Paper DV01

Reduce, Reuse, R Shiny: A framework for building generalized and extensible R Shiny applications

Nathan Kosiba, Cytel Inc., Durham, NC, USA

ABSTRACT

While building Shiny applications to support study activities has become more common in the industry, most of them end up left behind upon study conclusion or have to be heavily modified to use again. What if we coded applications once and used them across a wide range of studies to support the same activities? By implementing application development concepts such as data flexibility, generalizing core components, and utilizing application metadata, we can achieve that very question. While these additions alone won't make applications suddenly able to be deployed for every study in your portfolio, by building applications with these concepts in mind the process of adding a new study can be much improved. By utilizing the application development concepts above to enable reusability across studies, time can be spent improving their functionality and adding new features instead of coding the same thing again and again.

INTRODUCTION

In this paper, development concepts will be discussed with the goal of outlining a robust framework for developing Shiny applications. Through the implementation of these concepts in development workflows, generalized codebases can be produced that can create Shiny applications with ease and flexibility. The main concepts included are data pipelines, application metadata, and extensible application views.

While data for a Shiny application can be loaded in many ways, adding flexibility to the location, method, and verification of application input data greatly increases the ease of application reuse and configuration. Application metadata is one of the core components of this methodology of development. Metadata is just data about data; it helps give guidelines on how data should be used and how that data should be summarized and visualized. Coding an application for extensible use means coding it to not just be used with the data and specifications you have currently, but to have it be able to take a variety of inputs and create a variety of outputs. This means that while you may be making a scatter plot of data for one study, the same view might need to include a violin plot for another.

While all of this sounds great, it is not an easy or simple process to create extensible applications. When coding for extensibility, the focus is not only on creating views that will display what the user wants but also on how to make them flexible enough to display what different users will need. By using metadata driven applications, this can be achieved so that users can see what they need on any given study. Metadata can be used to specify listing logic so that any listing can be created within the application. It could also specify which plots to create and which variables, filters, and presets to use to create them. By implementing these concepts, Shiny applications can be created that are able to be reused across studies..

SHINY APPLICATIONS IN CLINICAL TRIALS

Shiny applications have begun to become more common to support a variety of clinical trial activities across a multitude of functional areas. These applications often start as proof of concept pilots on a single study or for a small group within a functional area. This development stage is crucial for developing ideas and requirements into a functioning application or application codebase. After a proof of concept has been completed, many times the application code will be copied or moved to the next study, updating what needs to be updated and making improvements along the way. Before too long, the application being used on one study may look little like what another study is using. This fractured development leads to work being done countless times and an inconsistent set of features that may lead to user confusion.

While creating specific views has advantages, with a flexible framework the great views you build for one team can benefit all of the teams using your application. By coding an application to use metadata to

identify the inputs and specify your outputs, the same framework can be used throughout many use cases. With more Shiny applications being created, the opportunities to build reusable applications are abundant. With application development teams usually being small, being able to easily create applications for individual studies or configure a generalized application to use new study data can increase the reach and effectiveness of these teams.

KEY APPLICATION DEVELOPMENT CONCEPTS

Before diving in to how to create applications based on a reusable framework there are a few key concepts to outline that will be used throughout. These concepts are ones that might not be the first thought of a Shiny developer but are key components to creating extensible applications. Concepts included in this section are system environment variables, reactivity, and Shiny modules.

ENVIRONMENT VARIABLES

The first concept is the use of system environment variables. These are variables that can be set in the system environment to be used throughout any code that is run within the system. This can be a local coding session or a deployed application environment. These variables can be easily set in any environment and are just as easy to access. In deployed environments, system variables are usually set during set up either through a configuration menu as used in POSIT Connect or through a `.env` file that is deployed with the application. Additionally, if using locally or if there is a set up script to your deployment you can simply use `Sys.setenv("env_var_name")` to set the variable value for use later.

REACTIVITY

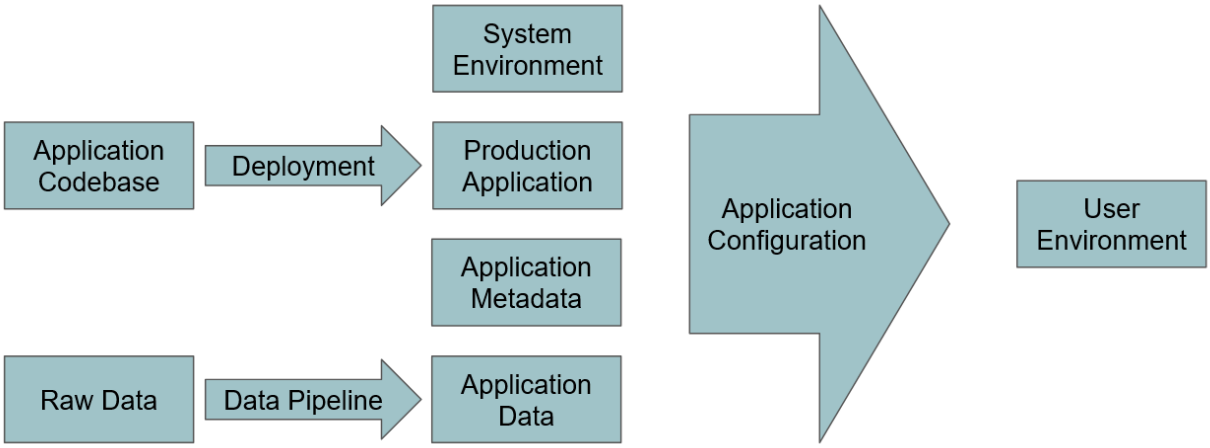
Reactivity is a key, but sometimes frustrating, part of building shiny applications. It is what lets us interact between elements within the application and store values that may change during application use. In the simplest form, reactivity is the ability to take a change in an application element such as a button click and turn it into an action that needs to be taken. This concept of being able to react to actions taken by a user or another system allows us to create networks of interactions within an application. This network allows us to communicate between individual elements and update shared data or choices. While this is a cursory overview of reactivity, the key piece used in creating extensible applications is the network of interactions and shared elements.

SHINY MODULES

Shiny modules are a way of breaking up code into pieces for reusability and flexibility. Shiny modules are just a paired UI and a Server function that can be called numerous times within an application to create independent application sections. Shiny modules are namespaced using a unique ID to pair code elements and handle reactivity. While modules are a key aspect to any well organized Shiny application codebase, they are even more important when creating extensible applications. The ability to reuse the same code to create distinct elements through the use of modules and namespaces makes the creation of extensible views much more straightforward. To read more about Shiny modules or Shiny in general, the book Mastering Shiny by Hadley Wickham is a great resource.

APPLICATION FRAMEWORK

While creating a single Shiny application is straightforward and is a specific task, creating a reusable framework for creating Shiny applications takes more time and forethought. When thinking about creating such a framework, the first step is usually to think about the applications that it needs to support and what the functions of those applications are. The basic parts of most clinical trials Shiny applications are the input data, the application views and user interaction with those views, and any exported files that may be created. Each of these is treated differently when thinking about reusability and extensibility. The below diagram shows where each of these pieces fits into a shiny application that has been developed using an extensible framework.



DATA PIPELINES

One built-in advantage of building Shiny applications for clinical trials is that there are already data standards in place. While we may not always get to use standardized trial data, the data collected falls into a few categories with many datasets having structural overlaps. Additionally, the use of things such as formats and column labels give application developers easy access to accessibility information that may be useful for display. The first step to creating any Shiny application or framework is to identify the type of data that will be used and how to load that data into the application.

Data structures for clinical trials can be broken down into three main groups: subject level data, event data, and results data. Subject level data is things such as demographics or other data that is expected to have one record in a dataset per subject. This data is mostly categorical and can be summarized and displayed easily to create compelling views. Event data is data about events, past or present, that have some sort of date and some amount of terminology with it. These are great for summarization and creating hierarchical views that tell a story or give context. Results data includes any results gathered during the study, usually structured with a value tied to some test or score from the trial. These are key datasets that are usually of high interest to users for exploration.

The pipeline in the phrase “data pipeline” is the flow of data from the source to the application. Any work that is needed to prepare the data for application use needs to be included. While some of this pipeline may rely on others, to create a robust and extensible application the data pipeline needs to be well defined and reliable. These pipelines should be easily repeatable and need little maintenance throughout the course of a study. When creating data pipelines, it is best to work backwards and start by thinking about how the data needs to be structured for the application and then figure out how to map and configure the data to fulfill those needs. The processing of the data may change between pipelines but the end result being standardized for application use is critical. Once the data has been processed, we can now load it into the application and use it to create views for our users.

When using data in an application, the first idea might be to either include it with the deployed application or put a file path for reading the data in the server code. This leaves us with an inflexible input source or even one that calls for application redeployment to update the data. While this works for an application, there are easy ways to implement a flexible data source that allows for a variety of input sources and easy data updating. The two easiest ways to add flexibility to input data is through the use of system environment variables and application metadata. By just simply updating a file path to use a system environment variable the data source is now flexible and can be changed without needing to redeploy or update any application code. Below is a code chunk that shows how a simple change can enable the reading of study data from any study for an application through changing the environment variable.

```
# inflexible file path read
appData <- read.csv(file = "path/to/study/data/demog.csv")

# flexible system environment variable
appData <- read.csv(file = Sys.getenv("demog_path"))
```

APPLICATION METADATA

Though metadata is a concept familiar to clinical trials, it is often thought of as the outline of a dataset to be coded and not the configuration of an application. In standard dataset programming, metadata is used to outline the contents and presentation of a variable. In application programming, we use metadata to outline how to use the data we are given. Application metadata goes beyond file paths and user lists, it can tell the app what data is present, what visualizations are needed, and even what models to use. Without application metadata, the work put into data pipelines, extensible views, and customizable exports are only as useful for the exact instances they are coded for.

Application metadata can be stored in many forms from a JSON or other static file to an entire database or other storage solution. To easily find each element, a named list can be useful with named elements for key pieces and even nested named lists for more complicated configurations. Each element should clearly correspond to a configuration in the application. See the below example of application metadata for a demographics boxplot with filtering variables of treatment and site identifier, x-axis variable options of sex, race, and region, and y-axis variable options of age, height, and weight.

```
# metadata for demographics plot
{"demographics": {
  "plot_type": "box",
  "filter_vars": ["trt", "siteid"],
  "x_axis_vars": ["sex", "race", "region"],
  "y_axis_vars": ["age", "height", "weight"]
  }
}
```

Metadata doesn't have to be limited to just configurations for choices or variables, it can even store executable code for use within the application. If, for example, you had a plot that included outlier detection, you could use metadata to put in a model to use to detect outliers. While this is more advanced than plot configuration, through the use of `eval_tidy()` code from the metadata can be executed within an application. This opens an endless amount of possibilities for customization and flexibility for every piece of an application. While anything is possible, developers still must keep usability from a user and developer point of view in mind. Not everything has to be metadata and there may still be some explicit references within extensible applications. The goal is to be flexible enough to be reusable but not so open-ended that configuration is an endless headache.

EXTENSIBLE APPLICATION VIEWS

With a properly configured data pipeline running, your application is ready to display that data to users. A view in an application is a page of that application that lets the user view the data in a specific manner. Views can be as simple as outputting a dataset and as complicated as the developer can imagine and build. Views are normally made up of a few parts, each with their own challenges and techniques. For this section we will talk about user interfaces, displays such as plots or tables, and informative elements to go along with the data displayed in the view.

When building an application for extensibility, user interfaces are crucial as they must be flexible enough to accept a variety of input options but structurally sound. While views might always have certain interactive elements such as axis or plot selections, they may need any number of filtering or grouping variable inputs. While these normally would just be individually coded, when building an extensible view they may need to be generated using metadata. Creating generative elements may seem like a daunting task but with a little bit of creativity and the right tools it becomes easy.

The first shift is to use UI containers created using `uiOutput()` instead of explicitly creating UI elements using `selectInput()` or other input functions. This allows us to create a space to put UI input elements that are created in the server function. While this may complicate the basic case of using the Shiny application for one study it opens up the ability to use the same code for a variety of studies. When paired with application metadata, the ability to create whatever filter, grouping, or other variables needed allows us to reuse views for a multitude of purposes.

The most fun part of creating extensible applications is the ability to use different types of plots when the need arises. While a simple box plot might be the initial best fit, the ability to switch out for a simple line plot to see trends or a violin plot to visualize the data slightly different can create truly custom

applications. The ability to change charts isn't new, by simply adding a selection and using a few if loops chart selection is right there for the user. The real extensibility comes from being able to create concise views in a flexible manner. Being able to configure views to show the data in a specific way without adding extra user interactions can help with both presentation and accessibility. While some users may want endless choices, others may want a more curated experience and it is up to the developer to find a balance in design.

One of the tougher parts of building extensible views is the reactivity involved in extensible views. In traditional views, the input and output elements are explicitly defined with the ability to reference these events cleanly and with little extra thought. When building extensible views the use of elements generated by code makes referencing them difficult and is one way point where extra care must be taken. Additionally, when creating elements using server code extra care must be taken to ensure namespacing is correct. When creating these extensible views I find it best to go element by element and create the reactive network, building the reactive network piece by piece.

There are a few nuances of referencing reactive elements that should be pointed out for creating extensible views. The most basic one is how you reference an input or an output object. Traditionally this is done using `input$element_id` but for most cases our identifiers for our elements are constructed with code so most referencing will use the less used `input[[element_id]]` syntax where the identifier may be a string created from other values. Another trick that is very useful is using `bindEvent(once = TRUE)` which allows us to create listeners that fire once and can be used nicely on application startup. This can be used to populate UI containers, update inputs already created with data values, or do any other initial tasks that are needed to correctly configure the application. The last thing to point out is that by creating observers and using `req()` to create stops for required elements it can help not trigger listeners that don't have the necessary pieces in place yet. While these are small things by using them when creating extensible views it can make managing reactivity much easier.

CODE EXAMPLE

Below is a basic example of applying all of the concepts discussed in this paper. While the code may not fully demonstrate everything talked about, it should give a starting point for anyone interested in building Shiny applications in a more flexible and extensible manner.

```
# Traditional UI and server reactive code
ui <- function(id) {
  ns <- NS(id)
  tagList(
    selectInput(ns("trt_filter"), "Filter Treatment",
                c("Placebo", "High Dose", "Low Dose")),
    selectInput(ns("x_axis_var"), "X-Axis Variable",
                c("sex", "race", "region")),
    selectInput(ns("y_axis_var"), "Y-Axis Variable",
                c("age", "height", "weight")),
    plotOutput(ns("demographics_plot"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    demographics_data <- read.csv(file = Sys.getenv("demog_path"))
    output$demographics_plot <- ({
      demographics_data %>%
        filter(trt %in% input$trt_filter) %>%
        ggplot(aes(input$x_axis_var, input$y_axis_var)) +
          geom_boxplot()
    )}
}

# Extensible UI and server reactive code
ui <- function(id) {
  ns <- NS(id)
```

```
  tagList(
    uiOutput(ns("filtering_output")),
    selectInput(inputId = ns("x_axis_var"), label = "X-Axis Variable",
                choices = NULL),
    selectInput(inputId = ns("y_axis_var"), label = "Y-Axis Variable",
                choices = NULL),
    plotOutput(ns("demographics_plot"))
  )
}

server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns
    rvs <- reactiveValues(
      subset_demo = list()
      )

    # read in demographics data and metadata flexibly
    demographics_data <- read.csv(file = Sys.getenv("demog_path"))
    app_meta <- reactive({read_json(file = Sys.getenv("meta_path"))})

    # turn metadata into selections
    output$selections_output <- renderUI({
      for(var in names(app_meta()$demographics$filter_vars)) {
        selectInput(ns(paste0("filter_", var)),
                    choices = sort(unique(demographics_data[[var]])))
      }
    )}

    # update inputs using metadata
    observe({
      updateSelectInput(session, inputId = "x_axis_var",
                        choices = app_meta$demographics$x_axis_vars)
      updateSelectInput(session, inputId = "y_axis_var",
                        choices = app_meta$demographics$y_axis_vars)
    )} %>%
      # bind to the app meta reactive object and ignore init and NULL
      # this gives us an event that only triggers on application start
      bindEvent(app_meta, ignoreNULL = TRUE, ignoreInit = TRUE)

    # subset data based on filters
    observe({
      temp_demo <- demographics_data
      for(var in names(app_meta()$demographics$filter_vars)) {
        temp_demo <- temp_demo %>%
                      filter(var %in% input[[paste0("filter_", var)]])
      }

      rvs$subset_demo <- temp_demo
    )}

    # update output plot
    output$demographics_plot <- ({
      plot <- ggplot(rvs$subset_demo(),
                     aes(input$x_axis_var, input$y_axis_var))
      # if loop to switch plot based on metadata
      # this could be a function to be more organized
      if (app_meta()$demographics$type == "line") {
        plot +
          geom_line() +
```

```
          geom_point()
      } else if (app_meta()$demographics$type == "box") {
        plot +
          geom_boxplot()
      }
    )}
  })
}
```

CONCLUSION

While this paper provides an overview of the elements needed to build and use an application framework, to understand and apply them takes practice. Each of the concepts outlined have a place in Shiny application development and they are useful ideas to gain an understanding of. The ability to reuse applications and build fully flexible and extensible frameworks makes creating new applications trivial. This allows developers to spend more time updating and expanding the codebase rather than tracking down every string they need to replace to make the application work for a new study. While starting by building applications entirely using this framework is likely impractical, integrating these concepts and changing your thinking slowly will eventually evolve your workflow to create more extensible applications.

RECOMMENDED READING

Mastering Shiny by Hadley Wickham

CONTACT INFORMATION

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

Nathan Kosiba
Principal R Programmer
Cytel, Inc.
675 Massachusetts Ave
Cambridge, MA 02139
Nathan.Kosiba@cytel.com
www.cytel.com