Developing Shiny application

Best Practice Guide

October 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Contents

1 Introduction

To understand the terms of data analysis, or to learn a specific programming language (R in this case), can be difficult for many people. Providing a single user-friendly application containing visualisations of the results might be very useful. To do so, it is important to put together requirements for such application. You have to ask yourself (or your customer), which unit it will be running on (cloud server, user laptop, mobile, ...), how big data sets you will be processing, or what features or functions you will need, and so on. Keep in mind, that requirements can change during the time based on the user feedback. Moreover, it is better to start with a simple application and gradually add new functionalities rather than plan and implement whole application at once. Therefore, the recommendation is to implement individual functions as standalone modules. The modules can then be simply modified, added, or removed from the application. Further, to obtain some user feedback on the application, a mock-up design has to be created. The mock-up design helps you to ensure that the application satisfies all the specified requirements during the process. It also helps to check, if there is something missing in the app, if the app is understandable, or it is easy to work with. You can also realise that application contains features which are not necessarry for the whole software, and can be removed.

2 Developing process

The developing process can be summarised as follows:

- define requirements for the application,
- create mock-up design,
- select suitable technology (packages, tools),
- create project enviroment (implement R base codes, move the codes to the application, use version control, create tests for the application, user testing, etc.)
- if everything is all right, deploy the app.

With this process at hand, you will be able to deploy the app without any further problems. If there are some bugs, or the requirements are updated, just update the design, the codes, and build app again. In the text below, example of developing a simple "Shiny app" is provided. Also, brief summaries of R packages suitable for building the Shiny app are contained in the text.

2.1 Requirements

The requirements are integral part of all the softwares and applications, which are being developed around the world. It is the very first step to make clear what are your needs. You have to specify the requirements on the software, or you have to ask your customer about them, at the very beginning of the development process. For example, consider the situation, in which you have to create following application:

- the application has to run either on Windows OS, Linux OS, or macOS, i.e. on notebook or desktop PC (R language is cross-platform!),
- the user will work with own datasets (upload dataset, tidy dataset, show dataset, etc.),
- the user will want to plot some dataset summaries,
- the user will need some clustering methods,
- also an authentication to the app will be needed (for restricted data).

If the list of requirements exists, you can start developing the application. Since the Shiny serves as extension for R to create application using R code, it is a good habit to implement funcionalities at R itself and test them, at first. Then, move these functions, or modules, into the Shiny app. While moving into the Shiny app, you have to think about a mock-up design of the application, i.e. how to put all the features together and connect them to a user interface.

2.2 Mock-up design

The mock-up design is being created to provide a better idea of the graphical user iterface. The draft version helps you realise:

- what inputs are needed, and in which format (numeric, character, dataframe, ...),
- what outputs will be produced, and in which format (numeric, character, dataframe, ...),
- the need of (interactive) plots/images, if any,
- if the application will be single or multi page,
- etc.

You can create the mock-up desing by hand on the paper, use some of the graphical softwares (designers), or implement basic UI straight into Shiny (if you are familiar with Shiny). If you have no experiences with creating mock-up design (the application at all), we recommend you to use some of the existing designers. You can find plenty of them on the internet. Just search, e.g., for "applications mock-up designer". They help you imagine how the inputs look like, help you create (responsive) layouts, and so on. Drawing design by hand can be the fastest way, especially if you are experienced in mock-up creation, but for a customer, output of the mock-up designer is probably more suitable.

2.3 Package tool selection based on mock-ups and requirements

With the mock-up design and the requirements at hand, you can decide which package tool, you will use. You can work with package providing simple predefined template, or you can use packages providing feancy styling of the app components, etc.

Focusing on the R programming language, which is commonly used for the data analysis, there are packages for the graphical interface such as Shiny, BS4Dash, or ShinyMobile in the R universe. These frameworks provide a set of functions which allow you to create standalone web applications or embed them into R markdown documents. You can create application with graphical interface similar to the existing webpages (webpage applications), or if you are familiar with CSS styling and JS, there could be no difference! Below, a brief introduction of each is given.

2.3.1 Shiny

Shiny is an R package that makes it easy to build interactive web apps straight from R. You can host standalone apps on a webpage or embed them in R markdown documents or build dashboards. You can also extend your Shiny apps with CSS themes, htmlwidgets, and JavaScript actions. You do not need any web development experience at all. Shiny includes built-in input widgets, which can be easily used with minimal syntax knowledge.

For more information visit https://shiny.rstudio.com/ .

2.3.2 BS4Dash

BS4Dash package provides an extension for a Shiny package. It relies on the Bootstrap 4, which is not natively supported by Shiny. BS4Dash allows you to develop a Shiny app with more professional look and feel. Some of the main features are fullscreen toggle, popover, tooltips, right sidebar, dashboard user dropdown, beautiful preloaders, etc.

For more information visit https://rinterface.github.io/bs4Dash/ .

2.3.3 ShinyMobile

ShinyMobile is built on the top of the latest Framework 7. It supplies functions to develop an application especially for iOS and Android. In other words, it includes Shiny widgets adapted for all mobile platforms. ShinyMobile is PWA capable, meaning that it can be displayed full screen on many mobile devices.

For more information visit https://rinterface.github.io/shinyMobile/ .

2.3.4 Shiny extensions

There are lots of frameworks for the R distribution, which can be used to create the application. There is also a gitlab repository containing structured list of many (not all) Shiny extensions at https://github.com/nanxstats/awesome-shiny-extensions. This is a good starting point to understand what is done in terms of Shiny development. Of course, you can look for others on the internet yourself.

2.4 Creation of the project environment

The requirements help you to get an idea how complex your application will be. If the application will be simple, you can already start to implement the functions, the modules, without any advanced tools. However, some applications can be really complex. More complex applications have code readability issues. This can lead to difficult debugging due to many interconnections between modules, and so on. Also be aware of interface complexity, i.e. there are many inputs, or outputs defined in user interface to make application clearly understandable. To avoid problems like these, it is a good habit to find a balance between those two complexities. Think twice about the application before you start to implement it, but do not worry if it looks scary at first sight. In the R universe, there exist frameworks, which help you to manage the whole project from the beginning, starting with the implementation step all the way to the product deployment, so there will be less troubles in your development process. One of these frameworks, the Golem package, is described below.

2.4.1 Golem package

> Golem is a toolkit for simplifying the creation, development, and deployment of a shiny application.

Everything about the golem package can be found at https://engineering-shiny.org/. The book also presents the idea of engineering process to develop and successfully deploy the app. Here is a short introduction of the golem package as mentioned in the provided book.

If you create a new golem project, in this case it is called golex (golem example), a structured folder architecture will be created.

golex

├── DESCRIPTION

├── NAMESPACE

├── R

│ ├── app_config.R

│ ├── app_server.R

│ ├── app_ui.R

│ └── run_app.R


```
├── dev
│   ├── 01_start.R
│   ├── 02_dev.R
│   ├── 03_deploy.R
│   └── run_dev.R
├── inst
│   ├── app
│   │   └── www
│   │   └── favicon.ico
│   └── golem-config.yml
└── man

└── run_app.Rd
```

At the **DESCRIPTION** file, you will add series of metadata about the application. The name of the author(s), version of the app, what is its purpose, etc. It will be filled automatically by running the scripts from **\dev** folder. The **NAMESPACE** defines how to interact with the rest of the package (application) and it will never be touched by you. This file will be also created in an automated way when running the documenting process of your R package. All R functions (all app functions) will be held in **/R** folder. Each R file used during development should be stored in **/dev** folder. In the **/inst/app/ww,** you will keep every CSS or JS files which define the final look of your application. The golem configuration can be set in **golem-config.yml** or you can use **golem_opts** in the **run_app()** function. The purpose of the last **/man** folder is to held package documentation.

As you can see, the golem package really helps you to implement and successfully deploy the app. It helps you to maintain the package structure and even build the app. For more ideas to create production-grade Shiny app, and other information about the golem package, please visit the link above. You can use another source of information, of course. There are plenty of them in the internet.

2.4.2 GitHub/GitLab

Successful development of larger applications cannot be possible without any "smart" code archives. If you are creating a complex package, you can easily make a mistake and you will disrupt the rest of the code. In that case, you will want to use last functional version of the application, so you have to archive it from time to time. The better way is to use a version system like Github, Gitlab, Bitbucket, and so on. You can upload whole new project, or just changes, fetch new data, etc. by simple **git** commands from terminal (command line). It will check every change from the last files to a new one, so if you make a mistake, you just use previous version of a file. Moreover, if you do not like terminal's (command line's) commands, you can use some of the existing graphical git softwares to manage your git repository or you can use built-in git system in the RStudio. If the git repository is initialised within the project folder, you can do all git commands from RStudio itself. The last option is to use web application of the used version systems to manage stored files. Most version systems provide this functionality. It is only up to you, which way is the most suitable for you.

2.5 Tests

Before the package is ready to deploy, the tests have to be done. It automatically checks if the application can be built without any problems, functions behave as expected, and so on. In general, the tests can be divided in three sections:

- Unit tests - functions behave as expected,
- Server function tests - testing of reactive components and outputs,
- Snapshot-based tests - simulate user actions, like clicking on the buttons, take snapshots of the application state, and compare the app state with the saved snapshots.

You can for example use **runTests()** function in Shiny 1.5.0 and more to test your app. Simple example of application with defined tests can be created by **shinyAppTemplate("app_name")** with used option **1: All**. It will generate folder structure with **/tests**. For more information, see https://shiny.rstudio.com/articles/testing-overview.html. Code tests can be also done with a testthat package, see https://testthat.r-lib.org/.

2.5.1 Continuous integration

In short, Continuous Integration (CI) means that multiple developers push small changes frequently into shared repository. Moreover, after each push it is important to run automated tests to verify the integration of a new code into the package (application). Some of the version systems are CI-friendly. For example Github has a feature Github Actions to realise CI.

2.6 Release

If you try to find **what is a software release** on the internet, you can for example find a webpage https://www.techtarget.com/searchsoftwarequality/definition/release, in which the release is defined as

> A release is the distribution of the final version or the newest version of a software application. A software release may be public or private and generally signifies the unveiling of a new or upgraded version of the application.

In other words, you can release the application after all implementation is done and all tests are valid, i.e. you can deploy the app to the product.

3 Building blocks

Some of the possible functionalities of the Shiny will be discussed in this section. Here you will learn how to create a login screen, which plot libraries exist, how to use interactive data tables, how to transfer data, connect your application to a database, and learn about the difference between R6 and S3 model.

3.1 Login screen

You do not need to worry about whether the Shiny is able to provide a login screen, because it can! To learn how to create a login page, you can visit https://towardsdatascience.com/r-shiny-authentication-incl-demo-app-a599b86c54f7.

The author of the given hands-on tutorial considers 3 approaches to build the login page:

1. the basics of the authentication is covered,
2. the login form and the corresponding server logic is packed into seperate modul (as we mention above, using modules within app is powerful),
3. the use of shinyauthr package is discussed (it also includes password hashing).

You can find other tutorials on the internet, e.g. you can inspire yourself at https://stackoverflow.com. The important thing is that you are able to create the login screen in the Shiny without any obstacles. It is up to you which method you will use.

3.2 Visualisations - plot libraries

The first step to understand the data is to visualise them. The Shiny provides several methods to do so. You can use static plots, interactive plots, or you can even display the data on interactive maps using Leaflet package. Brief summaries for each of them are stated below.

3.2.0.1 Static Usage of the static plots in the Shiny is quite the same as in R itself. Data can be visualised using for example ggplot2 library. Hovewer, there is one difference in the Shiny. You have to use `plotOutput()` command on the UI side, and `renderPlot()` on the server side. Displaying the images is also possible in the Shiny. To render an image, you will use `imageOutput()` and `renderImage()`. Look into the help to get familiar with these commands.

3.2.0.2 Interactive Interactive plots within the Shiny app are powerful feature. `plotOutput()` defines four different mouse events:

- **click** for a single mouse click,
- **dblclick** for a double click,
- **hover** to do something when mouse stays in the same place within the plot for a while,
- **brush** to use a rectangular selection tool.

The examples of the given events can be found https://mastering-shiny.org/action-graphics.html.

For some advanced interactive plots, visit https://shiny.rstudio.com/articles/plot-interaction-advanced.html.

While creating an interactive plots, you have to keep in mind that it takes some time to respond to the event. In other words, when you click into a plot, the application has to register the event, evaluate the event (do defined operations), and render updated plot. You also have to take into account a response of the server (transfer speed) if you deployed the app at the website.

3.2.0.3 Leaflet As it is stated at the website https://rstudio.github.io/leaflet/.

> Leaflet is one of the most popular open-source JavaScript libraries for interactive maps. It's used by websites ranging from The New York Times and The Washington Post to GitHub and Flickr, as well as GIS specialists like OpenStreetMap, Mapbox, and CartoDB.
>
> This R package makes it easy to integrate and control Leaflet maps in R.

The Leaflet package provides functions to easily integrate the maps into the Shiny app. How to use the Leaflet package is described at https://rstudio.github.io/leaflet/shiny.html.

3.3 Tables

There are two ways to display data in tables. A static one and a dynamic one. It depends whether you need to just display the data or you need to filter and sort visualised data.

3.3.0.1 Static Rendering data in the static table is done using `tableOutput()` and `renderTable()`. It will display the data in the given place of the app. Static table is suitable for displaying small matrices and data frames.

3.3.0.2 Interactive Interactive tables are the best way to represent data frames in the Shiny app. You can use built-in commands such as `dataTableOutput()` and `renderDataTable()` or use another package such as **reactable**, see https://glin.github.io/reactable/index.html.

Interactive tables have wide range of utilities like search function, sort columns, define number of visible rows per page, navigate through the table's pages, and so on.

3.4 Data transfer logic

To work with a data within the Shiny app, you have to (in general) **upload** the data into it. In the UI, use `fileInput()` with defined id and label. On the server side, things are a bit more complicated. Most inputs in the Shiny return simple vectors, but `fileInput()` returns a data frame with four columns:

- **name** as original file name on the user's computer,
- **size** as the size of the file,
- **type** in which the MIME type of the file is specified,
- **datapath** as a path to where the data has been uploaded on the server.

The simplest way to eliminate undesirable file formats is to define the allowed (acceptable) formats in `fileInput()`. Allowed formats are defined by the parametr `accept`, e. g. `fileInput("upload", "Upload a file", accept = c(".csv", ".tsv"))`. To avoid any further errors, it is necessary to use `req(input$upload)` on the server side, because after the page is loaded, the `input$upload` is `NULL`.

To create a download action on the UI side, use `downloadButton()` or `downloadLink()`. A command downloadHandler(filename, content) is used on the server side. It has two parameters

- **filename** - a function, which returns a file name,
- **content** - a function with one argument (`file`), which is the path to save the file.

For more information about uploading and downloading data, visit https://mastering-shiny.org/action-transfer.html.

3.5 Database connection

As the Shiny was becoming more popular, there was a growing need to connect the app to the external database in the Shiny verse. Keep in mind that using the external database is not the best way in general. The better way is to use local in-memory data, but if you already have the data in the external database or the data does not fit in your memory, the connection to the external database is right for you.

You can use **dplyr** package to manage remote data as the local in-memory data. If you need to do more advanced operations, the **DBI** package will be more suitable for you. How to use **dplyr** and **DBI** package with a remote database is described at https://shiny.rstudio.com/articles/overview.html.

Beware of secure vulnerability when attempting the external database. To learn something about the SQL injection prevention, visit https://shiny.rstudio.com/articles/sql-injections.html.

4 Alpha version

Imagine a situation where you have to create an application in which you will load data of a network graph from the CSV file and run a simulation of the throughput of the graph. The vertices of the graph will represent individual stations in the process and the weights will be a time needed to move from one station to another. The input of the simulator will also be a number of people that will go through the graph.

The example will start with preparing the new project in RStudio and will continue through the creation of necessary functions to the successful run of the whole application. Only the basic Shiny functions will be used.

First, run the RStudio and install the Golem package as well as the Shiny package (if you have not already done it), and load them:

```
install.packages("golem, shiny, shinyjs")
library("golem, shiny, shinyjs")
```

Make sure that you have also installed **dplyr**, **plotly**, and **utils** packages.

Then, make a new empty project **File > New Project... > New Directory > Create Package for Shiny App using golem**. You can name the directory, e.g., as **graphapp**. It will create the directory structure of the package as it has been discussed above. It will also open a **dev/01_start.R** file in which you can define a description of the package in lines **21-29**, e.g.,

```
pkg_name = "graphapp,
pkg_title = "Graph simulator",
pkg_description = "Graph simulator loads a network graph from a CSV file and runs the simulation of its
author_first_name = "John",
author_last_name = "Doe",
author_email = "john.doe@email.com",
repo_url = NULL
```

It also contains other usefull functions like function to prepare git repository, prepare testing infrastructure, and so on. After you have defined the description as above you can simply run **01_start.R** and everything will be set automatically. You can check the **DESCRIPTION** file if you want to be sure. As all is set up, run `golem::run_dev()`. The very first app window will appear (Fig.1) and you can move into **R** folder where all needed functions for the app will be defined.

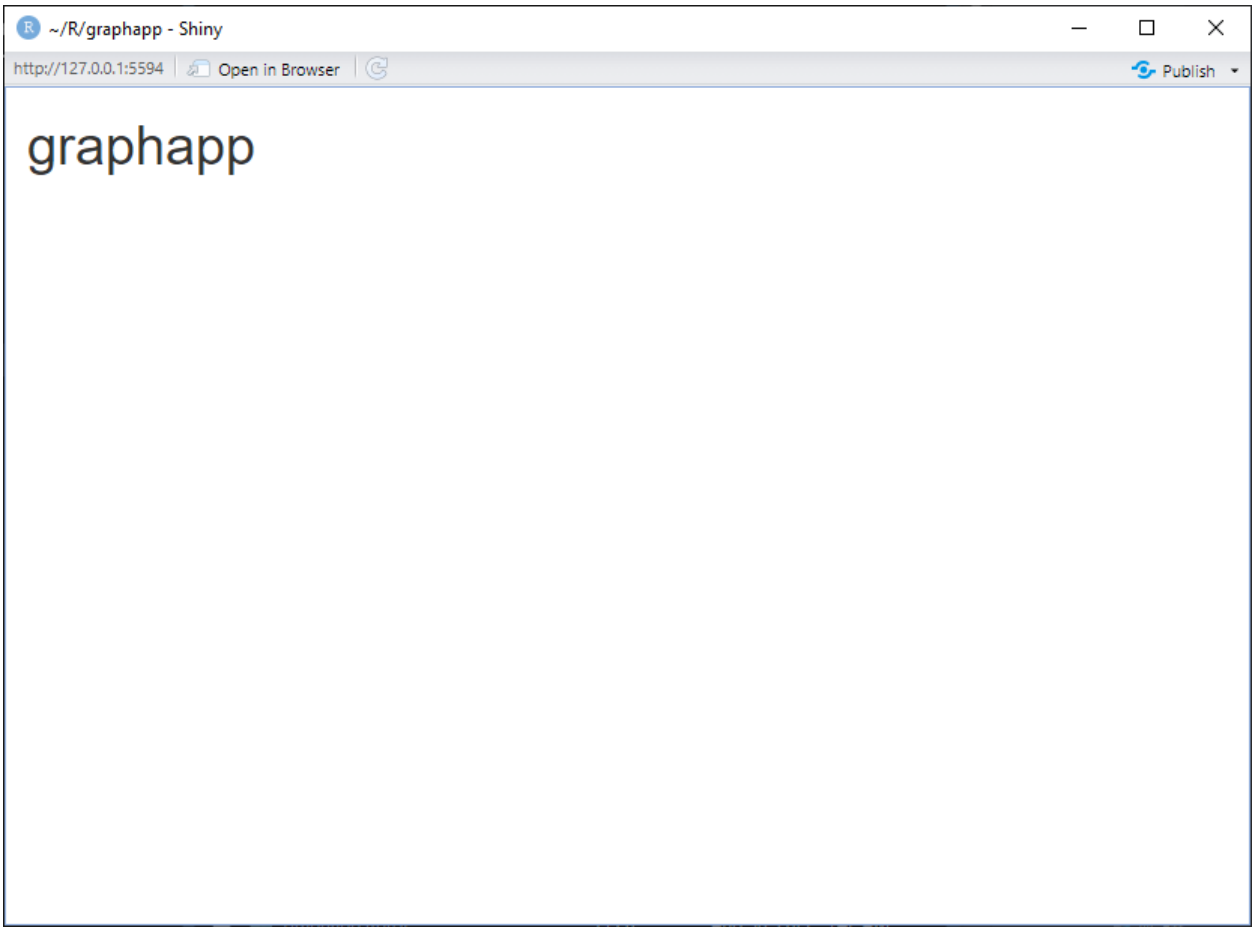

Fig.1: The first run of the application.

You can create your own header using the basic **h1** tag along with css on the UI side (**app_ui.R**), e.g.,

```
app_ui <- function(request) {
    tagList(
        # code snippet...

        # Your application UI logic
        h1("Graph Simulator", style="background-color: #23b5fe;
                                    color: white;
                                    padding: 20px 15px;
                                    margin: 0;
                                    margin-bottom: 10px")
    )
}
```

Given code is inserted in **app_ui.R** file and results in a differently styled header as seen in Fig.2.

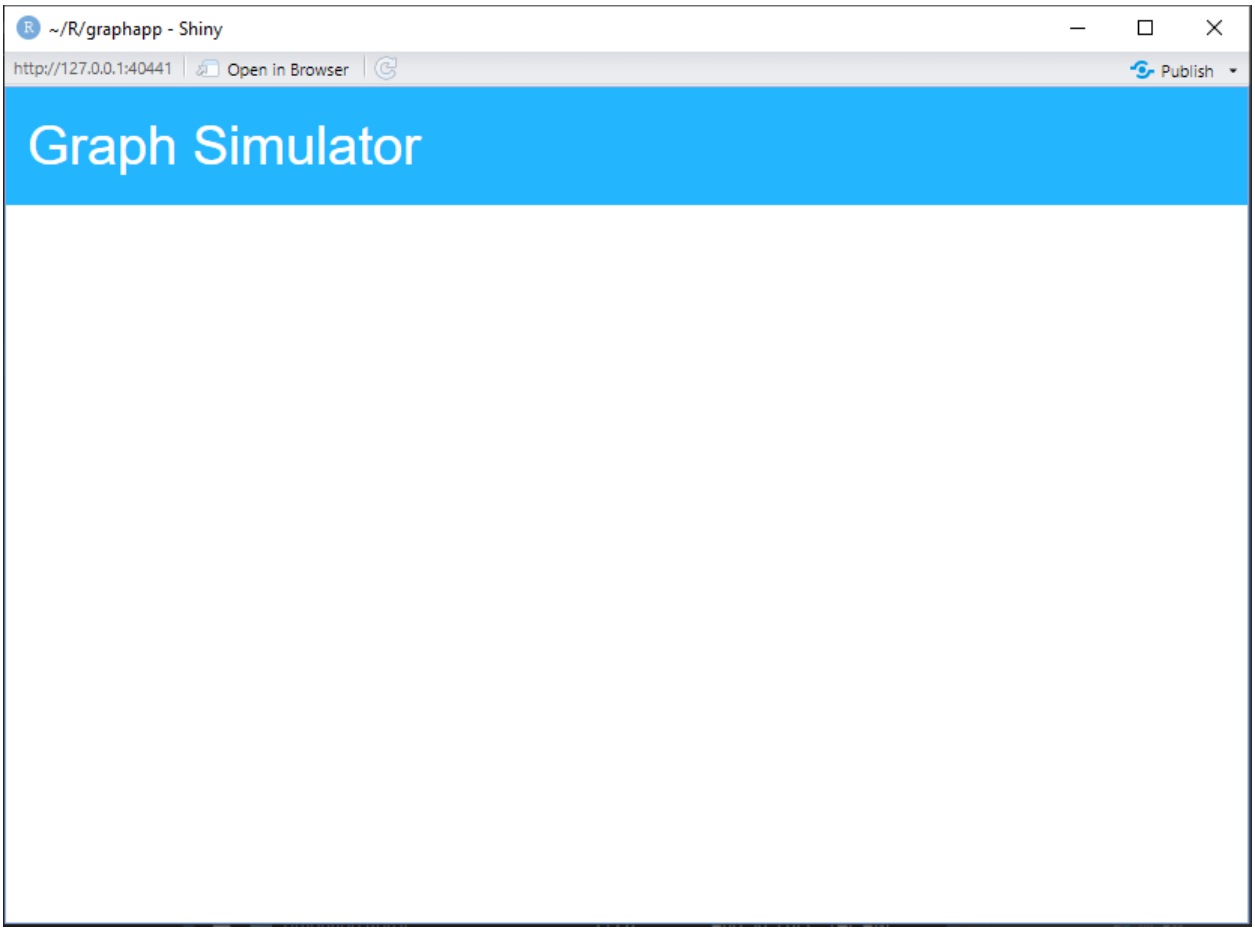

Fig.2: Updated header.

To divide the application into "loading page" and "simulation page", you can use Shiny's tabsets layout

```
app_ui <- function(request) {
    # code snippet...

    fluidPage(
        tabsetPanel(
            tabPanel("Input",
                     h3("Load network graph")
            ),
            tabPanel("Simulation",
                     h3("Run the simulation")
            )
        )
    )
}
```

The resulting layout is shown in Fig.3.

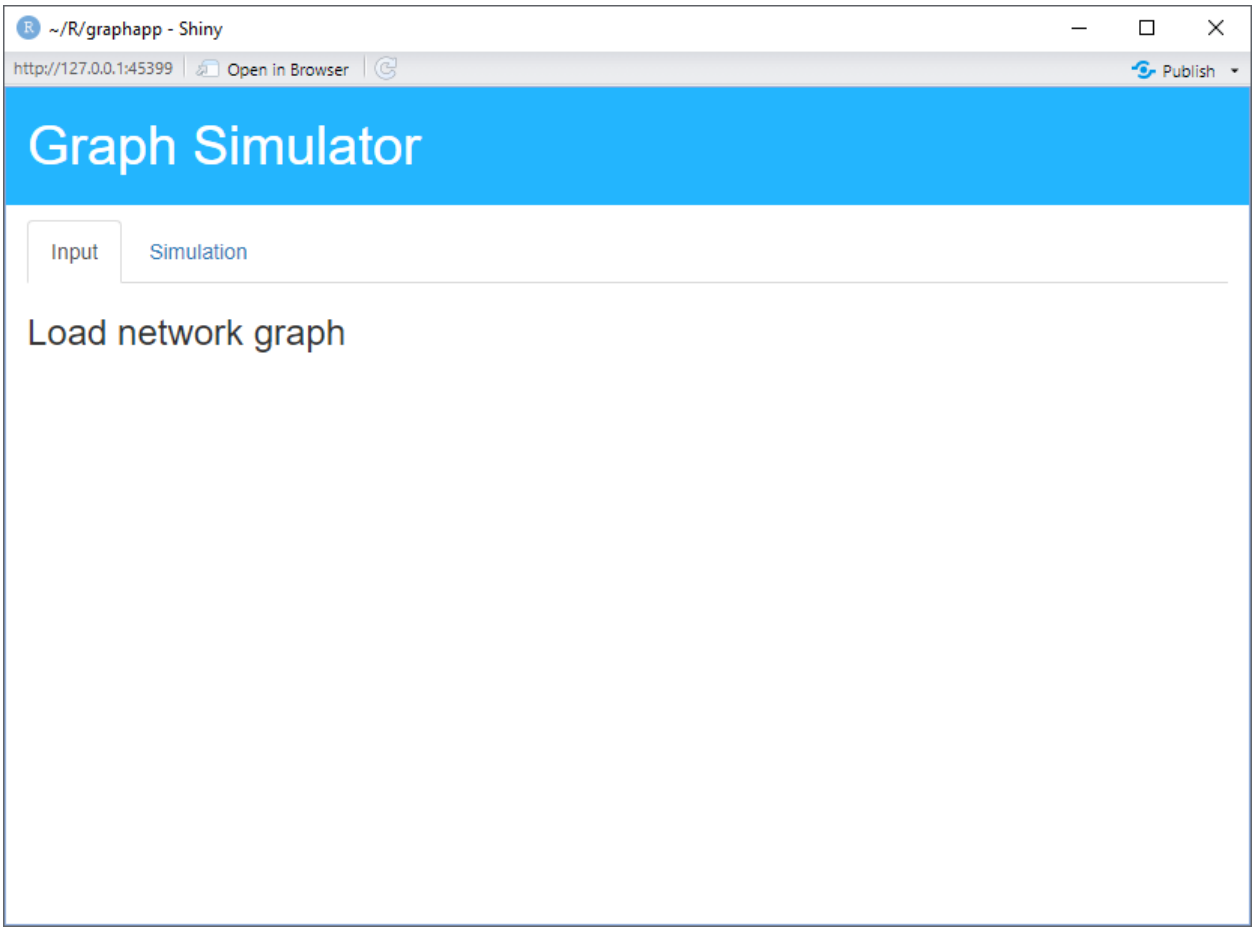

Fig.3: Layout of the application.

Now, you need to put some logic in code to upload a csv file and subsequently plot it. On the UI side, insert following lines of code in the **tabsetPanel()**

```
tabPanel("Input",
         h3("Load network graph"),
         fileInput("upload", label = NULL),
         plotly::plotlyOutput("network")
)
```

The **fileInput()** creates a button to open a file explorer. Once the file is selected, it is sent to the server side under the id **input***upload*∗∗.*The*∗∗*plotlyOutput*()∗∗*preparesthegroundfortheupcomingoutputfromtheserversideundertheid* ∗*output***network**. On the server side, i.e. **app_server.R**, the following needs to be done.

```
app_server <- function(input, output, session) {
  # dataframe variable(s) ----
  r <- reactiveValues(
   df_network = NULL
  )

  # Upload network ----
  observeEvent(input$upload,
```



```
  {
      r$df_network <- utils::read.csv(input$upload$datapath,
                                      header = TRUE)
  })

  # Plotly ----
  output$network <- plotly::renderPlotly(
  {
    req(r$df_network)
    plot_network(r$df_network )
  })
}
```

The data of the graph is prepared in a reactive way using **reactiveValues()**. You can add as many variables as you need. In this example, one variable $\mathbf{r}df_network**willbesufficient.Further,the**input**\mathbf{upload}$ is **NULL** when the app is launched. There is no data available, yet. When its state is changed, it will trigger the **observeEvent()**. An alternative is to use the **req()** which ensures the following lines of code are triggered after some data arrives (look into **renderPlotly()**). It blocks the **NULL** state to proceed. Thus, when the data $(\mathbf{r}df_network**)isready,theyaresenttothe**plot_network()**function.Theoutputofthe**plot_network()**representsdatainaformatsuitableforthe **plotlyOutput()** **andtheyarestoredin ** **output**\mathbf{network}$. At the end of the process, the **output$network** is used by **plotlyOutput()** on the UI side. As the **plot_network()** is quite complex, the full code can be downloaded here plot_network() code. Feel free to browse through the given code.

An example CSV file can be downloaded here. In short, the CSV file consists of 5 columns: **from**, **to**, **weight**, **level**, and **set**. Columns **from** and **to** contain the name of nodes connected by a single edge. The **weight** represents the weight of each edge. The **level** holds the levels of each edge, i.e. that the given edge belongs to the set of "the first floor" ("the first row"), "the second floor" ("the second row"), and so on. The **set** represents the common sets of the edges within "one floor" ("one row"). Different sets of one row will be drawed by different color. So, after the **example_network.csv** is uploaded into the app, the graph of the network is plotted, see Fig.4.

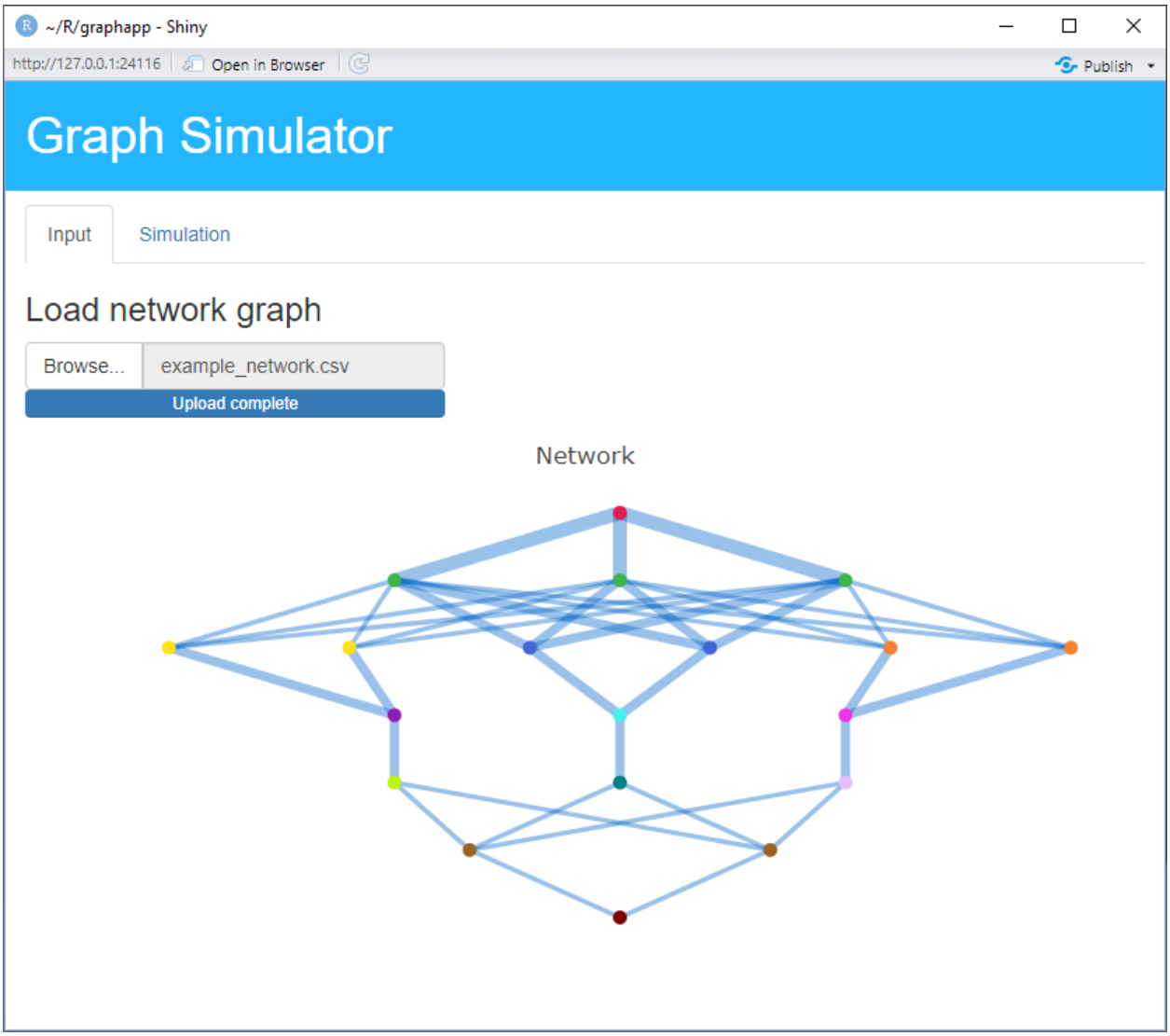

Fig.4: Loaded graph.

At this moment, you have prepared the input data for the simulation. Thus, the logic for the simulation along with the simulation function has to be created. As it was said above, the input for the simulator is the graph itself and the number of people that will go through the graph. To do so, you can use the following code on the UI side into **tabsetPanel()**.

```
tabPanel("Simulation",
    numericInput("input_people",
                 label = "Set the number of people that should enter the
                          network.",
                 value = 100,
                 min = 10,
                 max = 5000,
                 step = 1
    ),
    textOutput("error_people"),
    actionButton("run",
```



```
                label = "Compute simulation"
    ),
    h3("Simulation of the graph"),
    textOutput("error_graph"),
    plotly::plotlyOutput("simulation_plot")
)
```

By the **numericInput()**, the number of people will be set. The minimum number of people is set to 10 and the maximum to 5000. If the number of people will be out of range, the error will be triggered on server side and outputed into **textOutput("error_people")**. The **actionButton()** will start the simulation. The simulation will be plotted by **plotlyOutput()**. Moreover, the **textOutput("error_graph")** will output an error message that the button was clicked but there is no graph loaded.

The texts in **textOuput()** has black font as default. To change it, you have to add **tags***style* ∗ ∗*into* ∗ ∗*tags***head** on the UI side. Add `shinyjs::useShinyjs()`as well, it will be used later.

```
tags$head(
    # some code snippet...

    tags$style("#error_people, #error_graph { color: red; }"),
    shinyjs::useShinyjs()
)
```

On the server side, prepare reactive values for the simulation.

```
app_server <- function(input, output, session) {
    # code snippet...

    # variables for simulation
    r_simulation <- reactiveValues(
        continue = NULL,
        simulation = NULL,
        animation = NULL
    )
}
```

Then, observe if the **actionButton()** was clicked.

```
app_server <- function(input, output, session) {
    # code snippet...

    # observe if action button was clicked
    observeEvent(input$run,{
        r_simulation$continue <- NULL
        r_simulation$simulation <- NULL
        r_simulation$animation <- NULL

        shinyjs::disable("run")

        r_simulation$continue <- TRUE
    })
}
```


As you can see, all the variables are reset after button was clicked. The `shinyjs::disable("run")` disables the action button so you will not be able to click on it until after the computation of the simulation is done. The `r_simulation$continue <- TRUE` tells the server that the algorithm for the simulation can start. The simulation reads as follows: If the **r__simulate***continue**is**TRUE**, check if the **input***input***input__people** is within the range. If not, output the error. If yes, check if the network is loaded and there is an exit (sink) vertex in it. If something is missing, output the error. If it is successful, compute the simulation and store it into **r__simulation$simulate**.

The **check__network()** function is part of the **plot__network()** code which was mentioned above. The simulate_flow() function is available here: simulate_flow(). Do not forget to add `@import shinyjs` at the top of **app__server.R**.

```
app_server <- function(input, output, session) {
    # code snippet...

    # Simulation ----
    observeEvent(r_simulation$continue,
    {
        # Check input_people ----
        if (input$input_people > 5000 |
            input$input_people < 10)
        {

            # Output error if it is out of range
            output$error_people <- renderText("Input people should be
                                               integer between 10 and
                                               5000!")
            shinyjs::enable("run")
            input_people <- NULL
        } else {
            input_people <- round(input$input_people)
        }

        # Check network ----
        if (is.null(r$df_network))
        {
            # output error if there is no graph loaded
            shinyjs::enable("run")
            output$error_graph <- renderText("The graph is not loaded!")
        } else {
            network_check <- TRUE
            if (check_network( r$df_network ) > 0)
            {
                network_check <- FALSE
                shinyjs::enable("run")
                output$error_graph <- renderText("The network definition
                                                  is missing exit.")
            }
        }

        req(input_people, r$df_network, network_check)
        # if successful, reset error texts
        output$error_people <- renderText("")
        output$error_graph <- renderText("")
```



```
        res <- simulate_flow(df_edges = r$df_network,
                             input_people = input_people)
        r_simulation$simulation <- res[[1]]
  })
}
```

Finally, you have to output the animation. It can be done by following code.

```
app_server <- function(input, output, session) {
    # code snippet...

    observeEvent(r_simulation$simulation,
    {
        print("Animating...")
        r_simulation$animation <-
            animate_simulation(df_edges = r$df_network,
                               r_simulation$simulation)
    })

    observeEvent(r_simulation$animation,
    {
        output$simulation_plot <- plotly::renderPlotly(
        {
            r_simulation$animation
        })
        shinyjs::enable("run")
    })
}
```

In other words, if the **r_simulation$simulation** is ready, prepare the animation **by animate_simulation()**, which can be downloaded here animate_simulation(). If the animation is ready, output it by **plotly** and enable the action button.

Congratulations, everything is ready to run the app. Type `golem::run_dev()` into the R console and try it yourself. The app should look like as in Fig.5, below.

4.1 Adjustments based on customers review

There was an update from the users that they want to add following features:

- create and edit graph within the application,
- export the graph into a csv file,
- add special events that may occur in the simulation (e.g. lunch break).

Such application already exists. You can try it yourself at https://shiny.vsb.cz/app/nts-shiny. The source codes are available here NTS backend and NTS Shiny. It extends the functionality of the app above. The **bs4dash** features was used to easily style the app.

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, the Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, the United Kingdom, France, the Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, the Republic of North Macedonia, Iceland, and Montenegro. This project has received funding from the Ministry of Education, Youth and Sports of the Czech Republic (ID:MC2101).

