Extending *iSEE*

Kevin Rue-Albrecht, Federico Marini, Charlotte Soneson, and Aaron Lun

2019-12-30

Contents

Preface

The Bioconductor *iSEE* package provides functions for creating an interactive graphical user interface (GUI) using the RStudio *Shiny* package for exploring data stored in *SummarizedExperiment* objects, including row- and column-level metadata (Rue-Albrecht et al., 2018). In this book we describe how to create web-applications that leverage builtin panels and develop new ones. We also present case studies to illustrate the development of custom panels.

Chapter 1

Panel classes

1.1 Overview

The types of panels available to compose an *iSEE* app are defined as a hierarchy of S4 classes.

- `Panel`*
  - `DotPlot`*
    - `ColumnDotPlot`*
      - `RedDimPlot`
      - `ColDataPlot`
      - `FeatAssayPlot`
    - `RowDotPlot`*
      - `RowDataPlot`
      - `SampAssayPlot`
  - `Table`*
    - `RowTable`*
      - `RowStatTable`
    - `ColumnTable`*
      - `ColStatTable`
  - `HeatMapPlot`

Some of those classes are "virtual" (indicated by *), meaning that they cannot be directly instanciated as panels in the GUI. Instead, virtual panel classes define families of panels that share groups of properties.

Virtual classes are meant to be used as the parent of one or more concrete classes. In contrast, concrete classes must define fully-functional panels that can be embedded in a GUI, interact with other panels, receive and process data, and generate an output such as a plot or a table, accompanied by a code chunk to display in the code tracker for reproducibility.

1.2 The Panel class

The top-most class is called `Panel`. It is a virtual class that defines the core properties common to any panel - existing or future - that may be displayed in the interface.

Refer to `help("Panel-class")` for more information about the slots and methods provided by this core class.

1.3 The DotPlot and Table panel families

The `Panel` virtual class is directly derived into two major virtual sub-classes:

- `DotPlot`
- `Table`

Those classes introduce properties that are specific to distinct subsets of panel types.

The `DotPlot` class introduces parameters specific to panels where the output is a `ggplot` object and each row in the data-frame is represented as a point in a plot.

The `Table` class introduces parameters specific to panels where the main output is a data-frame directly displayed as a table in the GUI.

In addition, the `HeatMapPlot` class defines a special concrete panel class that directly extends the `Panel` class, as it introduces a set of parameters distinct from both the `DotPlot` and `Table` panel families. This panel type is described in further details in a separate section below.

1.4 The ColumnDotPlot and RowDotPlot panel families

1.5 Built-in ColumnDotPlot panel classes

1.6 Built-in RowDotPlot panel classes

1.7 The ColumnTable and RowTable panel families

1.8 Built-in ColumnTable panel classes

1.9 Built-in RowTable panel classes

1.10 The HeatMapPlot panel class

This type of panel introduces parameters specific to panels where the output is a heat map, with each row representing a feature and each column representing a sample in the `se` object.

Chapter 2

The app server

2.1 Reactive objects

2.2 Persistent (non-reactive) objects

2.3 The app memory

The app `memory` is a list of instances created from available panel classes, which defines the order in which individual panels are displayed in the GUI.

2.4 The panel API

2.4.1 .cacheCommonInfo

Each individual panel type (e.g., *Reduced dimension plot*) and family of panel types (e.g., *Column dot plot*) defines a `.cacheCommonInfo` function.

This function is called for each panel instance in memory when the app is initialized. It allows the app to efficiently compute a single time common information that only depends on the input `se` object, and may be frequently reused during the runtime of the app.

Following the hierarchy of panel types, each call to the signature takes a panel instance `x` and the `se` object, and caches common information relevant to all instances of that panel type in the `se` object itself, before calling `callNextMethod()` to invoke the next parent signature.

The top-most signature - for the `Panel` class - returns the `se` object that contains all the cached information.

Note that this function only populates the cache for the first panel of each type; it is a no-op if the common cache has already been initialized.

2.4.2 .refineParameters

Each individual panel type (e.g., *Reduced dimension plot*) and family of panel types (e.g., *Column dot plot*) defines a `.refineParameters` function.

This function is called for each panel instance in memory when the app is initialized, and also when a new panel is added to the GUI. It inspects the parameters of a given panel instance, and replaces invalid parameters with sensible values for a given `se` object.

Following the hierarchy of panel types, each call to the signature takes an instance `x` and the `se` object, and first calls `callNextMethod()` to invoke the next parent signature, to refine generic parameters before processing specific ones.

The called signature ultimately returns the updated instance panel `x`, or `NULL` if the panel instance is not available for this app.

2.5 Initialization of the app server

The app server is initialized as soon as a valid `se` object is provided. This can be either in the call to `iSEE(se)` or using the Shiny file upload button in apps that were launched without providing the `se` arguments, e.g., `iSEE()`.

The `initialize_server` function takes the `se` object and the list holding reactive values used to trigger re-rendering of the GUI, as described above.

The very first step invokes the function `.sanitize_SE_input` on the `se` object. This function coerces the `se` to `SingleCellExperiment`, flattens nested DataFrames, adds row and column names, and removes other non-atomic fields. In addition, it also sanitizes the `SingleCellExperiment` object by moving internal fields into the column- or row-level metadata, making them visible in the *Column statistics table* and *Row statistics table* panels, respectively. The function returns both the sanitized `se` object that will be used by the app, and the list of R commands that will be displayed in the code tracker for users.

Next, the server invokes the `checkColormapCompatibility` function. This function takes the `se` object and the optioanl `colormap` provided to `iSEE()`, and carries out a number of compatibility checks between the two objects. The function collects a character vector of issue messages that are displayed - if any - as warning messages in GUI during initialization.

Next, the `.cacheCommonInfo` and `.refineParameters` are successively invoked on each panel instance initialized in the app memory. As described in a separate section above, the first function precomputes and caches information specific to the `se` object and frequently used throughout the runtime of the app. The second function ensures that each panel instance is initialized with valid parameters; it replaces any invalid parameters with sensible values for a given `se` object.

Next, persistent (non-reactive) objects are initialized:

- the app `memory` (see this section)
- the count of panels of each type, used to assign increasing ID to new panel instances
- the list of commands to display in the code tracker for each panel instance
- the list of data point coordinates selectable in each panel instance[1]
- a list of miscellaneous cached information[2]

[1]Data points downsampled for rendering speed performance remain selectable, even though they are not visible in the plot.

[2]The plot that contain the legend keys of *Heatmap* panels is currently cached as miscellaneous information retrieved separately when rendering the GUI.

Chapter 3

The plotting API

3.1 .getPlottingFunction

Each panel type available for use in the GUI defines a `.getPlottingFunction`.

This function is called within `.createRenderedOutput`, which is triggered by observers when the value of the panel input widgets are changed by users, or when a new panel is added to the GUI.

The `.getPlottingFunction` function inspects the parameters for a given panel instance, and uses the app `memory` of all active panels and parameters, the coordinates of data points in each plot panel, the `se` object, and the `colormap` to generate all the information necessary to render the outputs of this panel and those that depend on it.

For `DotPlot` panels, the output is a list that includes:

- the list of commands to display in the code tracker
- the coordinates of data points in the plot
- the `ggplot` object

For `Table` panels, the output is a `datatable`.

For the `HeatMap` panel, the function does not return any value. Instead it sets relevant elements in the `output` object of the Shiny session.

Chapter 4

Developing new panels

First, we need to load the *iSEE* package for this chapter. This action imports all the builtin panel class definitions, including the virtual class `Panel` that is the base class for any *iSEE* panel class.

```
library(iSEE)
```

We also set up an example using our favorite dataset, creating a `SingleCellExperiment` object with some precomputed dimensionality reduction results.

```
library(scRNAseq)
sce <- ReprocessedAllenData(assays="tophat_counts")
```

```
## snapshotDate(): 2019-12-27

## see ?scRNAseq and browseVignettes('scRNAseq') for documentation

## loading from cache

## see ?scRNAseq and browseVignettes('scRNAseq') for documentation

## loading from cache

## see ?scRNAseq and browseVignettes('scRNAseq') for documentation

## loading from cache
```

```
library(scater)
sce <- logNormCounts(sce, exprs_values="tophat_counts")
sce <- runPCA(sce, ncomponents=4)
sce <- runTSNE(sce)
```

4.1 Create a new S4 class

In the chapter Panel classes, we saw how each type of panel is defined as an S4 class, organised in a hierarchy that allows new panel classes to inherit sets of the properties and functionality from parent classes.

Then, developing a new panel type starts with the creation of a new class that inherits from the `Panel` class.

While it is possible to create a new panel class that directly inherits from the top-most virtual `Panel` class, this is the most advanced use case that we will describe in a later section.

Instead, new concrete panels classes can be rapidly derived from other concrete parent panel classes, using the inheritance relationships between classes to reuse properties and functionality defined in all of the parent classes.

The choice of a parent class for the new panel depends on the properties that we want that new panel to have. For instance, to create a panel that inherits all the functionality of the *Reduced dimension plot* panel type, we simply define a new class that extends the `RedDimPlot` class. We call that new class `RedDimHexPlot`.

```
setClass("RedDimHexPlot", contains="RedDimPlot")
```

Chapter 5

Add a constructor function

At this point, it is possible to create instances of the new panel class already. To facilitate this, new panels should provide a constructor function - named identically to the class - to accept arbitrary arguments controlling the initialization of new panel instances created by the `new` function.

Here, we define a simple constructor function that passes all incoming arguments to the `new` function *as is*.

```
RedDimHexPlot <- function(...) {
    new("RedDimHexPlot", ...)
}
```

At this point, we can already use instances of this new panel class in *iSEE* apps.

```
RedDimHexPlot1 <- RedDimHexPlot()
initial <- list(RedDimHexPlot1)
app <- iSEE(sce, initial = initial)
```

However, that would not be very exciting as instances of this new panel class would behave exactly like the those of the `RedDimPlot` class.

```
RedDimPlot1 <- RedDimPlot()
RedDimHexPlot1 <- new("RedDimHexPlot")
initial <- list(RedDimHexPlot1, RedDimPlot1)
app <- iSEE(sce, initial = initial)
```

Chapter 6

Set the panel name in the GUI

The panel class that we created so far also inherited the name of the parent panel class. In other words, instances of both classes are indistinguishable from each other in the GUI.

The name of each panel displayed in the GUI is defined by the `.fullName` method. To clearly distinguish the new panel class in the GUI, we overwrite this method to display a name different from the parent class.

```
setMethod(".fullName", "RedDimHexPlot", function(x) "Reduced dimension hexagonal plot")
```

With that, running `app` again now highlights how panels of the new class now display a different title from the parent class.

Chapter 7

Define the commands generating a plot output

Importantly, the API separates the generation of commands processing data from `sce` into a data-frame, from the generation of commands producing a `ggplot` object using the processed data-frame. If a new `DotPlot` panel class is meant to process data in the same way as its parent panel, only to display in a different way, it is then possible to overwrite only the `.generateDotPlot` method.

Importantly, the `.generateDotPlot` method requires two key arguments: `labels` provides the plot labels for each of the aesthetics in the plot data, and `envir` provides the environment in which the plotting commands are to be evaluated to produce the `ggplot` object.

In particular, `.generateDotPlot` can rely on certain environment variables promised by the contract. Above all, those include `plot.data`, the data-frame that contains one row per data point to display. Using those environment variables, `.generateDotPlot` can make decisions altering the plotting commands and the resulting `ggplot` object.

We invite readers to refer to the **"Generating the ggplot object"** section of `help(".generateDotPlot")` for more information.

As an example, we overwrite the `.generateDotPlot` function for the new `RedDimHexPlot` class to simply show the number of data point in the plotting area as a heatmap dividing the plane into regular hexagons. Notably, the function can immediately rely on the `plot.data` data-frame that is computed by methods defined for the parent class `RedDimPlot`. We also use the pre-computed aesthetic `labels` associated with each column of `plot.data`, while setting a fixed `"Count"` label for the `fill` aesthetic associated with the count of observation in each hexagonal bin.

```
setMethod(".generateDotPlot", "RedDimHexPlot", function(x, labels, envir) {
    print(labels)
    stopifnot(require(ggplot2))

    plot_cmds <- list()
    plot_cmds[["ggplot"]] <- "ggplot() +"

    # Adding hexbins to the plot.
    plot_cmds[["hex"]] <- "geom_hex(aes(X, Y), plot.data) +"
    plot_cmds[["labs"]] <- "labs(fill='Count') +"
    plot_cmds[["labs"]] <- sprintf(
        "labs(x='%s', y='%s', title='%s', fill='%s') +",
        labels$X, labels$Y, labels$title, "Count"
        )
    plot_cmds[["theme_base"]] <- "theme_bw() +"
    plot_cmds[["theme_legend"]] <- "theme(legend.position = 'bottom')"

    gg_plot <- eval(parse(text=plot_cmds), envir)

    list(plot=gg_plot, commands=plot_cmds)
})
```

Running `app` again highlights how the `RedDimHexPlot` panel fills each hexagonal bin with a color indicating the number of data points present in the corresponding area in the `RedDimPlot` panel.

Chapter 8

Case study I

8.1 Overview

In this case study, we will create a custom panel class to regenerate sample-level PCA coordinates using only a subset of points transmitted as a multiple column selection from another panel. We call this a **dynamic reduced dimension plot**, as it is dynamically recomputing the dimensionality reduction results rather than using pre-computed values in the `reducedDims()` slot of a `SingleCellExperiment` object.

8.2 Class basics

First, we define the basics of our new `Panel` class. As our new class will be showing each sample as a point, we inherit from the `ColumnDotPlot` virtual class. This automatically gives us access to all the functionality promised in the contract, including interface elements and observers to handle multiple selections and respond to aesthetic parameters.

```
library(iSEE)
setClass("DynRedDimPlot", contains="ColumnDotPlot",
    slots=c(NGenes="integer", Type="character"))
```

We add a slot specifying the type of dimensionality reduction result and the number of highly variable genes to use. Any new slots should also come with validity methods, as shown below.

```
library(S4Vectors)
setValidity2("DynRedDimPlot", function(object) {
    msg <- character(0)

    if (length(n <- object[["NGenes"]])!=1L || n < 1L) {
```

```
        msg <- c(msg, "'NGenes' must be a positive integer scalar")
    }
    if (!isSingleString(val <- object[["Type"]]) ||
        !val %in% c("PCA", "TSNE", "UMAP"))
    {
        msg <- c(msg, "'Type' must be one of 'TSNE', 'PCA' or 'UMAP'")
    }

    if (length(msg)) {
        return(msg)
    }
    TRUE
})
```

```
## Class "DynRedDimPlot" [in ".GlobalEnv"]
##
## Slots:
##
## Name:                NGenes                Type      ColorByColData
## Class:              integer           character           character
##
## Name:  ColorByFeatNameAssay ColorBySampNameColor      ShapeByColData
## Class:            character           character           character
##
## Name:         SizeByColData          FacetByRow       FacetByColumn
## Class:            character           character           character
##
## Name:               ColorBy  ColorByDefaultColor      ColorByFeatName
## Class:            character           character           character
##
## Name:        ColorByRowTable      ColorBySampName      ColorByColTable
## Class:            character           character           character
##
## Name:               ShapeBy              SizeBy        SelectEffect
## Class:            character           character           character
##
## Name:           SelectColor         SelectAlpha            ZoomData
## Class:            character             numeric             numeric
##
## Name:             BrushData       VisualBoxOpen       VisualChoices
## Class:                 list             logical           character
##
## Name:            ContourAdd        ContourColor           PointSize
## Class:              logical           character             numeric
##
```

```
## Name:            PointAlpha            Downsample             SampleRes
## Class:              numeric               logical               numeric
##
## Name:              FontSize        LegendPosition               PanelId
## Class:              numeric             character               integer
##
## Name:           PanelHeight            PanelWidth         SelectBoxOpen
## Class:              integer               integer               logical
##
## Name:       SelectRowSource       SelectColSource           DataBoxOpen
## Class:            character             character               logical
##
## Name:         SelectRowType        SelectRowSaved         SelectColType
## Class:            character               integer             character
##
## Name:        SelectColSaved    MultiSelectHistory
## Class:              integer                  list
##
## Extends:
## Class "ColumnDotPlot", directly
## Class "DotPlot", by class "ColumnDotPlot", distance 2
## Class "Panel", by class "ColumnDotPlot", distance 3
```

It is also worthwhile specializing the `initialize()` method to provide a default for new parameters:

```
setMethod("initialize", "DynRedDimPlot",
    function(.Object, Type="PCA", NGenes=1000L, ...)
{
    callNextMethod(.Object, Type=Type, NGenes=NGenes, ...)
})
```

8.3 Setting up the interface

The most basic requirement is to define some methods that describe our new panel in the `iSEE()` interface. This includes defining the full name and desired default color for display purposes:

```
setMethod(".fullName", "DynRedDimPlot", function(x) "Dynamic reduced dimension plot")

setMethod(".panelColor", "DynRedDimPlot", function(x) "#0F0F0F")
```

We also add interface elements to change the result type and the number of genes. This is most easily done by specializing the `.defineDataInterface` method:

```
library(shiny)
setMethod(".defineDataInterface", "DynRedDimPlot", function(x, se, select_info) {
```

```
    plot_name <- .getEncodedName(x)

    list(
        selectInput(paste0(plot_name, "_Type"), label="Dimensionality reduction Type",
            choices=c("PCA", "TSNE", "UMAP"), selected=x[["Type"]]),
        numericInput(paste0(plot_name, "_NGenes"), label="Number of genes",
            min=1, value=x[["NGenes"]])
    )
})
```

We call `.getEncodedName()` to obtain a unique name for the current instance of our panel, e.g., `DynRedDimPlot1`. We then `paste0` the name of our panel to the name of any parameter to ensure that the ID is unique to this instance of our panel; otherwise, multiple `DynRedDimPlot`s would override each other. One can imagine this as a poor man's Shiny module.

8.4 Creating the observers

We specialize `.createObservers` to define some observers to respond to changes in our new interface elements. Note the use of `callNextMethod()` to ensure that observers of the parent class are also created; this automatically ensures that we can respond to changes in parameters provided by `ColumnDotPlot`.

```
setMethod(".createObservers", "DynRedDimPlot",
    function(x, se, input, session, pObjects, rObjects)
{
    callNextMethod()

    plot_name <- .getEncodedName(x)

    # TODO: expose .define_protected_parameter_observers for developer use,
    # which would allow these steps to be a one-liner.
    type_field <- paste0(plot_name, "_Type")
    observeEvent(input[[type_field]], {
        previous <- pObjects$memory[[plot_name]][["Type"]]
        if (identical(previous, input[[type_field]])) {
            return(NULL)
        }
        pObjects$memory[[plot_name]][["Type"]] <- input[[type_field]]
        .requestCleanUpdate(plot_name, pObjects, rObjects)
    })

    num_field <- paste0(plot_name, "_NGenes")
    observeEvent(input[[num_field]], {
        previous <- pObjects$memory[[plot_name]][["NGenes"]]
```

```
        if (identical(previous, input[[num_field]])) {
            return(NULL)
        }
        pObjects$memory[[plot_name]][["NGenes"]] <- input[[num_field]]
        .requestCleanUpdate(plot_name, pObjects, rObjects)
    })
})
```

Both the `NGenes` and `Type` parameters are what we consider to be “protected” parameters, as changing them will alter the nature of the displayed plot. By using `.requestCleanUpdate()`, we instruct `iSEE()` to destroy existing brushes and lassos when these parameters are changed, as brushes/lassos made on the previous plot do not make sense when the coordinates are recomputed.

8.5 Making the plot

When working with a `ColumnDotPlot` subclass, the easiest way to change plotting content to override the `.generateDotPlotData` method. This should add a `plot.data` variable to the `envir` environment that has columns `X` and `Y` and contains one row per column of the original `SummarizedExperiment`. It should also return a character vector of R commands describing how that `plot.data` object was constructed. The easiest way to do this is to create a character vector of commands and call `eval(parse(text=...), envir=envir)` to evaluate them within `envir`.

```
setMethod(".generateDotPlotData", "DynRedDimPlot", function(x, envir) {
    commands <- character(0)

    if (!exists("col_selected", envir=envir, inherits=FALSE)) {
        commands <- c(commands,
            "plot.data <- data.frame(X=numeric(0), Y=numeric(0));")
    } else {
        commands <- c(commands,
            ".chosen <- unique(unlist(col_selected));",
            "set.seed(100000)", # to avoid problems with randomization.
            sprintf(".coords <- scater::calculate%s(se[,.chosen], ntop=%i, ncomponents=2);",
                x[["Type"]], x[["NGenes"]]),
            "plot.data <- data.frame(.coords, row.names=.chosen);",
            "colnames(plot.data) <- c('X', 'Y');"
        )
    }

    commands <- c(commands,
        "plot.data <- plot.data[colnames(se),,drop=FALSE];",
        "rownames(plot.data) <- colnames(se);")
```

```
    eval(parse(text=commands), envir=envir)

    list(data_cmds=commands, plot_title=sprintf("Dynamic %s plot", x[["Type"]]),
        x_lab=paste0(x[["Type"]], "1"), y_lab=paste0(x[["Type"]], "2"))
})
```

We use functions from the *scater* package to do the actual heavy lifting of calculating the dimensionality reduction results. The `exists()` call will check whether any column selection is being transmitted to this panel; if not, it will just return a `plot.data` variable that contains all `NA`s such that an empty plot is created. If `col_selected` does exist, it contains a list of character vectors specifying the active and saved multiple selections that are being transmitted. In this case, we do not care about the distinctin between active/saved selections so we just take the union of all of them.

Of course, this is not quite the most efficient way to implement a plotting panel that involves recomputation. A better approach would be to cache the x/y coordinates and reuse them if only aesthetic parameters have changed, thus avoiding an unnecessary delay from recomputation. Doing so requires overriding `.renderOutput()` to take advantage of the cached contents of the plot, so for simplicity, we will not do that here.

8.6 Finishing touches

For this particular panel class, an additional helpful feature is to override `.multiSelectionInvalidated`. This indicates that any brushes or lassos in our plot should be destroyed when we receive a new column selection. Doing so is the only sensible course of action as the reduced dimension coordinates for one set of samples have no obvious relationship to the coordinates for another set of samples; having old brushes or lassos hanging around would be of no benefit at best, and be misleading at worst.

```
setMethod(".multiSelectionInvalidated", "DynRedDimPlot", function(x) TRUE)
```

8.7 In action

Let's put our new panel to the test. We use the `sce` object, preprocessed in a previous chapter, including some precomputed dimensionality reduction results.

The plan is to create a (fixed) reduced dimension plot that will transmit to our dynamic reduced dimension plot. Setting up the iSEE instance is as easy as:

```
rdp <- RedDimPlot(PanelId=1L)
drdp <- new("DynRedDimPlot", SelectColSource="RedDimPlot1")
app <- iSEE(sce, initial=list(rdp, drdp))
```

Brushing at any location in the `RedDimPlot` will trigger dynamically recomputation of results in our `DynRedDimPlot`.

Chapter 9

Case study II

9.1 Overview

In this case study, we will create a panel class to dynamically compute differential expression (DE) statistics between the active sample-level selection and the other saved selections from a transmitting panel.

9.2 Class basics

First, we define the basics of our new `Panel` class. As our new class will be showing each gene as a row, we inherit from the `RowTable` virtual class. This automatically gives us access to all the functionality promised in the contract, including interface elements and observers to respond to multiple selections. We also add a slot specifying the log-fold change threshold to use in the null hypothesis.

```
library(iSEE)
setClass("DGETable", contains="RowTable", slots=c(LogFC="numeric"))
```

Any new slots should come with validity methods, as shown below.

```
library(S4Vectors)
setValidity2("DGETable", function(object) {
    msg <- character(0)

    if (length(val <- object[["LogFC"]])!=1L || val < 0) {
        msg <- c(msg, "'NGenes' must be a non-negative number")
    }
    if (length(msg)) {
        return(msg)
    }
```

```
    TRUE
})
```

```
## Class "DGETable" [in ".GlobalEnv"]
##
## Slots:
##
## Name:               LogFC          Selected            Search
## Class:            numeric         character         character
##
## Name:       SearchColumns           PanelId       PanelHeight
## Class:          character           integer           integer
##
## Name:          PanelWidth      SelectBoxOpen   SelectRowSource
## Class:            integer           logical         character
##
## Name:     SelectColSource        DataBoxOpen     SelectRowType
## Class:          character           logical         character
##
## Name:      SelectRowSaved      SelectColType    SelectColSaved
## Class:            integer         character           integer
##
## Name:  MultiSelectHistory
## Class:                list
##
## Extends:
## Class "RowTable", directly
## Class "Table", by class "RowTable", distance 2
## Class "Panel", by class "RowTable", distance 3
```

It is also worthwhile specializing the `initialize()` method to provide a default for new parameters. We hard-code the `SelectColType` setting as we want to obtain all multiple selections from the transmitting panel, in order to be able to perform pairwise DE analyses between the various active and saved selections. (By comparison, the default of `"Active"` will only transmit the current active selection.)

```
setMethod("initialize", "DGETable",
    function(.Object, LogFC=0, ...)
{
    callNextMethod(.Object, LogFC=LogFC, SelectColType="Union", ...)
})
```

9.3 Setting up the interface

The most basic requirement is to define some methods that describe our new panel in the `iSEE()` interface. This includes defining the full name and desired default color for display purposes:

```
setMethod(".fullName", "DGETable", function(x) "Differential expression table")

setMethod(".panelColor", "DGETable", function(x) "#AAFF00")
```

We also add interface elements to change the result type and the number of genes. This is most easily done by specializing the `.defineDataInterface` method:

```
library(shiny)
setMethod(".defineDataInterface", "DGETable", function(x, se, select_info) {
    plot_name <- .getEncodedName(x)
    list(
        numericInput(paste0(plot_name, "_LogFC"), label="Log-FC threshold",
            min=0, value=x[["LogFC"]])
    )
})
```

As we discussed before, we `paste0` the name of our panel to the name of any parameter to ensure that the ID is unique to this instance of our panel.

9.4 Creating the observers

We specialize `.createObservers` to define some observers to respond to changes in our new interface elements. Note the use of `callNextMethod()` to ensure that observers of the parent class are also created; this automatically ensures that we can respond to changes in parameters provided by `RowTable`.

```
setMethod(".createObservers", "DGETable",
    function(x, se, input, session, pObjects, rObjects)
{
    callNextMethod()

    plot_name <- .getEncodedName(x)

    num_field <- paste0(plot_name, "_LogFC")
    observeEvent(input[[num_field]], {
        previous <- pObjects$memory[[plot_name]][["LogFC"]]
        if (identical(previous, input[[num_field]])) {
            return(NULL)
        }
        pObjects$memory[[plot_name]][["LogFC"]] <- input[[num_field]]
        .requestUpdate(plot_name, rObjects)
```

```
    })
})
```

The distinction between protected and unprotected parameters is less important for `Table`s; as long as the types of the columns do not change between renderings, any column or global selections (i.e., search terms) are usually still sensible.

9.5 Making the table

When working with a `RowTable` subclass, the easiest way to change plotting content to override the `.generateTable` method. This is expected to generate a `data.frame` in the evaluation environment, returning the commands required to do so. In this case, we want to perform one-sided t-tests between the active selection and any number of saved selections. We will use the `findMarkers()` function from *scran* to compute the desired statistics. This performs all pairwise comparisons, so is not as efficient as could be, but it will suffice for this demonstration.

```
setMethod(".generateTable", "DGETable", function(x, envir) {
    empty <- "tab <- data.frame(Top=integer(0), p.value=numeric(0), FDR=numeric(0));"

    if (!exists("col_selected", envir, inherits=FALSE) ||
        length(envir$col_selected)<2L ||
        !"active" %in% names(envir$col_selected))
    {
        commands <- empty
    } else {
        commands <- c(".chosen <- unlist(col_selected);",
            ".grouping <- rep(names(col_selected), lengths(col_selected));",
            sprintf(".de.stats <- scran::findMarkers(logcounts(se)[,.chosen],
    .grouping, direction='up', lfc=%s)", x[["LogFC"]]),
            "tab <- as.data.frame(.de.stats[['active']]);"
        )
    }

    eval(parse(text=commands), envir=envir)

    list(commands=commands, contents=envir$tab)
})
```

Readers may notice that we prefix internal variables with `.` in our commands. This ensures that they do not clash with global variables created by `iSEE()` itself (which is not an issue when running the app, but makes things difficult when the code is reported for tracking purposes).

9.6 In action

Let's put our new panel to the test. We use the `sce` object, preprocessed in a previous chapter, including some precomputed dimensionality reduction results.

The plan is to create a (fixed) reduced dimension plot that will transmit to our DGE table. Setting up the iSEE instance is as easy as:

```
rdp <- RedDimPlot(PanelId=1L, SelectBoxOpen=TRUE)
dget <- new("DGETable", SelectColSource="RedDimPlot1")
app <- iSEE(sce, initial=list(rdp, dget))
```

Brushing (or lassoing) at any location and saving the selection will trigger dynamic recompution of results in our `DGETable`. We can repeat this with any number of saved selections.

Bibliography

Rue-Albrecht, K., Marini, F., Soneson, C., and Lun, A. T. L. (2018). isee: Interactive summarizedexperiment explorer. *F1000Res*, 7:741.