinyApp(ui = ui, server = server)
```

Данный код создает интерактивную карту с маркерами для трех российских городов: Москвы, Санкт-Петербурга и Екатеринбурга. Карта центрируется на центральной части России с масштабом зума 4. На карте добавляются маркеры для каждого города с соответствующим названием в информационном окне.

Этот алгоритм поможет создать базовую интерактивную карту любой сложности с использованием пакетов leaflet и Shiny в среде R. Есть возможность дополнить этот код, добавив другие элементы и функциональности для создания более сложных карт. Далее рассмотрим базовые функции для работы с картами, использованные в том числе в примерах выше.

10.2.1. Базовые функции для работы с картами

Функция setView() в пакете leaflet используется для установки начальных координат (долготы и широты) и уровня масштабирования (zoom) карты.

setView(lng, lat, zoom) устанавливает центр карты по заданным координатам: lng (долгота) и lat (широта). Параметр zoom определяет уровень масштабирования карты. Чем выше значение zoom, тем больше будет увеличение карты.

Пример использования функции setView():

```
leaflet() %>%
  setView(lng = 37.617635, lat = 55.755826, zoom = 10)
```

Здесь функция setView() устанавливает центр карты в Москве (долгота 37.617635, широта 55.755826) с уровнем масштабирования 10. При запуске карта будет отображаться с заданными параметрами центра и масштаба.

Функция setView() является одной из ключевых функций в пакете leaflet, позволяющей настроить начальное положение карты. Она помогает пользователям определить, какая область карты будет отображаться при загрузке веб-приложения.

Функция addTiles() в пакете leaflet используется для добавления базовых тайлов (картографических слоев) на карту. Тайлы представляют собой фоновые изображения карт, обеспечивающих контекст и ориентацию на карте. Leaflet предоставляет несколько встроенных поставщиков тайлов, таких как OpenStreetMap, Stamen, Mapbox и другие. Можно выбрать нужный поставщик тайлов, указав его имя внутри функции addTiles(), либо использовать значение по умолчанию.

Пример использования addTiles() с тайлами OpenStreetMap:

```
leaflet() %>%
  addTiles()
```

В этом примере функция addTiles() добавляет базовый слой тайлов OpenStreetMap на карту. При запуске появятся фоновые изображения карты OpenStreetMap.

Также есть возможность настроить внешний вид тайлов, указав дополнительные параметры в функции addTiles(), такие как urlTemplate (URL-шаблон тайлов), options (опции отображения тайлов) и другие.

В целом, функция addTiles() является важной частью создания интерактивных карт с помощью пакета leaflet, так как обеспечивает базовый фоновый слой для отображения данных на карте. Работа с addTiles() позволяет пользователю выбирать подходящий стиль и контекст карты для их визуализации.

10.2.2. Добавление маркеров на карту

Функция addMarkers() в пакете leaflet используется для добавления маркеров на интерактивную карту. Маркеры представляют собой точечные обозначения на карте, которые могут содержать информацию и быть связаны с интерактивными всплывающими окнами. Можно указать координаты каждого маркера, текст или HTML-код для всплывающего окна (popup), а также другие параметры стиля и взаимодействия с маркерами.

Пример использования addMarkers() для добавления маркеров на карту:

```
library(shiny)
library(leaflet)

# Подготовка данных с координатами российских городов
data <- data.frame(
  city = c("Москва", "Санкт-Петербург", "Екатеринбург"),
  lon = c(37.617635, 30.315868, 60.605702),
  lat = c(55.755826, 59.939095, 56.838011)
)

# UI для веб-интерфейса Shiny
ui <- fluidPage(
  titlePanel("Интерактивная карта с маркерами
российских городов"),
  leafletOutput("map")
)

# Серверная часть для веб-интерфейса Shiny
server <- function(input, output, session) {
  output$map <- renderLeaflet({
    leaflet(data = data) %>%
      addTiles() %>%
      addMarkers(popup = ~city)
  })
}

# Запуск веб-приложения Shiny
shinyApp(ui = ui, server = server)
```

В примере выше создается интерактивная карта с тремя маркерами для городов Москва, Санкт-Петербург и Екатеринбург. Функция addTiles() добавляет базовый слой тайлов на карту, а функция addMarkers() добавляет маркеры с указанными координатами и названиями городов во всплывающих окнах.

Функция addMarkers() в пакете leaflet имеет несколько атрибутов, которые позволяют настраивать внешний вид и поведение маркеров на карте. Некоторые основные атрибуты функции addMarkers() и их описание:

– **lng** и **lat:** эти атрибуты указывают на долготу и широту местоположения маркера, соответственно. Маркеры будут размещены на карте в указанных координатах;

– **popup:** позволяет задать текст или HTML-код, который будет отображаться во всплывающем окне при клике на маркер. Можно использовать переменные из данных с помощью символа ~;

– **label:** задает текст, отображающийся рядом с маркером. Этот текст будет виден всегда, без необходимости клика на маркер;

– **icon:** данный атрибут используется для настройки иконки маркера. Применяются стандартные или собственные иконки;

– **clusterOptions:** позволяет создавать кластеры маркеров, если на карте присутствует большое количество маркеров. Это улучшает производительность и визуальное отображение карты;

– **options:** применяется для настройки различных параметров стиля и взаимодействия с маркерами, таких как цвет, размер, прозрачность и другие.

Пример использования атрибутов функции addMarkers():

```
library(shiny)
library(leaflet)

# Подготовка данных с координатами российских городов
data <- data.frame(
  city = c("Москва", "Санкт-Петербург", "Екатеринбург"),
  lon = c(37.617635, 30.315868, 60.605702),
  lat = c(55.755826, 59.939095, 56.838011)
)

# UI для веб-интерфейса Shiny
ui <- fluidPage(
  titlePanel("Интерактивная карта с маркерами российских
городов"),
  leafletOutput("map")
)

# Серверная часть для веб-интерфейса Shiny
server <- function(input, output, session) {
  output$map <- renderLeaflet({
    leaflet(data = data) %>%
      addTiles() %>%
      addMarkers(lng = 30.315868, lat = 59.939095,
popup = "Санкт-Петербург", label = "СПБ",
                 icon = leaflet::makeIcon(iconUrl =
"https://tattoo-stickers.ru/40016-thickbox_default/
razvodnye-mosty-belye-nochi-v-sankt-peterburge.jpg",
iconWidth = 30, iconHeight = 30),
                 options = markerOptions
(iconSize = c(100, 100), iconAnchor = c(45, 45))
      )

  })
}

# Запуск веб-приложения Shiny
shinyApp(ui = ui, server = server)
```

Это пример кода, который добавляет маркер на карту с атрибутами popup, label, icon и options. Есть возможность настраивать данные атрибуты в соответствии с вашими потребностями для создания интерактивных и информативных карт.

Работа с addMarkers() позволяет визуализировать и выделить конкретные точки или объекты на карте, делая ее более информативной и интерактивной для пользователей.

10.2.3. Добавление полигонов и линий

Полигон в географическом контексте – это плоская геометрическая фигура, ограниченная замкнутой ломаной линией, состоящей из конечного числа отрезков. Это многоугольник, который определяется тремя или более точками (вершинами) на плоскости, соединенными прямыми линиями (сторонами). Полигоны используются в GIS (геоинформационных системах), таких как Leaflet, для представления ареалов, например, границ городов, парков, водоемов и других земельных участков.

В контексте программирования карт с использованием Leaflet, полигон – это объект, который можно создать и добавить на карту для изображения различных географических территорий. Он может быть задан массивом координат, который определяет вершины полигона. Пользователи могут видеть полигон на карте, взаимодействовать с ним, например, кликать на него, чтобы получить больше информации, изменять его стиль и т. д.

Каждый полигон имеет различные стилевые настройки, такие как цвет заливки, прозрачность, цвет границы, что делает его полезным инструментом для визуализации и разграничения географических областей на интерактивной карте.

Функция addPolygons() в пакете Leaflet для R используется для добавления полигонов на карту. Рассмотрим, как могут быть использованы базовые параметры функции addPolygons() вместе с пакетом leaflet в приложении Shiny:

```
library(shiny)
library(leaflet)

ui <- fluidPage(
  titlePanel("Интерактивная карта с полигонами"),
  leafletOutput("mymap")
)

server <- function(input, output) {
```

```
  # Предположим, у нас уже есть данные для полигонов
  polygons_data <- data.frame(
    lat = c(55.7558, 55.7558, 55.7522, 55.7522),
    lng = c(37.6173, 37.6233, 37.6233, 37.6173)
  )

  output$mymap <- renderLeaflet({
    leaflet() %>%
      addTiles() %>%
      addPolygons(
        data = polygons_data,
        lng = ~lng,
        lat = ~lat,
        fillColor = "blue",
        color = "#0000FF",
        fillOpacity = 0.5,
        opacity = 1.0,
        weight = 2,
        smoothFactor = 0.5
      )
  })
}

shinyApp(ui, server)
```

– **data** содержит данные полигона, обычно, это data.frame или другой объект, содержащий координаты широты и долготы каждой вершины полигона;

– **fillColor** задает цвет заливки полигонов;

– **color** определяет цвет линии границ полигона;

– **fillOpacity** задает уровень прозрачности заливки полигона;

– **weight** определяет толщину линий границ полигона;

– **smoothFactor** контролирует степень сглаживания углов полигона;

– опция **highlight** позволяет изменить стиль полигона при наведении на него курсора;

– **label** позволяет добавить текстовые метки на полигоны.

В приведенном примере создается базовое приложение Shiny с одним пользовательским интерфейсом ui, который содержит название и вывод карты. Серверная часть server генерирует карту с тайлами OpenStreetMap и добавляет на нее полигон на основе предопределенного набора координат с помощью addPolygons(). Здесь используется текст-заполнитель для данных в polygons_data, который можно легко заменить реальным набором данных о полигонах. На карте будет отображаться полигон с заданными параметрами отображения.

Функция addPolylines() в пакете leaflet для R используется для добавления линий и маршрутов на интерактивные карты. Это полезный инструмент для визуализации путей, маршрутов или любых других соединений между точками на карте.

Основное использование addPolylines() включает в себя следующие аргументы:

- **data:** набор данных, содержащий координаты точек линии;
- **lng и lat:** поля или вычисления, указывающие долготу и широту каждой точки линии соответственно;
- **color:** цвет линии;
- **weight:** толщина линии;
- **opacity:** прозрачность линии;
- **другие опции** для настройки стиля линии, такие как dashArray для создания пунктирной линии и smoothFactor для сглаживания линий.

Код для добавления линий в приложении Shiny с использованием leaflet может выглядеть следующим образом:

```
library(shiny)
library(leaflet)

ui <- fluidPage(
  titlePanel("Интерактивная карта с линиями"),
  leafletOutput("mymap")
)

server <- function(input, output) {

  # Данные линий
  lines_data <- data.frame(
    lat = c(55.7522, 55.7601),
    lng = c(37.6156, 37.6184)
  )

  output$mymap <- renderLeaflet({
    leaflet() %>%
      addTiles() %>%
      addPolylines(
        data = lines_data,
        lng = ~lng,
        lat = ~lat,
        color = "red",
        weight = 2,
        opacity = 0.5
      )
  })
}

shinyApp(ui, server)
```

В этом коде addTiles() используется для добавления стандартного слоя карты. Затем функция addPolylines() использует данные lines_data для отрисовки линии, соединяющей точки с заданной широтой и долготой. Полученная карта будет интерактивной и отобразит заданную линию между двумя точками в приложении Shiny.

10.2.4. Добавление информационных слоев

В Leaflet информационные слои (или контрольные слои) используются для предоставления пользователю возможности включать и выключать отображение определенных слоев на карте, например, различных видов тайлов для карты, маркеров, полигонов или линий. Это позволяет создать интерактивное взаимодействие, где пользователь сам может выбирать, какую информацию он хочет видеть.

addLayersControl()

Добавление информационных слоев в приложение Shiny с использованием Leaflet может быть выполнено с помощью функции addLayersControl(). Расширим предыдущий пример, добавив информационные слои для полигонов и линий, а также предоставим пользователям возможность включать и выключать эти слои:

```
library(shiny)
library(leaflet)

ui <- fluidPage(
  titlePanel("Интерактивная карта с информационными
слоями"),
  leafletOutput("mymap")
)

server <- function(input, output) {

  # Данные для полигонов
  polygons_data <- data.frame(
    lat = c(55.7558, 55.7558, 55.7522, 55.7522),
    lng = c(37.6173, 37.6233, 37.6233, 37.6173)
  )

  # Данные для линий
  lines_data <- data.frame(
    lat = c(55.7522, 55.7601),
    lng = c(37.6156, 37.6184)
  )
```

```
  output$mymap <- renderLeaflet({
    leaflet() %>%
      addTiles() %>%
      addPolygons(
        data = polygons_data,
        lng = ~lng,
        lat = ~lat,
        fillColor = "blue",
        color = "#0000FF",
        fillOpacity = 0.5,
        opacity = 1.0,
        weight = 2,
        smoothFactor = 0.5,
        group = "Полигоны"
      ) %>%
      addPolylines(
        data = lines_data,
        lng = ~lng,
        lat = ~lat,
        color = "red",
        weight = 2,
        opacity = 0.5,
        group = "Линии"
      ) %>%
      addLayersControl(
        overlayGroups = c("Полигоны", "Линии"),
        options = layersControlOptions
(collapsed = FALSE)
      )
  })
}

shinyApp(ui, server)
```

В примере выше есть два информационных слоя: один для полигонов, а другой для линий. Каждый слой помечен с помощью аргумента group, что позволяет функции addLayersControl() использовать эти группы для создания интерфейса управления слоями. Легенда карты будет автоматически добавлена на карту, и пользователи смогут выбирать, какие слои они хотят видеть.

Стоить отметить важный момент: в несложных примерах данные зачастую задаются напрямую в коде, но в реальных проектах они часто извлекаются из файлов или баз данных.

Для разработки приложения Shiny с использованием базы данных SQLite потребуется структура из нескольких файлов. Создадим простое приложение, которое позволяет пользователю визуализировать

полигоны и линии на карте, данные для которых сохранены в базе данных. Мы разобьем код на три файла: global.R, ui.R и server.R.

Структура каталогов проекта:

my_shiny_app/
|-- global.R
|-- ui.R
|-- server.R
|-- geodata.db

Файл global.R содержит общий код, который будет использоваться и серверной, и клиентской частью приложения. В основном, это библиотеки и код, связанный с базой данных.

global.R:

```
library(shiny)
library(leaflet)
library(DBI)
library(RSQLite)

# Переменная для подключения к базе данных
# Предполагаем, что база данных 'geodata.db' уже
создана и заполнена данными
db <- dbConnect(RSQLite::SQLite(), "geodata.db")
```

Файл ui.R описывает структуру интерфейса пользователя.

ui.R:

```
ui <- fluidPage(
  titlePanel("Интерактивная карта с полигонали
и линиями"),
  leafletOutput("mymap")
)
```

Файл server.R содержит логику серверной части приложения, работу с базой данных и рендеринг карты.

server.R:

```
source("global.R")

server <- function(input, output, session) {

  # Извлекаем данные для полигоналей и линий
  polygons_data <- dbReadTable(db, "polygons")
  lines_data <- dbReadTable(db, "lines")
```

```
  output$mymap <- renderLeaflet({
    leaflet() %>%
      addTiles() %>%
      addPolygons(
        data = polygons_data,
        fillColor = "blue",
        weight = 1,
        opacity = 1,
        color = "white",
        fillOpacity = 0.5
      ) %>%
      addPolylines(
        data = lines_data,
        color = "red",
        weight = 2,
        opacity = 0.5
      )
  })

  # Закрываем соединение с базой данных при
завершении работы приложения
  session$onSessionEnded(function() {
    dbDisconnect(db)
  })
}
```

Для запуска приложения Shiny с такой структурой файлов необходимо положить эти файлы в один каталог и убедиться, что в наличии файл базы данных geodata.db со следующей структурой:

Таблица 'polygons':
id | lat | lng

Таблица 'lines':
id | lat | lng

Для создания файла geodata.db с таблицами polygons и lines, содержащими координаты для простых фигур, которые представляют российские города, необходимо выполнить следующий код:

```
library(RSQLite)

# Создаем новое подключение к базе данных SQLite
con <- dbConnect(SQLite(), dbname = "geodata.db")
```

```
# Создаем таблицы 'polygons' и 'lines'
dbExecute(con, "CREATE TABLE polygons (id INTEGER
PRIMARY KEY, lat REAL, lng REAL)")
dbExecute(con, "CREATE TABLE lines (id INTEGER
PRIMARY KEY, lat REAL, lng REAL)")

# Вставляем примерные данные в таблицу 'polygons'
(координаты центральной точки какого-либо города)
# Здесь используются примерные координаты для Москвы
dbExecute(con, "INSERT INTO polygons (id, lat, lng)
VALUES (1, 55.7558, 37.6173)")

# Вставляем примерные данные в таблицу 'lines'
(можно использовать для представления дорог)
# Эти данные полностью условны и не соответствуют
реальным дорогам
dbExecute(con, "INSERT INTO lines (id, lat, lng)
VALUES (1, 55.7558, 37.6176)")

# Проверяем вставку данных, выполняем запрос SELECT
polygons_data <- dbGetQuery(con, "SELECT * FROM
polygons")
lines_data <- dbGetQuery(con, "SELECT * FROM lines")

# Смотрим, что получилось
print(polygons_data)
print(lines_data)

# Отключаемся от базы данных
dbDisconnect(con)
```

После выполнения кода в рабочем каталоге появится файл geodata.db, содержащий таблицы polygons и lines с примерными данными. Есть возможность расширить эту базу данных, добавив дополнительные реальные данные о координатах российских городов.

Затем необходимо запустить приложение с помощью команды shiny::runApp(«путь_к_вашему_каталогу/my_shiny_app»). Shiny автоматически подхватит файлы ui.R и server.R и запустит приложение. Файл global.R будет вызван первым, чтобы установить все необходимые глобальные настройки, такие как подключение к базе данных и загрузка библиотек.

addCircleMarkers(), addPopups(), addGeoJSON()

addCircleMarkers() добавляет на карту круговые маркеры, которые обычно используются для обозначения точек на карте. Размер и цвет кругов можно настроить для передачи дополнительной информации.

Пример использования:

```
    leaflet(data = df) %>%
      addTiles() %>%
      addCircleMarkers(lng = ~Longitude, lat = ~Latitude,
radius = ~SizeVariable)
```

Здесь df представляет собой dataframe, содержащий столбцы Longitude, Latitude и опционально SizeVariable, который может контролировать размеры маркеров.

addPopups() добавляет всплывающие окна, которые могут быть прикреплены к конкретным объектам или координатам на карте. Когда пользователь кликает или наводит курсор на объект карты, может появиться всплывающее окно с дополнительной информацией.

Пример использования:

```
    leaflet(data = df) %>%
      addTiles() %>%
      addMarkers(lng = ~Longitude, lat = ~Latitude) %>%
      addPopups(lng = ~Longitude, lat = ~Latitude,
popup = ~PopupText)
```

В этом примере df – это dataframe с координатами и текстом для всплывающих окон в столбце PopupText.

addGeoJSON() используется для добавления GeoJSON объектов на карту. GeoJSON – это формат обмена геоданными, использующийся для кодирования различных структур данных, таких как точки (для местоположений), линии (для улиц, шоссе и границ) и многоугольники (для областей, таких как страны, внутренние границы, округа и т. д.).

Пример использования:

```
    leaflet() %>%
      addTiles() %>%
      addGeoJSON(geojson)
```

Тут geojson – это переменная, содержащая GeoJSON. данные, которые могут быть загружены из файла или напрямую заданы в коде в виде строки или списка в формате JSON.

Создадим пример интерактивной карты с использованием библиотеки leaflet в R, которая объединяет addCircleMarkers(), addPopups()

и addGeoJSON(). Будем отображать круговые маркеры на карте, при клике на которые появятся всплывающие окна с информацией, а также добавим GeoJSON слой.

В приложении Shiny данные часто хранятся в виде локальных файлов, базы данных или иногда извлекаются непосредственно из интернета. Для примера создадим каталог с рядом файлов. Они разделены по своей функциональности для удобства организации кода и могут быть размещены следующим образом:

my_shiny_app/
|-- global.R
|-- ui.R
|-- server.R

Отдельные файлы имеют следующее назначение:

– **global.R:** содержит код, который используется как сервером, так и интерфейсом пользователя. Как правило, это библиотеки, инициализация подключения к базам данных, общие переменные;

– **ui.R:** содержит код описания интерфейса пользователя, в нем создаются различные UI-элементы (например, карты, панели, таблицы);

– **server.R:** содержит серверную логику приложения. Это могут быть функции рендеринга, обработки ввода пользователя и взаимодействия с данными.

Рассмотрим, как можно загрузить данные из локальных файлов в global.R.

global.R:

```
library(shiny)
library(leaflet)
library(jsonlite)
library(readr)
# Подготовим данные в глобальном окружении
# Путь к данным маркеров, файл .csv
markers_path <- "data/markers_data.csv"
# Путь к данным GeoJSON, файл .geojson в папке www
geojson_path <- "www/geojson_data.geojson"
# Чтение данных маркеров в датафрейм из CSV файла
markers_data <- read_csv(markers_path)
# Чтение данных GeoJSON в список из файла GeoJSON
geojson_data <- fromJSON(geojson_path,
simplifyVector = FALSE)
```

При запуске приложения Shiny важно убедиться, что установлены самые последние версии всех необходимых пакетов и что файлы данных (geojson_data.geojson и markers_data.csv) находятся в их соответствующих местах в каталогах приложения.

Предположим, что имеются следующие данные:

```
# Пример данных для круговых маркеров и всплывающих окон
markers_data <- data.frame(
  id = 1:2,
  lat = c(55.7558, 59.9343),
  lng = c(37.6173, 30.3351),
  info = c("Москва, Россия", "Санкт-Петербург, Россия")
)

# Пример GeoJSON слоя (здесь упрощенный пример;
в реальности GeoJSON будет сложнее)
geojson_data <- '{"type": "FeatureCollection",
"features": [
  {"type": "Feature", "properties": {}, "geometry": {
    "type": "LineString", "coordinates": [[37.6173,
55.7558], [30.3351, 59.9343]]
  }}
]}'
```

Теперь создадим приложение Shiny, которое отобразит эти данные.

ui.R:

```
library(shiny)
library(leaflet)

ui <- fluidPage(
  titlePanel("Интерактивная карта России"),
  leafletOutput("mymap")
)
```

server.R:

```
source("global.R")

server <- function(input, output) {
```

```
   output$mymap <- renderLeaflet({
     leaflet() %>%
       addTiles() %>%
       addCircleMarkers(
         data = markers_data,
         lng = ~lng,
         lat = ~lat,
         radius = 6,
         color = "#0078FF",
         stroke = TRUE,
         fillOpacity = 0.8
       ) %>%
       addPopups(
         lng = ~lng,
         lat = ~lat,
         popup = ~info,
         options = popupOptions(closeButton = TRUE)
       ) %>%
       addGeoJSON(geojson_data) # Добавляем GeoJSON
 слой с линией
   })
 }

 shinyApp(ui, server)
```

global.R:

```
library(shiny)
library(leaflet)

# Подготовим данные в глобальном окружении
# Данные для круговых маркеров и всплывающих окон
markers_data <- data.frame(
  id = 1:2,
  lat = c(55.7558, 59.9343),
  lng = c(37.6173, 30.3351),
  info = c("Москва, Россия", "Санкт-Петербург,
Россия")
)


# Данные GeoJSON
geojson_data <- jsonlite::fromJSON('{"type":
"FeatureCollection", "features": [
  {"type": "Feature", "properties": {}, "geometry": {
    "type": "LineString", "coordinates":
[[37.6173, 55.7558], [30.3351, 59.9343]]
  }}
]}', simplifyVector = FALSE)
```

Чтобы запустить данное приложение Shiny, нужно сохранить каждый из этих файлов в одну директорию и запустить файл server.R из среды R. Убедиться, что установлены все необходимые пакеты (shiny, leaflet, jsonlite) и подключение к интернету активно, так как leaflet будет загружать тайлы карты онлайн.

Пример будет работать, если он запускается на локальной машине. В реальной ситуации данные для GeoJSON следует загружать из внешнего файла или источника данных. В клетке global.R используется jsonlite::fromJSON для преобразования строки GeoJSON в список R, который можно использовать с addGeoJSON().

10.3. Конфигурация взаимодействия с картой

Конфигурация взаимодействия с картой в контексте веб-карт и геоинформационных систем – это процесс настройки различных параметров и функций, определяющих способы взаимодействия пользователя с картой. Включает в себя управление масштабированием, перемещением, маркерами, элементами управления и другими аспектами картографического интерфейса.

Приведем некоторые примеры конфигурации взаимодействия с картой и их назначение.

1. **Zoom Control** (элемент управления масштабированием): увеличивает или уменьшает масштаб карты. Настройка этого элемента позволяет определить, будет ли он отображаться на карте или будет скрыт.

2. **Scroll Wheel Zoom** (масштабирование колесом мыши): функция позволяет пользователю масштабировать карту, используя колесо мыши. Настройка этой функции позволяет включить или отключить этот способ масштабирования.

3. **Dragging** (перемещение карты): пользователь может перемещать карту, удерживая левую кнопку мыши и перетаскивая карту. Настройка этой функции позволяет включить или отключить перемещение карты.

4. **Double Click Zoom** (масштабирование двойным щелчком мыши): пользователь может увеличивать масштаб карты, дважды щелкнув по ней. Настройка функции позволяет включить или отключить данный способ масштабирования.

5. **Fullscreen Control** (полноэкранный режим): элемент управления, который дает возможность пользователю переключаться в полноэкранный режим для более удобного просмотра карты.

Функция addZoomControl() в пакете leaflet используется для добавления элемента управления масштабированием на карту. Этот элемент предоставляет пользователям возможность увеличивать и уменьшать масштаб карты. Рассмотрим пример кода, который демонстрирует использование функции addZoomControl() вместе с библиотекой Shiny:

```
library(shiny)
library(leaflet)

ui <- fluidPage(
  titlePanel("Пример карты с элементом управления
масштабированием"),
  leafletOutput("map")
)

server <- function(input, output, session) {
  output$map <- renderLeaflet({
    leaflet() %>%
      addTiles() %>%
      addMarkers(lng = 30.315868, lat = 59.939095) %>%
      addZoomControl()
  })
}

shinyApp(ui = ui, server = server)
```

Для использования функции addZoomControl() необходимо поместить ее после добавления тайлов и маркеров на карту, что отобразит элемент управления масштабированием на карте.

Функция addScaleBar() в пакете leaflet применяется для добавления элемента управления масштабом на карту. Элемент показывает масштаб карты в единицах измерения (например, километры или мили) и обеспечивает пользователей информацией о масштабе карты. Пример кода, который демонстрирует использование функции addScaleBar() вместе с библиотекой Shiny:

```
library(shiny)
library(leaflet)

ui <- fluidPage(
  titlePanel("Пример карты с масштабной линейкой"),
  leafletOutput("map")
)
```

```
server <- function(input, output, session) {
  output$map <- renderLeaflet({
    leaflet() %>%
      addTiles() %>%
      addMarkers(lng = 30.315868, lat = 59.939095) %>%
      addScaleBar(position = "bottomright")
  })
}

shinyApp(ui = ui, server = server)
```

Код показывает процесс создания веб-приложения Shiny, которое отображает интерактивную карту с элементом управления масштабом. При запуске кода в среде R (например, RStudio) появится карта с элементом управления масштабом в нижнем правом углу. Для использования функции addScaleBar() нужно добавить ее после тайлов и маркеров на карту. Пример поможет понять, как использовать addScaleBar() вместе с библиотекой Shiny для создания интерактивной карты с элементом управления масштабом.

Функция addLayersControl() принимает параметры overlayGroups, определяющие группы слоев, которые могут быть переключены, и options, позволяющие настроить поведение элемента управления слоями.

Конфигурация взаимодействия с картой в пакете leaflet для R определяет различные аспекты поведения и функциональности карты, такие как масштабирование, перемещение, управление элементами управления и другие важные функции. Пример кода, демонстрирующий конфигурацию взаимодействия с картой в пакете leaflet с использованием библиотеки Shiny:

```
library(shiny)
library(leaflet)

ui <- fluidPage(
  titlePanel("Пример взаимодействия с картой"),
  leafletOutput("map")
)

server <- function(input, output, session) {
  output$map <- renderLeaflet({
    leaflet() %>%
      addTiles() %>%  # Добавляем тайлы карты
      addMarkers(lng = 30.315868, lat = 59.939095) %>%  #
Добавляем маркеры
      setView(lng = 30.315868, lat = 59.939095, zoom = 10)
%>%  # Устанавливаем начальное положение и масштаб карты
```

```
        addZoomControl() %>%  # Добавляем элемент управления
  масштабированием
        setOptions(scrollWheelZoom = TRUE) %>%  # Включаем
  масштабирование колесом мыши
        setOptions(dragging = TRUE) %>%  # Включаем
  перемещение карты
        setOptions(doubleClickZoom = TRUE) %>%  # Включаем
  масштабирование двойным щелчком мыши
        addFullscreenControl()  # Добавляем элемент
  управления для полноэкранного режима
    })
  }

  shinyApp(ui = ui, server = server)
```

Объяснение кода:

– **addTiles():** добавляет стандартные тайлы карты на карту;

– **addMarkers():** помещает маркер на карту с указанными координатами;

– **setView():** устанавливает начальное положение и масштаб карты;

– **addZoomControl():** добавляет элемент управления масштабированием на карту;

– **setOptions(scrollWheelZoom = TRUE):** включает масштабирование карты с помощью колеса мыши;

– **setOptions(dragging = TRUE):** включает перемещение карты с помощью мыши;

– **setOptions(doubleClickZoom = TRUE):** запускает масштабирование карты при двойном щелчке мыши;

– **addFullscreenControl():** добавляет элемент управления для перехода в полноэкранный режим.

Приведенный пример демонстрирует различные аспекты взаимодействия с картой, такие как масштабирование, перемещение, управление элементами управления и другие функции. Можно настраивать эти параметры в соответствии с конкретными потребностями и добавлять дополнительные функции взаимодействия с картой по мере необходимости.

Конфигурация взаимодействия с картой важна для создания удобного и интуитивно понятного интерфейса для пользователей, работающих с географическими данными. Путем настройки различных параметров и функций можно оптимизировать пользовательский опыт и обеспечить более эффективное использование картографического приложения.

Глава 11
СОЗДАНИЕ ИНТЕРАКТИВНОГО ВЕБ-ПРИЛОЖЕНИЯ С SHINY

Часть 1

Рассмотрим создание приложения на R с использованием библиотеки Shiny для отображения пожаров на территории Свердловской области. Краткое руководство, которое описывает, как поэтапно можно разработать такое приложение.

1. **Установка необходимых пакетов:**
   - shiny для создания интерактивного веб-приложения;
   - shinydashboard для более удобного и привлекательного интерфейса;
   - leaflet для отображения карт;
   - RSQLite для взаимодействия с SQLite базой данных, где можно хранить данные о пожарах.
2. **Создание базы данных пожаров:** создайте SQLite базу данных с таблицей, содержащей следующие поля: ID пожара, дата и время, широта, долгота, название местности, тип местности, степень угрозы, прогнозируемая зона распространения и дополнительные примечания.
3. **Создание серверной части приложения:**
   - настройте серверное взаимодействие с базой данных;
   - разработайте систему предсказания пожаров, используя, например, исторические данные и алгоритмы машинного обучения;
   - используйте leaflet для отображения интерактивной карты с маркерами пожаров.
4. **Создание пользовательского интерфейса:** с помощью shinydashboard сформируйте панель управления с картой и элементами управления для фильтрации и отображения данных.

Пример приложения:

```
library(shiny)
library(shinydashboard)
library(leaflet)
library(DBI)
library(RSQLite)
```

```
# Создаем фейковую базу данных
create_fake_fire_db <- function() {
  conn <- dbConnect(RSQLite::SQLite(), ":memory:")
  dbWriteTable(conn, "fires", data.frame(
    id = 1:25,
    datetime = seq.POSIXt(from = as.POSIXct("2023-01-01
00:00:00"), by = "days", length.out = 25),
    lat = runif(25, 56, 57),
    lon = runif(25, 60, 61),
    location_name = paste("Локация", 1:25),
    type = sample(c("Лесной", "Торфяник", "Степной"), 25,
replace=TRUE),
    threat_level = sample(c("Низкая", "Средняя",
"Высокая"), 25, replace=TRUE),
    prediction_zone = runif(25, 0, 5),
    notes = replicate(25, paste(sample(c("Примечание",
"Требуется наблюдение", "Важно"), 5, replace=TRUE),
collapse=" "))
  ))
  return(conn)
}

# Создаем приложение
ui <- dashboardPage(
  dashboardHeader(title = "Карта Пожаров Свердловской
Области"),
  dashboardSidebar(),
  dashboardBody(
    leafletOutput("fireMap", height = 600)
  )
)

server <- function(input, output) {
  db <- create_fake_fire_db()

  output$fireMap <- renderLeaflet({
    fires <- dbReadTable(db, "fires")

    leaflet(data = fires) %>%
      addTiles() %>%
      addMarkers(~lon, ~lat, popup = ~paste(location_name,
"<br>", type, "<br>", threat_level, "<br>", notes))
  })
}

# Запуск приложения
shinyApp(ui, server)
```

Данный пример кода на языке R состоит из нескольких частей, каждая из которых выполняет определенную функцию в контексте создания интерактивного веб-приложения с помощью библиотеки Shiny для отображения и анализа данных о пожарах на территории Свердловской области. Шаг за шагом объяснение того, что происходит в коде.

1. **Подключение библиотек:** загружаются необходимые пакеты для работы с Shiny, интерактивными картами и базой данных:

– shiny для создания интерактивного веб-приложения;

– shinydashboard для создания структурированного и красивого пользовательского интерфейса;

– leaflet для добавления карт и работы с географическими данными;

– DBI и RSQLite для работы с базами данных SQLite.

2. **Функция создания искусственной базы данных**

create_fake_fire_db – данная функция создает базу данных SQLite на «лету» (в памяти) и заполняет ее случайными данными о пожарах для демонстрационных целей. В базе есть следующие поля: ID пожара, время, координаты (широта и долгота), название локации, тип местности, уровень угрозы, прогнозируемая зона распространения и дополнительные заметки.

3. **Определение интерфейса пользователя**

Описывается внешний вид приложения с помощью функций dashboardPage, dashboardHeader, dashboardSidebar, dashboardBody и leafletOutput. В заголовке страницы задается название приложения. В основной части (dashboardBody) располагается элемент leafletOutput для отображения карты.

4. **Создание серверной части приложения**

В функции server происходит:

– загрузка базы данных созданной функцией create_fake_fire_db;

– отрисовка интерактивной карты с помощью leaflet, с использованием данных из базы данных. На карту добавляются маркеры (addMarkers), соответствующие координатам местоположений пожаров. Всплывающие окна (popup) каждого маркера отображают информацию о пожарах: название локации, тип местности, уровень угрозы и заметки.

5. **Запуск приложения:** с помощью shinyApp(ui, server) приложение объединяет пользовательский интерфейс (UI) и серверную часть, после чего оно готово к запуску.

Код содержит весь необходимый минимум для разработки простого интерактивного веб-приложения, которое можно расширять и модифицировать, добавляя реальные данные.

Часть 2

Улучшим приложение, добавив несколько полезных и интерактивных функций.

1. **Фильтрация данных по уровню угрозы:** пользователи смогут выбирать пожары для отображения на карте по уровню угрозы.

2. **Отображение деталей:** когда пользователь кликнет на маркер, откроется более подробное описание пожара.

3. **Обновление карты в реальном времени:** добавим кнопку, которая обновит данные на карте.

4. **Добавление пользовательского ввода:** пользователи смогут добавлять информацию о новых пожарах.

```
library(shiny)
library(shinydashboard)
library(leaflet)
library(DBI)
library(RSQLite)

# Создаем фейковую базу данных
create_fake_fire_db <- function() {
  conn <- dbConnect(RSQLite::SQLite(), ":memory:")
  dbWriteTable(conn, "fires", data.frame(
    id = 1:25,
    datetime = seq.POSIXt(from = as.POSIXct("2023-01-01
00:00:00"), by = "days", length.out = 25),
    lat = runif(25, 56, 57),
    lon = runif(25, 60, 61),
    location_name = paste("Локация", 1:25),
    type = sample(c("Лесной", "Торфяник", "Степной"),
25, replace=TRUE),
    threat_level = sample(c("Низкая", "Средняя",
"Высокая"), 25, replace=TRUE),
    prediction_zone = runif(25, 0, 5),
    notes = replicate(25, paste(sample(c("Примечание",
"Требуется наблюдение", "Важно"), 5, replace=TRUE),
collapse=" "))
  ))
  return(conn)
}

# Создаем приложение
ui <- dashboardPage(
  dashboardHeader(title = "Карта Пожаров Свердловской
Области"),
  dashboardSidebar(
    selectInput("threatLevel", "Уровень угрозы:",
```

```
                    choices = c("Все", "Низкая", "Средняя",
"Высокая")),
      actionButton("update", "Обновить карту"),
      textInput("locationName", "Название местности"),
      actionButton("addFire", "Добавить информацию
о пожаре")
      # Можно добавить другие элементы для ввода данных
о пожаре
    ),
    dashboardBody(
      leafletOutput("fireMap", height = 600)
    )
  )

  server <- function(input, output) {
    db <- create_fake_fire_db()

    data <- reactive({
      fires <- dbReadTable(db, "fires")
      if (input$threatLevel != "Все") {
        fires <- fires[fires$threat_level == input$threat-
Level, ]
      }
      fires
    })

    observeEvent(input$update, {
      output$fireMap <- renderLeaflet({
        leaflet(data()) %>%
          addTiles() %>%
          addMarkers(~lon, ~lat, popup = ~paste(loca-
tion_name, "<br>", type, "<br>", threat_level, "<br>",
notes))
      })
    })

    observeEvent(input$addFire, {
    })

    output$fireMap <- renderLeaflet({
      leaflet(data()) %>%
        addTiles() %>%
        addMarkers(~lon, ~lat, popup = ~paste(loca-
tion_name, "<br>", type, "<br>", threat_level, "<br>",
notes))
    })
  }

  # Запуск приложения
  shinyApp(ui, server)
```

В коде был добавлен выбор уровня угрозы, позволяющий фильтровать данные на карте. Кнопка «Обновить карту» перерисовывает карту в соответствии с выбранным фильтром. Также присутствует возможность добавления новой информации о пожаре через текстовое поле и кнопку «Добавить информацию о пожаре». Новые данные можно добавлять в базу данных, но функционал этой возможности нужно реализовать дополнительно.

Часть 3

Снова улучшим приложение, добавив некоторые интерактивные возможности и функционал для повышения удобства пользователей:

– фильтрацию данных на карте в реальном времени;

– возможность удаления данных о пожаре;

– добавление автоматического определения текущего времени при добавлении нового пожара;

– расширенное оформление всплывающих подсказок;

– отображение текущего состояния загрузки данных на карту;

– добавление точек на карту;

– обновление карты;

– добавление комментариев: поможет разработчикам и пользователям понять логику каждого блока кода и упростит содержание и последующую поддержку приложения. Чтобы сделать код более понятным и информативным, были добавлены комментарии к различным блокам кода для объяснения их предназначения.

```
library(shiny)
library(shinydashboard)
library(leaflet)
library(DBI)
library(RSQLite)

# Функция для инициализации и создания базы данных
в памяти
# ================================================
=====
create_fake_fire_db <- function() {
  conn <- dbConnect(RSQLite::SQLite(), ":memory:")
  dbWriteTable(conn, "fires", data.frame(
# Прямо указываем все столбцы, которые будут сформированы
в таблице
    id = 1:25,
    datetime = seq.POSIXt(from = as.POSIXct("2023-01-01
00:00:00"), by = "days", length.out = 25),
```

```
    lat = runif(25, 56, 57),
    lon = runif(25, 60, 61),
    location_name = paste("Локация", 1:25),
    type = sample(c("Лесной", "Торфяник", "Степной"), 25,
replace = TRUE),
    threat_level = sample(c("Низкая", "Средняя",
"Высокая"), 25, replace = TRUE),
    prediction_zone = runif(25, 0, 1),
    notes = replicate(25, paste(sample(c("Примечание",
"Требуется наблюдение", "Важно"), 5, replace = TRUE),
collapse = " "))
  ))
  return(conn)
}

# Создаем пользовательский интерфейс
# ================================================
=====
ui <- dashboardPage(
  # Код для создания UI с заголовком, боковой панелью
с элементами управления и телом страницы с картой
  dashboardHeader(title = "Карта Пожаров Свердловской
Области"),
  # Боковая панель с элементами управления
  dashboardSidebar(
    # Элементы формы для взаимодействия с пользователем
    selectInput("threatLevel", "Уровень угрозы:",
choices = c("Все", "Низкая", "Средняя", "Высокая")),
    actionButton("update", "Обновить карту"),
    textInput("locationName", "Название местности",
value = ""),
    numericInput("latitude", "Широта", value = 56.8,
min = 55, max = 58),
    numericInput("longitude", "Долгота", value = 60.6,
min = 59, max = 62),
    textInput("notes", "Примечания", value = ""),
    actionButton("addFire", "Добавить информацию
о пожаре"),
    actionButton("deleteFire", "Удалить последний пожар"),
    uiOutput("formFeedback")
  ),
  # Основное тело приложения с картой
  dashboardBody(
    # Вывод карты
    leafletOutput("fireMap", height = 600),
    hr(),
    fluidRow(column(4, verbatimTextOutput("loadingState")))
  )
)
```

```
# Определение серверной логики приложения
# ========================================================
======
server <- function(input, output, session) {
  # Инициализация базы данных
  db <- create_fake_fire_db()

  # Объявление реактивных значений для хранения данных
из базы данных
  data <- reactiveVal(dbReadTable(db, "fires"))

  # Обновление карты в зависимости от выбранного уровня
угрозы
  observe({
    req(input$threatLevel)
    df <- data()

    if (input$threatLevel != "Все") {
      df <- df[df$threat_level == input$threatLevel, ]
    }
    # Рендеринг карты с текущими данными
    output$fireMap <- renderLeaflet({
      # Отображение маркеров на карте с использованием
библиотеки leaflet
      leaflet(df) %>%
        addTiles() %>%
        addCircleMarkers(~lon, ~lat, radius = ~predic-
tion_zone * 10,
                         color = ~ifelse(threat_level =
= "Высокая", "red", ifelse(threat_level == "Средняя",
"orange", "green")),
                         popup = ~paste("Местность:",
location_name, "<br>",
                                        "Тип:", type,
"<br>",
                                        "Уровень угрозы:",
threat_level, "<br>",
                                        "Примечания:",
notes))
    })
  })

  # Обработка события обновления данных на карте
  observeEvent(input$update, {
    # Перезагрузка данных из базы данных
    data(dbReadTable(db, "fires"))
  })

  # Определение обработки события добавления нового пожара
  observeEvent(input$addFire, {
```

```
    # Проверка входных данных и добавление записи
    if (is.na(input$latitude) || is.na(input$longitude) ||
        input$latitude < -90 || input$latitude > 90 ||
        input$longitude < -180 || input$longitude > 180) {
      output$formFeedback <- renderUI({
        div(style = "color: red;", "Ошибка: Введите
корректные координаты.")
      })
    } else {
      new_entry <- data.frame(
        id = max(data()$id, na.rm = TRUE) + 1,
        datetime = format(Sys.time(), "%Y-%m-%d %H:%M:%S"),
        lat = input$latitude,
        lon = input$longitude,
        location_name = input$locationName,
        type = "Новый",
        threat_level = "Неизвестно",
        prediction_zone = runif(1, 0, 1),
        notes = input$notes,
        stringsAsFactors = FALSE
      )
      dbWriteTable(db, "fires", new_entry, append = TRUE)
      data(dbReadTable(db, "fires"))
      output$formFeedback <- renderUI({
        div(style = "color: green;", "Пожар успешно
добавлен.")
      })
    }
  })

  # Определение обработки события удаления последнего
пожара
  observeEvent(input$deleteFire, {
    # Удаление последней записи и обновление реактивного
значения
    fires_data <- data()
    if (nrow(fires_data) > 0) {
      fires_data <- fires_data[-nrow(fires_data), ]
      dbDisconnect(db)
      db <- dbConnect(RSQLite::SQLite(), ":memory:")
      dbWriteTable(db, "fires", fires_data)
      data(fires_data)
      output$formFeedback <- renderUI({
        div(style = "color: green;", "Последний пожар
успешно удален.")
      })
    } else {
      output$formFeedback <- renderUI({
        div(style = "color: red;", "Нет данных
для удаления.")
      })
```

```
    }
  })
  # Реактивный текстовый вывод для отображения времени
последнего обновления
  output$loadingState <- renderText({
    # Автоматическое обновление раз в секунду
    invalidateLater(1000, session)
    # Форматированный текст с текущей датой и временем
    paste("Время последнего обновления:", format(Sys.time(),
"%Y-%m-%d %H:%M:%S"))
  })

  # Рендеринг карты с данными о пожарах и интерактивными
маркерами
  output$fireMap <- renderLeaflet({
    leaflet(data()) %>%
      addTiles() %>%
      addCircleMarkers(~lon, ~lat, radius = ~prediction_zone
* 10,
                       color = ~ifelse(threat_level ==
"Высокая", "red",
                                       ifelse(threat_level
== "Средняя", "orange", "green")),
                       popup = ~paste("Местность:",
location_name, "<br>",
                                      "Тип:", type, "<br>",
                                      "Уровень угрозы:",
threat_level, "<br>",
                                      "Примечания:",
notes))
  })
}

# Запуск приложения
shinyApp(ui, server)
```

Данное приложение Shiny обеспечивает интерфейс для мониторинга пожаров, позволяя пользователям просматривать, добавлять и удалять информацию о пожарах на интерактивной карте. Детально рассмотрим его логику и функции.

1. **Библиотеки и инициализация**

Библиотеки shiny, shinydashboard, leaflet, DBI, и RSQLite предоставляют функциональность для разработки интерактивных веб-приложений, отображения карт и работы с базами данных.

2. **Создание тестовой базы данных**

Функция create_fake_fire_db() инициализирует базу данных SQLite в памяти и наполняет ее тестовыми данными о пожарах. Данные

включают параметры, такие как идентификатор, время, координаты, местоположение, тип, уровень угрозы, прогнозируемая зона и примечания.

3. **Пользовательский интерфейс (UI)**

UI создается с использованием dashboardPage, dashboardHeader, dashboardSidebar и dashboardBody. Это включает выбор уровня угрозы, кнопки для обновления информации на карте, добавления и удаления пожаров, ввода данных о новом пожаре и отображения обратной связи путем formFeedback.

4. **Серверная логика**

Функция server обрабатывает логику приложения и реактивное взаимодействие с пользователем:

– инициализируется база данных с помощью create_fake_fire_db();

– реактивное значение data создается для хранения и обработки данных, считываемых из базы данных;

– событие observe принимает изменения в форме выбора уровня угрозы и фильтрует данные для карты;

– renderLeaflet отображает интерактивную карту с маркерами, которые соответствуют данным пожаров;

– observeEvent для кнопки «Обновить карту» перезагружает данные из базы данных;

– observeEvent для добавления пожара валидирует входные данные и добавляет новую запись в базу данных, после чего обновляет интерфейс пользователя;

– observeEvent для удаления последнего пожара удаляет данные из базы данных и обновляет интерфейс пользователя;

– renderText обеспечивает вывод времени последнего обновления через регулярные интервалы времени.

5. **Запуск приложения:** shinyApp(ui, server) запускает приложение с установленным UI и серверной логикой.

Механики работы различных блоков:

– **интерактивная карта:** обновляется в реальном времени, отражая фильтрацию данных и ввод пользователя;

– **реактивность:** используя реактивные выражения и наблюдателей событий, приложение реагирует на действия пользователя, обновляя интерфейс и данные динамично;

– **валидация данных:** перед добавлением новой информации происходит проверка корректности координат, чтобы предотвратить введение некорректных данных;

– **CRUD операции:** в приложении реализованы операции создания, чтения и удаления данных (CRUD). Пользователи могут добавлять новые записи, просматривать существующие и удалять их.

Для каждого из этих шагов требуется тщательное тестирование и проверка. При добавлении новых функций важно обеспечить, чтобы они интегрировались с существующей логикой и не вызывали ошибок или конфликтов.

Подведение итогов

Чтобы улучшить приложение, можно рассмотреть следующие функции и добавления.

1. Функция геолокации: позволяет пользователям добавить свое текущее местоположение на карту.
2. Выбор даты и времени: использование виджета для выбора даты вместо ввода текста.
3. Расширенный фильтр: добавление возможности фильтрации по дате, типу местности и другим параметрам.
4. Интеграция реальных данных: если есть доступ к API с данными о пожарах, можно интегрировать эти данные в приложение.
5. Адаптивные маркеры: меняют цвет или форму в зависимости от степени угрозы.
6. Панель информации: для показа более подробной информации о выбранном пожаре в отдельной панели.
7. Экспорт данных: кнопка для сохранения отфильтрованных данных в CSV-файл.
8. Функционал редактирования: позволяет редактировать существующие записи о пожарах.
9. Визуализация прогноза распространения пожара: для отображения потенциального распространения пожара на основе прогнозируемой зоны.
10. Фильтрация данных по дате и времени пожара.
11. Интерактивное изменение масштаба круговых маркеров в зависимости от уровня угрозы.
12. Добавление слоев карты для разных типов отображения (например, OpenStreetMap, Satellite).
13. Возможность выбора конкретного пожара для удаления.
14. Использование модальных окон для подтверждения действий пользователя.
15. Визуализация области прогноза распространения пожара.
16. Оптимизация запросов к базе данных.

В исходный код можно добавить представленные функции, чтобы улучшить пользовательский опыт и расширить функциональность приложения.

Интерактивное веб-приложение, представленное в третьей части, является мощным инструментом мониторинга пожаров, который выдвигает визуальное и интерактивное представление текущего состояния пожарной безопасности в выбранной области. Функциональность приложения варьируется от отображения местоположений пожаров на карте до управления данными пожаров через добавление, обновление и удаление записей.

Разработанный интерфейс является чрезвычайно пользовательски дружелюбным, позволяя пользователям легко вводить и манипулировать данными через интуитивно понятные элементы управления. Расширенная реактивность и обратная связь обеспечивают мгновенное обновление информации, делая его отличным средством для оперативного реагирования на изменения и обновления. Использование фреймворка Shinydashboard дает приложению профессиональный внешний вид и ощущение современного веб-ресурса.

Интегрированная база данных SQLite является легкой и эффективной для хранения большого объема данных и манипуляции им без необходимости сложной настройки. Приложение демонстрирует гибкость и масштабируемость, благодаря чему может быть адаптировано и расширено в соответствии с будущими требованиями интеграции с реальными источниками данных.

Таким образом, данный проект является великолепным примером применения стека технологий R в реальных приложениях, предлагая вместе с тем широкие возможности для доработки и улучшения. Отличительной чертой является его открытость для дальнейшей интеграции, например, с системами предсказания пожароопасности и принятия решений, которые могут включать искусственный интеллект и машинное обучение. Этот проект не только демонстрирует силу данных и визуализации в решении критических вопросов, но и открывает путь для дальнейших инноваций в области цифрового управления пожарной безопасностью.

Глава 12
ЗАЩИТА И БЕЗОПАСНОСТЬ ПРИЛОЖЕНИЙ

Защита и безопасность веб-приложений Shiny являются критически важными для сохранения конфиденциальности данных, предотвращения неавторизованного доступа и обеспечения безопасного взаимодействия с пользователем.

12.1. Аутентификация и авторизация

Аутентификация и авторизация являются ключевыми компонентами защиты веб-приложений, включая те, что созданы с использованием Shiny. Они помогают верифицировать личность пользователя (аутентификация) и определяют, к каким ресурсам и операциям у него есть доступ после входа в систему (авторизация).

12.1.1. Аутентификация

Аутентификация обычно представляет собой процесс, в ходе которого пользователь предоставляет уникальные данные, подтверждающие его личность, например, имя пользователя и пароль.

В Shiny есть несколько способов реализации аутентификации.

1. **Shiny Server Pro или RStudio Connect:** эти платформы предоставляют встроенные возможности для аутентификации пользователей.

2. **Shinyproxy:** это прокси-сервер, который поддерживает аутентификацию пользователей и может управлять множеством приложений Shiny.

3. **Custom Authentication:** если требуется больше гибкости, можно создать собственный механизм аутентификации, например, использовать базу данных пользователей, интегрировать с OAuth или другими провайдерами аутентификации.

Пример кода для простейшей формы аутентификации в Shiny:

```
library(shiny)

ui <- fluidPage(
  textInput("username", "Username"),
  passwordInput("password", "Password"),
  actionButton("login", "Log in")
)
```

```
server <- function(input, output, session) {
  observeEvent(input$login, {
    if (input$username == "admin" && input$password ==
"password") {
      # В случае успешной аутентификации
      # показать содержимое приложения или перенаправить
пользователя
    } else {
      # В случае ошибки показать сообщение
    }
  })
}

shinyApp(ui = ui, server = server)
```

Очевидно, что такой метод является базовым и не рекомендуется для производственного использования ввиду низкого уровня безопасности.

12.1.2. Авторизация

Авторизация следует после успешной аутентификации и обеспечивает контроль доступа к определенным ресурсам или операциям. В Shiny есть возможность реализовать авторизацию на уровне серверного кода с проверкой того, имеет ли аутентифицированный пользователь права для выполнения определенных действий или доступа к определенным данным. Например,

```
output$table <- renderTable({
  if (userHasPermission(input$username, "view_table")) {
    # Показать данные
  } else {
    # Показать сообщение об ошибке
  }
}, req(input$username))

# Функция для проверки прав пользователя
userHasPermission <- function(username, permission) {
  # Проверка наличия разрешения в базе данных или любой
системе управления правами
}
```

В обоих случаях аутентификация и авторизация должны быть тщательно продуманы для обеспечения надежности приложения и защиты конфиденциальной информации.

12.2. HTTPS и SSL/TLS

HTTPS и SSL/TLS используются для защиты данных, передаваемых между приложением и пользователем.

HTTPS (Secure HyperText Transfer Protocol) – это защищенная версия протокола передачи данных HTTP. Данные шифруются с использованием SSL/TLS протокола перед отправкой от клиента к серверу или наоборот. Это особенно важно при передаче конфиденциальной информации, такой как пароли, данные кредитных карт или личные сведения, чтобы предотвратить их перехват и использование третьими лицами. Веб-сайты, использующие HTTPS, обычно отображают замочек в адресной строке браузера, говорящий о безопасности соединения.

SSL/TLS (Secure Sockets Layer & Transport Layer Security) представляет собой криптографические протоколы, предназначенные для обеспечения защищенности передачи данных в интернете. SSL – это более старая версия протокола, в то время как TLS – более новая и безопасная версия. Эти технологии шифруют передаваемую информацию таким образом, что ее можно дешифровать только на стороне получателя, что обеспечивает конфиденциальность и подлинность передачи данных.

Почему использование HTTPS важно для Shiny приложений:

– **конфиденциальность:** HTTPS предотвращает возможность прослушивания передаваемых данных несанкционированными лицами;

– **целостность данных:** проверка того, что данные не были изменены в процессе передачи без знания и согласия отправителя и получателя;

– **аутентификация:** подтверждает, что пользователь действительно взаимодействует с задуманным сайтом, а не с поддельным.

Для развертывания Shiny приложения по HTTPS, при использовании собственного сервера потребуется следующее.

1. Получить SSL/TLS сертификат от сертификационного центра или воспользоваться бесплатными сервисами вроде Let's Encrypt.
2. Настроить веб-сервер (например, Apache, Nginx) на прием HTTPS соединений.
3. Конфигурировать сервер Shiny для работы через защищенное соединение. Настройка может включать в себя указание пути к сертификату и приватному ключу.

Для Shiny Server Pro или RStudio Connect настройка HTTPS может отличаться, поэтому важно следовать официальной документации этих продуктов.

Кроме того, если используются облачные платформы или сервисы хостинга, такие как Shinyapps.io или другие платформы-как-услуги (PaaS), они часто автоматически защищают трафик с помощью HTTPS.

Настройка HTTPS на собственном сервере

1. **Получение сертификата:**
   - создайте запрос на подпись сертификата (CSR) и приватный ключ;
   - отправьте CSR в центр сертификации (CA), чтобы получить сертификат SSL/TLS.
2. **Настройка веб-сервера:**
   - укажите в конфигурации веб-сервера путь к вашему SSL/TLS сертификату и приватному ключу;
   - настройте перенаправление с HTTP на HTTPS, чтобы все запросы автоматически переадресовывались на защищенную версию.
3. **Обновление и поддержание сертификатов:** следите за сроком действия вашего сертификата и вовремя продлевайте его.

Использование HTTPS является стандартом для современных веб-приложений, и, несмотря на усилия, связанные с настройкой и поддержкой, оно является необходимым условием для обеспечения безопасности и доверия пользователей.

12.3. Валидация и санитизация ввода

Валидация и санитизация ввода являются основными мерами безопасности для предотвращения атак на веб-приложения, включая приложения Shiny. Валидация проверяет, соответствуют ли вводимые данные определенным требованиям или форматам, в то время как санитизация очищает данные, чтобы предотвратить внедрение вредоносного кода.

12.3.1. Валидация данных

Валидация убедится, что пользовательский ввод соответствует ожидаемому формату и является допустимым, прежде чем обрабатывать его дальше. Примеры валидации включают проверку строк на соответствие формату электронной почты, проверку чисел на нахождение в определенном диапазоне и т. д.

В контексте Shiny, серверная функция может использоваться для проверки входных данных, и, если ввод недействителен, можно возвращать сообщение об ошибке пользователю.

```
server <- function(input, output, session) {
  observeEvent(input$submit, {
    if (!isValidEmail(input$email)) {
      # отображение сообщения об ошибке
      return()
    }
    # продолжение обработки данных
  })
}

# Вспомогательная функция для проверки электронной
почты
isValidEmail <- function(email) {
  grepl("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-
Z]{2,}$", email)
}
```

12.3.2. Санитизация данных

Санитизация данных удаляет или заменяет недопустимые символы в данных, чтобы предотвратить возможные атаки, такие как cross-site scripting (XSS) или SQL-инъекции. В Shiny HTML-рендеринг элементов предоставляется с некоторыми функциями санитизации, так что вредоносные скрипты обычно не выполняются при выводе в текстовые поля или аутпуты.

Однако если вы используете необработанный HTML или скрипты через функции uiOutput или renderUI, вам нужно быть особенно осторожными и санитизировать данные перед их отображением.

Для запрета прямого вывода HTML в Shiny можно использовать htmlEscape:

```
output$mytext <- renderText({
  htmlEscape(input$mytext)
})
```

Рассмотрим лучшие практики:

– **не доверять пользовательским данным:** всегда считать пользовательские данные потенциально опасными и производить их валидацию и санитизацию;

– **использовать существующие библиотеки:** важно применять надежные библиотеки и фреймворки для валидации и санитизации данных;

– **применять оборачивание и экранирование:** для передачи пользовательского ввода в запросы к базам данных использовать механизмы подготовленных запросов и параметризацию, чтобы предотвратить SQL-инъекции;

– **тестировать на уязвимости:** проведение регулярных проверок на наличие уязвимостей для XSS, CSRF и других атак;

– **использовать контекстно-зависимую санитизацию:** метод санитизации данных должен соответствовать месту, где эти данные будут использоваться (например, санитизация URL отличается от санитизации SQL-запросов).

Применение этих методов в приложениях Shiny поможет защитить данные пользователей и обеспечить безопасность работы приложения.

12.4. Изолированная среда разработки

Изолированная среда разработки относится к практике использования отдельных, контролируемых условий для разработки, тестирования и развертывания веб-приложений, включая приложения Shiny. Это ключевая стратегия для обеспечения безопасности, управления конфигурацией и предотвращения непредвиденных сайд-эффектов, которые могут произойти при изменении кода или системы.

В контексте Shiny, развертывание приложения в изолированной среде обеспечивает следующее:

– **безопасность:** ограничивается доступ к системным ресурсам, таким как файлы, другие процессы и сетевые интерфейсы, что помогает предотвратить несанкционированный доступ и потенциальные атаки;

– **стабильность:** изолирование зависимостей и конфигураций для каждого приложения уменьшает конфликты между библиотеками и изменениями в системе;

– **воспроизводимость:** облегчается тестирование и развертывание приложений, поскольку изолированные среды можно настраивать и дублировать с высокой степенью точности;

– **управление ресурсами:** позволяет точно контролировать использование CPU, памяти и других ресурсов, выделяемых для приложения.

Изолированные среды могут быть реализованы с использованием следующих технологий:

– **Docker:** создание изолированной среды с помощью контейнеров, которые инкапсулируют приложение и все его зависимости. Использование Docker является популярным решением при разработке

и развертывании приложений Shiny, особенно в сочетании с такими инструментами, как ShinyProxy;

– **иртуальные машины:** использование полноценно изолированных виртуальных машин с возможностью управления через виртуальные машины, такие как VMware или VirtualBox;

– **Shiny Server (Pro) и RStudio Connect:** эти платформы обеспечивают уровень изоляции для приложений и возможность управления доступом к системным ресурсам.

При развертывании приложения Shiny следует:

– **ограничить права:** запускать приложение от пользователя с минимально необходимыми правами, исключая права суперпользователя;

– **осуществлять отслеживание и логирование:** регистрировать активность приложения для быстрого выявления и решения вопросов, связанных с безопасностью и производительностью;

– **проводить обновление:** регулярно обновлять все компоненты системы, включая R, пакеты Shiny, Docker и операционную систему, чтобы минимизировать уязвимости.

Изолированная среда разработки – это важный элемент стратегии безопасности, особенно для веб-приложений Shiny, работающих с чувствительными данными. Вынесение приложений в изолированную среду не только повышает безопасность, но и облегчает разработку, тестирование и поддержку приложений.

12.5. Журналирование и мониторинг

Журналирование и мониторинг являются неотъемлемыми составляющими поддержки и обеспечения безопасности приложений Shiny. Они позволяют отслеживать состояние приложения в реальном времени, реагировать на ошибки и потенциальные угрозы безопасности, а также анализировать тенденции использования приложения.

12.5.1. Журналирование

Журналирование в приложениях Shiny охватывает запись событий приложения, пользовательских действий и системных сообщений. В контексте Shiny Server и RStudio Connect логи обычно включают в себя информацию о пользователях, включая IP-адреса, время доступа и использованные браузеры; детали о входящих и исходящих запросах HTTP; ошибки приложения и сообщения из stderr и stdout; сообщения

об активации R выражений и реактивных событий; время выполнения операций и ресурсы, потребляемые приложением.

Есть возможность логировать собственные события приложения Shiny, используя функции shiny::log*, например, shiny::logInfo(), shiny::logWarn() или в R cat() или print() для записи сообщений в консоль, и эти сообщения будут захватываться Shiny Server.

12.5.2. Мониторинг

Под *мониторингом* обычно понимают активное наблюдение за состоянием и работой приложения. Для Shiny серверного приложения это включает в себя анализ использования приложения для оптимизации производительности и обнаружения узких мест, отслеживание времени ответа сервера и времени загрузки страниц, мониторинг состояния веб-сервера и инфраструктуры, отслеживание ошибок в приложении и количества активных пользователей.

Мониторинг может осуществляться с помощью специализированных инструментов и сервисов. Например, Prometheus и Grafana для сбора метрик и визуализации данных о работе приложений, Shiny Server Pro и RStudio Connect предоставляют встроенные средства для мониторинга активности приложений, ELK Stack (Elasticsearch, Logstash, Kibana) для анализа логов.

Рассмотрим лучшие практики:

– **сегментация логов:** отделять ошибки приложения от доступа и отладочных сообщений, чтобы упростить их анализ;

– **регулярные проверки логов:** настроить регулярный обзор лог-файлов, чтобы выявлять потенциальные проблемы;

– **оповещение:** настроить систему оповещений о критических ошибках или необычной активности;

– **аудит:** регулярно проводить аудиты безопасности на основе данных логов;

– **ротация и резервное копирование:** ротировать логи и сохранять их копии для обеспечения продолжительности данных и соответствия законодательным требованиям;

– **объем логирования:** настроить уровни логирования, чтобы избегать избыточности данных и упростить отладку.

Журналирование и мониторинг в приложениях Shiny повышают их надежность, доступность и обеспечивают ценную обратную связь для постоянного улучшения приложения.

12.6. Ограничение ресурсов

Ограничение ресурсов в контексте веб-приложений Shiny заключается в управлении доступными системными ресурсами, такими как процессорное время, оперативная память, количество одновременно запущенных приложений и время сессий пользователей, чтобы обеспечить надежную и эффективную работу служб.

Изучим, для чего необходимо ограничивать ресурсы.

1. **Предотвращение перегрузки сервера:** ограничения ресурсов защищают систему от чрезмерного использования одним приложением или пользователем, что может привести к снижению производительности или даже к аварийным ситуациям.

2. **Равномерное распределение ресурсов:** установление пределов помогает обеспечивать равный доступ к ресурсам для всех приложений и пользователей, что важно в многопользовательской среде.

3. **Управление производительностью:** ограничения позволяют контролировать ожидаемые показатели производительности и обеспечивать их соответствие требованиям SLA.

4. **Безопасность:** ограничения помогают предотвратить некоторые виды DOS-атак, когда злоумышленники пытаются исчерпать системные ресурсы, делая сервисы недоступными для других пользователей.

Как применять ограничения ресурсов в Shiny, предоставлено ниже.

1. **Настройки Shiny Server:** если используете Shiny Server (включая Pro версию), можно настроить ограничения в конфигурационном файле shiny-server.conf. Например, можно ограничить количество процессов (приложений) и пользователей, время бездействия сессии и время жизни процесса.

Bash:

```
    server {
      listen 3838;

      location / {
        app_idle_timeout 300; # 5 minutes
        sanitize_errors off;
        app_dir /srv/shiny-server;
      }
    }
```

2. **RStudio Connect:** предоставляет более продвинутые настройки управления и мониторинга приложений, включая ограничение времени работы и использования памяти.

3. **Docker и другие контейнеры:** при использовании Docker для развертывания приложений Shiny можно применять флаги --memory, --cpus при запуске контейнера для ограничения ресурсов.

Bash:

```
   docker run -d --name shiny_app --memory="1g" --
cpus="0.5" my_shiny_app
```

4. **Kubernetes:** в системе оркестрации Kubernetes можно использовать ограничения ресурсов на уровне конфигурации подов (pods).

YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: shiny-app
spec:
  containers:
  - name: shiny-app
    image: my_shiny_app
    resources:
      limits:
        memory: "1024Mi"
        cpu: "500m"
```

5. **Ограничения на уровне операционной системы:** на этом уровне можно применять инструменты, такие как ulimit, cgroups или Windows Job Objects, для управления ресурсами.

Изучим лучшие практики:

– **мониторинг ресурсов:** регулярно контролировать использование ресурсов с помощью инструментов мониторинга, чтобы понимать текущую нагрузку и соответственно корректировать ограничения;

– **тестирование нагрузки:** проведение тестов под нагрузкой, чтобы увидеть, как приложение ведет себя в условиях, приближенных к производству;

– **динамическое масштабирование:** применение систем, которые могут автоматически масштабироваться в зависимости от нагрузки, таких как Kubernetes, может помочь оптимизировать использование ресурсов.

Реализация данных подходов и инструментов позволит эффективно управлять ресурсами, используемыми приложениями Shiny, обеспечить стабильную работу и высокую доступность сервисов, а также повысить уровень безопасности ваших приложений.

12.7. Резервное копирование и восстановление

Резервное копирование и *восстановление* – это критически важные аспекты любой стратегии обеспечения безопасности и непрерывности бизнес-процессов для IT-систем, в том числе для веб-приложений Shiny. Они помогают минимизировать потерю данных и ускорить восстановление работоспособности системы после сбоев, аварий или атак.

Резервное копирование для Shiny приложений

1. **Код приложения:**

– регулярно сохраняйте копии кода и всех связанных с ним файлов (например, файлов конфигурации, описания зависимостей);

– используйте системы управления версиями, такие как Git, предоставляющие дополнительное преимущество отслеживания изменений и возврата к предыдущим версиям.

2. **Данные:**

– для приложений Shiny, хранящих данные пользователя или результаты вычислений, важно регулярно создавать резервные копии этих данных;

– в зависимости от инфраструктуры может потребоваться резервное копирование баз данных, файлов данных на сервере или внешних хранилищ данных.

3. **Конфигурация сервера:** сохраняйте конфигурации веб-серверов и серверов Shiny так, чтобы можно было быстро восстановить приложение в случае сбоя.

Восстановление для Shiny приложений

1. **План восстановления:** разработайте четкий план действий для восстановления из резервной копии, который будет включать в себя шаги восстановления данных и кода приложения.

2. **Автоматизация:** автоматизируйте процесс резервного копирования и восстановления, если возможно. Это может включать использование скриптов и задач планировщика.

3. **Тестирование:** периодически тестируйте процесс восстановления, чтобы убедиться, что резервные копии работают и их можно восстановить в случае необходимости.

Рекомендации по резервному копированию и восстановлению

1. **Частота и географическое распределение:**
   - определите частоту резервного копирования на основе важности данных и приемлемой потери данных в случае сбоя (временной точки восстановления);
   - храните резервные копии географически разнесенно, чтобы предотвратить потерю данных из-за местных сбоев (например, наводнений, пожаров).

2. **Безопасность резервных копий:** защищайте резервные копии, используя шифрование и безопасные методы доступа, чтобы они сами по себе не стали целью для атак.

3. **Облачные сервисы:** рассмотрите использование облачных решений для резервного копирования, таких как AWS S3, Google Cloud Storage или Azure Storage, которые предлагают надежные и масштабируемые решения.

Важно помнить, что наличие актуальных резервных копий может сэкономить много времени и ресурсов при внезапных инцидентах и является обязательной практикой для любого серьезного веб-приложения, включая приложения Shiny.

12.8. Разделение окружений

Разделение окружений в разработке программного обеспечения – это практика, при которой для различных этапов жизненного цикла проекта (разработка, тестирование, промежуточное развертывание (staging) и производство) используются разные среды. Каждая среда предназначена для определенных целей и может иметь свои уникальные настройки.

Разделение окружений для Shiny приложений

1. **Разработка (Development):**

– это локальные среды, где разработчики пишут и проверяют код;

– ошибки и эксперименты допустимы и ожидаемы;

– может быть настроена для подключения к базам данных разработки или моковым сервисам.

2. **Тестирование (Testing) или QA (Quality Assurance):**

– служит для всесторонней проверки новых изменений перед их выпуском в стейджинг или продакшн;

– подключается к базам данных тестирования, которые содержат данные, симулирующие реальные операционные данные, но без риска их повреждения.

3. **Стейджинг (Staging):**

– это точное воспроизведение продакшн-среды, что помогает протестировать поведение приложения в условиях, максимально приближенных к реальным;

– служит для последних проверок перед фактическим развертыванием и может быть использована для демонстрации функционала бизнес-заказчикам.

4. **Производство (Production):**

– это среда, где приложения доступны конечным пользователям и где производительность, доступность и безопасность критически важны;

– подключается к реальным операционным базам данных и ресурсам.

Почему разделение окружений важно:

– **безопасность:** уменьшается риск случайной порчи или раскрытия реальных данных;

– **стабильность:** изменения вносятся порционно и контролируемо, что способствует стабильности IT-систем;

– **надежность:** легче выявлять и исправлять ошибки на ранних этапах, не допуская их в продакшн;

– **управляемость:** Каждая среда может быть оптимизирована под свои уникальные цели и нагрузки.

Далее перечислены лучшие практики для разделения окружений:

– **автоматизация:** используйте инструменты для автоматизированного развертывания, которые могут гарантировать, что все среды настроены согласованно;

– **конфигурация как код:** управляйте конфигурациями через код (например, с помощью Ansible, Chef, Puppet или Docker), чтобы избежать «снежинок» – уникально настроенных ручным путем машин;

– **изоляция ресурсов:** поддерживайте строгую изоляцию ресурсов между средами для предотвращения влияния между ними;

– данные: используйте тестовые, анонимизированные или маскированные данные в не-продакшн средах;

– **мониторинг и логирование:** обеспечьте адекватный мониторинг и журналирование во всех средах для отслеживания ошибок и управления производительностью;

– **документирование:** ведите четкую документацию конфигураций и различий между средами.

Разделение окружений помогает командам разработчиков быть более производительными и минимизировать риски при разработке, тестировании и развертывании приложений Shiny, а также обеспечить более высокое качество конечного продукта для пользователей.

Даже если приложение Shiny предназначено только для внутреннего использования в компании, все равно имеет смысл следовать этим рекомендациям. Безопасность никогда не должна компрометироваться в угоду удобству или скорости разработки. Многие из вышеописанных мер могут быть реализованы через инфраструктуру, на которой размещается приложение, обеспечивая надежную защиту ваших данных и пользователей.

Глава 13
ЗАВЕРШАЮЩИЙ ПРОЕКТ

Перейдем к финальной стадии курса по программированию на R Shiny. За время обучения произошло знакомство с основами языка R, изучение ключевых элементов Shiny. Учебное пособие помогло научить создавать интерактивные пользовательские интерфейсы, визуализировать данные и интегрировать продвинутые функциональности для разработки динамичных приложений. Объединим эти знания и навыки в одном завершающем проекте.

Цель проекта

Завершающий проект курса предназначен для демонстрации полного понимания и умения практически применять все изученные концепции. Проект будет служить кульминацией обучения и возможностью получить ценный опыт, который можно применить в реальных ситуациях.

Задание

1. Выберите тему проекта, которая отражает реальную проблематику, и разработайте интерактивное веб-приложение Shiny, решающее стоящие перед ней задачи.

2. Приложение должно включать следующие элементы:

- аутентификация и авторизация пользователей для доступа к функциям приложения;
- адаптивный пользовательский интерфейс, соответствующий различным типам устройств;
- интеграция с внешними данными и API;
- реактивные обновления, интерактивные графики и карты;
- применение различных виджетов для интерактивного ввода и вывода данных.

3. Обеспечьте производительность и безопасность приложения, используя изученные методы кэширования данных, асинхронных операций, HTTPS и стратегии безопасности.

4. В документации проекта опишите использованные технологические решения, архитектуру приложения и предложите инструкции по развертыванию и тестированию приложения.

Проект будет оцениваться по следующим критериям:

- соответствие требованиям задания и комплексность функциональности;
- качество кода, включая читаемость, комментарии и соблюдение стандартов написания кода на R;

- эстетика и пользовательские качества интерфейса, включая адаптивный дизайн и UX;
- безопасность приложения, в том числе реализация мероприятий защиты данных и пользователей;
- практическая применимость и востребованность решаемых приложением задач.

По мере работы над проектом необходимо обращаться к материалам курса, внешним ресурсам, сообществу разработчиков и в соответствующую документацию для получения поддержки и разрешения возникающих вопросов.

Завершающий проект – это шанс проявить себя и задокументировать путешествие по миру разработки интерактивных веб-приложений. Нужно подходить к проекту как к возможности проявить креативность и инновации, дополнить свое портфолио и подготовиться к следующему этапу карьеры или учебного пути.

ЗАКЛЮЧЕНИЕ

В рамках данного учебного пособия были успешно рассмотрены ключевые аспекты работы с языком R и фреймворком Shiny, ориентированные на создание интерактивных веб-приложений. Освоение представленного материала позволило читателям приобрести базовые навыки, необходимые для дальнейшего развития в области аналитики данных и веб-разработки. Приобретенные компетенции могут быть применены для решения широкого спектра задач, связанных с визуализацией данных и созданием аналитических инструментов.

Полученные знания формируют прочный фундамент для дальнейшего углубленного изучения специализированных областей, таких как оптимизация кода на языке R, применение алгоритмов машинного обучения, работа с большими данными и повышение производительности приложений Shiny. Для эффективного развития в этих направлениях рекомендуется использовать разнообразные образовательные ресурсы, включая официальную документацию, материалы сообщества RStudio, профильные семинары и конференции, онлайн-курсы, специализированные блоги и научную литературу.

Особое внимание следует уделить практическому применению полученных навыков. Самостоятельная разработка проектов и эксперименты с различными подходами являются ключевыми факторами для закрепления теоретических знаний и развития практических компетенций. Важно не бояться ошибок, поскольку они являются неотъемлемой частью процесса обучения и способствуют более глубокому пониманию материала.

В заключение данный учебный материал предоставляет тем, кто его освоил, возможность овладеть базовыми принципами работы с R и Shiny, создавая предпосылки для дальнейшего профессионального роста в области аналитики данных и веб-разработки. Рекомендуется непрерывное самосовершенствование и активное применение полученных знаний для достижения высоких результатов в этой динамично развивающейся сфере.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Уикем, Х. Изучаем Shiny / Х. Уикем; перевод с английского А. Ю. Гинько. – Москва : ДМК Пресс, 2022. – 374 с. – ISBN 978-5-97060-964-4.

2. Лукьянов, П. Б. Моделирование и анализ в среде R с использованием пакета Shiny : учебно-методическое пособие для бакалавриата / П. Б. Лукьянов. – Москва : Прометей, 2023. – 102 с. – ISBN 978-5-00172-465-0.

3. Задорожный, С. С. Статистическая обработка данных на языке R / С. С. Задорожный. – Москва : Физический факультет МГУ им. М. В. Ломоносова, 2023. – 104 с.

4. Венэбльз У. Н. Рабочая группа разработки R. Введение в R. Заметки по R: среда программирования для анализа данных и графики. Версия 1.15.0 (2012-03-30) / У. Н. Венэбльз, Д. М. Смит, Рабочая группа разработки R ; перевод с английского А. А. Фоменко. – Москва, 2013. – 109 с.

5. Акберова, Н. И. Основы анализа данных и программирования в R : учебно-методическое пособие / Н. И. Акберова, О. С. Козлова. – Казань : Альянс, 2017. – 33 с.

6. Лонг, Дж. Д. R. Книга рецептов. Проверенные рецепты для статистики, анализа и визуализации данных / Дж. Д. Лонг, П. Титор; перевод с английского Д. А. Беликова. – Москва : ДМК Пресс, 2020. – 510 с. – ISBN 978-5-97060-835-7.

7. Филиппов, Ф. В. Обработка информации в среде RStudio : учебное пособие / Ф. В. Филиппов, А. Н. Губин. – Санкт-Петербург : СПбГУТ им. М. А. Бонч-Бруевича, 2016. – 86 с.

8. Wickham, H. R for Data Science / H. Wickham, G. Grolemund. – Sebastopol : O'Reilly Media, 2017. – 520 p. – ISBN 978-1-491-91039-9.

9. Shiny User's Guide : [website]. – URL: https://shiny.rstudio.com/articles/#shiny-user's-guide (дата обращения: 15.02.2024).

10. Lovelace, R. Geocomputation with R / R. Lovelace, J. Nowosad, J. Muenchow. – Boca Raton : CRC Press, 2019. – 416 p.

11. Leaflet // Leaflet: an open-source JavaScript library for mobile-friendly interactive maps : [website]. – URL: https://mourner.github.io/Leaflet/index.html (дата обращения: 10.10.2023).

12. Wickham, H. ggplot2: Elegant Graphics for Data Analysis / H. Wickham. – New York : Springer-Verlag, 2016. – 260 p.

Учебное издание

Стратонов *Дмитрий Дмитриевич*

Shiny. Методы и инструменты для разработки интерактивных веб-приложений на языке программирования R

ISBN 978-5-94984-942-2

9 785949 849422

Редактор П. С. Фенина
Оператор компьютерной верстки Т. В. Упорова

Подписано в печать 04.04.2025. Формат 60×84/16.
Бумага офсетная. Цифровая печать.
Уч.-изд. л. 7,99. Усл. печ. л. 9,07.
Тираж 300 экз. (1-й завод 26 экз.).
Заказ № 8101

ФГБОУ ВО «Уральский государственный лесотехнический университет».
620100, Екатеринбург, Сибирский тракт, 37.
Редакционно-издательский отдел.
Тел. 8 (343) 221-21-44.

Типография ООО «ИЗДАТЕЛЬСТВО УЧЕБНО-МЕТОДИЧЕСКИЙ ЦЕНТР УПИ».
620062, РФ, Свердловская область, Екатеринбург, ул. Гагарина, 35а, оф. 2.
Тел. 8 (343) 362-91-16.