Paper PP05

Back to the drawing board
How to quickly mock-up and test your application user interface

Barbara Mikulášová, Katalyze Data, Oxford, United Kingdom

ABTRACT
The buzz around the {shiny} package has only increased over the past year. Now, more and more R programmers use interactive dashboards to share their data insights with a diverse stakeholder audience – from their colleagues to project managers, sponsors, and clients.

But even for seasoned R programmers, the road to a robust and scalable application is a treacherous one. Prototyping, developing, and maintaining web applications require its own set of skills. A solid knowledge of the {shiny} package code is only the beginning.

Any dashboard or application development is a resource intensive process, and if not planned carefully, it can cause the team significant delays on the project deliveries, or worse, suboptimal user acceptance rates.
Quick and targeted prototyping can prevent many of these situations from happening. The application mock-up allows the R teams to identify the top priority features, and to carry out user acceptance tests in the early stages of the application development.

This paper is aimed at fellow data scientists and statisticians who want to start implementing {shiny} applications in their workflow but have little to no background in web application development. The readers will see strategies for creating and testing a quick user-interface mock-up design using {fakir} and {shinipsum} packages. With the new tricks in their arsenal, the attendees will know how to save time on their projects and improve user experiences by identifying, mocking, and implementing high-priority features from the get-go.

INTRODUCTION
The buzz around the {shiny} package has only increased over the past year. This popular R package provides a framework for building interactive web applications in R. Since its release in 2012 by Posit, a small universe of complementary packages further enhancing this framework, its maintenance, and user interface styling has been developed. Now, more and more R programmers use interactive dashboards to share their data insights with a diverse stakeholder audience – from their colleagues to project managers, sponsors, and clients. Regardless of the R developer's exposure to traditional web application frameworks and principles, anyone with solid knowledge of R could create a {shiny} application. However, this also introduced new design challenges which statistical programmers and analysts were not exposed to when working with more traditional R programming tasks.

This paper is aimed at data scientists and statisticians interested in implementing {shiny} applications in their workflows, even without a background in web application development. It will discuss the importance of prototyping user interface (UI henceforth) layout in application development, and how two R packages, {shinipsum} and {fakir}, can greatly accelerate the prototyping stage and improve clarity of the final design. Both packages are available on CRAN and are actively maintained by their creators, ThinkR group. To learn more about their work, please see their GitHub page (see the reference section).

As programmers, we love nothing more than to experiment with our code and see our creations in production. However, the web application development requires a different set of skills and designing approaches than an average statistical computing script in R.

IMPORTANCE OF PROTOTYPING IN UI DEVELOPMENT
Sufficient time must first be dedicated to conceptualising the application and its server logic modules. After thorough discussions with the end users, the development team needs to agree on the most appropriate sequence of feature development, modularization of code, the definition of business logic for the application, suitable scalable framework for enterprise grade {shiny} application and a lot more. These decisions are difficult to make with a theoretical image of an application in mind. It is the prototyping stage which gives space and time to experiment with the layout and the interconnectedness of the application's features. It is here that the developers and the users alike will see their ideas manifested into reality for the first time.

Of course, the prototyping stage can be very resource and time intensive. The peruse of the final application version requires the development team to create as many versions of the core application as possible in a short timeframe. Without the final prototype, neither the server logic mapping nor the UI design is finalised which will interfere with the team's ability to accurately assess the degree of complexity of the application and the next logical steps in its development. The importance of the prototyping stage should under no circumstances be underestimated.

Luckly, the two R packages, {shinipsum} and {fakir}, can make the process easier and faster. The packages are easy to use and have great integration with other R packages, notably the various visualisation libraries such as {ggplot2} or {dygraphs}. The following sections describe how the different functionalities from these packages can be used for fast and effective prototyping.

PLACEHOLDER UI ELEMENTS WITH {SHINIPSUM}

The {shinipsum} package was inspired by Loren-Ipsum, a famous dummy text block used to test quality of prints and web designs. In the cases when the visual design and positioning of various front-end elements take priority, the use of a nonsensical text blocks allows the testers to focus their attention on the intuitiveness of the layout instead on the content details. This is exactly what {shinipsum} package allows us to do.

With a single line of code (or three for customised elements), developers can generate placeholder UI elements without painstakingly crafting example graphs, text blocks, or statistical outputs. This not only saves the developers time which can be used to improve the UI layouts but also allows them to quickly change, move, and modify the placeholder elements on demand. Let's look at some examples.

Every UI element defined in the `ui()` function also needs a function defined in the `server()` function to generate the output which then populates the application on the front-end. We often think of these applications in pairs. One example from base {shiny} code is `plotOutput()` and its twin `renderPlot()` functions which generate a plot output element. An in-depth explanation of the {shiny} code is beyond the scope of this paper but there are many free resources written on this topic. The {shinipsum} likewise provides twin functions to be used in the front-end `ui()` function and the back-end `server()` function.

The below code shows how to generate a random dygraph as a placeholder element for an example dashboard using `dygraphOutput()` function inside the `ui()` function and its required server-side `renderDygraph()` function. The second function takes in a function which defines and generates the desired placeholder digraph plot. Here the code is also straightforward. Simply call `random_dygraph()` function to randomly generate the plot. This function takes additional arguments to further customize the plot such as adding titles or axis manipulation. Keep in mind that the code below is a snapshot of a larger code chain required to build an example application and does not contain all the closing brackets.

UI SIDE:

```
# Tab-sets layout
miainPanel(
    tabsetPanel(
        tabPanel("Sales",
            fluidPage(
                fluidRow(
                    column(width = 12
             ,dygraphOutput("digraph")
                           )
                      )
                    )
                )
              )
             )
```

SERVER SIDE:

```
output$dygraph <- renderDygraph({
       shinipsum::random_dygraph(main = "Some Important Graph")
})
```

The functions from {shinipsum} package are easy to work with especially thanks to the logical naming of the functions. The calls required in the UI side always end with `<type>Output()`, their twin rendering function on the server side always start with `render<Type>()`, and the function which generates the corresponding placeholder element always starts with

`random_<type>()` pattern. This makes it easy to lookup the function in the documentation or to quickly change the element type in the code. Additionally, the random plots can be further modified with {ggplot2} function calls. For stylistic improvements of the UI, a well-structured CSS file goes a long way.

SYNTHETIC DATA SETS WITH {FAKIR}

With the visual side of the UI layout taken care of, let's discuss the second prototyping package, {fakir}. This package is designed to randomly generate synthetic tabular datasets. In the time of writing this paper, the {fakir} version 0.2.0.9 can generate 8 various data sets in two languages. Originally, the package was introduced to help R programming educators to access a variety of data set to demonstrate programming and visualisation techniques in R but thanks to its flexibility, it soon found its way to {shiny} developer's toolbox. Being developed by the same creators as {shinipsum}, ThinkR group, this library incorporates nicely with the rest of the prototyping tools.

Synthetic datasets are ideal for cases where application data is unavailable, access is delayed or large number of datasets are needed to test data validation steps, interactive filters, load handling and scalability or even mapping of the application's reactive logic. With {fakir}, the developers can create a variety of data sets with a single line of code.
Let's have a look at an example. In this case, we want to test the interactive user filters on the sidebar of the application. Its unique choice values should be automatically updated when a new data set is imported. Firstly, `fake_products()` function was used to generate a table of fictitious product records with 100 entries. Since the data is randomly generated, the call requires a seed to ensure the dataset's reproducibility.

```
# Create Fake Products Data Set
fake_df <- fakir::fake_products(100, seed=1)
```

Now the data set can be used as an input for the drop-down filter and to generate a filtered table on the front-end as shown in the code below.

UI SIDE:

```
selectInput("category",
            "Product category",
            choices = unique(fake_df$category),
            selected = unique(fake_df$category)[1]
      )
```

SERVER SIDE:

```
output$table <- renderTable({
      dplyr::filer(fake_df, category == input$category)
})
```

Just like with {shinipsum} package, the function names from {fakir} also follow standardised naming convention which makes them easy to remember and work with. Every call starts with `fake_<type>()` pattern followed by the type of the data set. The synthetic dataset categories for {fakir} version 0.2.0 were fake products, ticket clients, base clients, user feedback, map of France, and two fake transport data sets. This is enough to cover a variety of situations that the {shiny} developers need to test and account for.

TAKING APPLICATION PROTOTYPES TO THE NEXT LEVEL

The main argument for using one of both packages is to speed up the initial prototyping stage of the application where numerous iterations of different UI layouts need to be tested and then adjusted before the developers and the stakeholders agree on the finalized version of the web application. With more UI layout candidates at hand – and all generated within a shorter time window, the developers can apply their resources to planning and testing tasks.

Building a prototype with dummy elements enables early A/B testing and user feedback gathering, which can guide development. This will allow you to adjust the scope of the required features as the end users will often request feature change at this stage. Thanks to the flexibility of the prototyping packages, making changes to the UI layout to mock a new set of features is a straightforward and comparatively easy task. This way, the developers can get a sense of the application's complexity and differentiate between the core functionalities and good to have features. This informs the

development strategy, helping to prioritise core features for version 1 and plan the sequence of future feature introductions.

On the server side, dummy datasets and placeholder mappings help developers identify code sections to isolate into separate functions for business logic or standalone modules for application features. Having the server architecture mapped out early helps to keep the main `ui()` and `serve()` functions clean, readable, and most importantly scalable. Lastly, the variety of synthetic data sets lets the developers to test and map the chains of reactive logic between the data and the different output elements of the dashboard. The topic of reactivity in the context of {shiny} applications is beyond the scope of this paper but there are many great resources on this topic online. Nonetheless, losing track of the reactive dependencies in a complex, multi-modular application is a common pitfall for {shiny} developers. This is why the mapping should be performed at the early stages and the prototyping R packages can help with just that.

CONCLUSION

In summary, this paper has described how {fakir} and {shinipsum} packages can accelerate the prototyping stage of {shiny} application development. These packages address common challenges faced by both new and experienced R developers.

The {shinipsum} package lets us generate visual dummy placeholder elements which are then used to populate the UI of the application. This process requires minimal code written which eases the programming burden placed on the programmers due to the iterative nature of the initial UI layout design prototyping. The {fakir} package is used to generate random tabular data sets which can be used to map data processing flows, validation steps, and the interactive user filters.

Both packages help developers to present a variety of mock-up design to the end-user for feedback fast which accelerates the time-to-production step. Additionally, having a finalized prototype application at hand, even if still populated by the generated dummy elements, grants the developers the opportunity to map out the front-end and the back-end architecture of the application. Armed with this knowledge, the team can better adjust the scope of work and resources needed for the application's development and maintenance. By incorporating {shinipsum} and {fakir} in the early stages of development, R programmers can streamline the prototyping process, ultimately delivering more user-friendly and efficient applications.

ACKNOWLEDGMENTS

The team from ThinkR has greatly contributed to the R community and provided Shiny developers with easy-to-use tools for fast application prototyping. Their work is available to view on their GitHub page and aspiring R programmers are welcomed to contribute to their existing code base via pull requests. As the author of this paper, I would like to relay my gratitude to the creators of the two featured packages.

REFERENCES

https://github.com/ThinkR-open
https://github.com/BarbMik

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Barbara Mikulášová
Katalyze Data
11 Wesley Walk, Witney OX28 6ZJ
barbara.mikulasova@katalyzedata.com
https://katalyzedata.com/