Making REST APIs automatically with R and Plumber

Adam Victor

November 26, 2017

Introduction

In class, we learned how to use Shiny to create web apps that allow users to manipulate variables and view R graphs that reflect the user's changes. While this works great for interactive data, Shiny apps aren't perfect for all use cases. They can require some effort to setup properly since all the UI elements need to be added and connected to backend components. Additionally, Shiny apps allow for a great user experience but aren't great for programmatic access. One of the best ways of allowing programmatic access to data is to create a REST API. A REST API is essentially a way of sending an HTTP request over the Internet and getting a result back in some usable format. REST APIs are great because they can be easily consumed from websites, mobile apps, or even Slack bots. However, like Shiny apps, REST APIs can sometimes be tedious and time consuming to setup. That's where Plumber comes in! Plumber is an extremely convenient library for setting up REST APIs for a server backend. This post will walk you through the basics of how to use Plumber to create an API by showing how to create a simple API for rating classes. Along the way, we'll also use the library httr to check what our API is sending back.

How to follow along

First, let me explain how you can follow along and get it running yourself. The most important thing to understand is that this tutorial contains both a server and a simple client. The server is the backend that runs the API; it takes requests and sends back results. The client is the code that sends the requests to the server and outputs the results for you to see. Along the way, we'll be adding code to 3 different files: server.R, surveys.R, and client.R. The code for the server itself is the file server.R. However, the server also loads the file surveys.R, which is where we will write the REST API code. These two files together, then, make up the backend server. As you probably guessed, the client code is contained in client.R. Wherever code is shown in this document, I'll make sure to note which of these 3 files it should be placed in. Make sure to pay attention to that if you're trying to follow along.

Another thing you need to keep in mind is that RStudio can only run one R process at a time. This means that if you're running the server in RStudio, you won't be able to run the client code at the same time. The way around this is to open a second RStudio program to run the other code. To reiterate, here's how to try this out yourself (once you have created the server.R, surveys.R, and client.R files). Open one RStudio program and open the server.R file. Make sure the working directory is set to the directory that contains surveys.R. Then, run the code in the server.R file, which will automatically start up a server using the API defined in surveys.R. Note: If you make changes to surveys.R, you will need to save it and then restart the server.R program. Open another RStudio program and open client.R. Then, run the code and verify that the results are what you expect!

A final reminder is that a couple packages will need to be installed. This can be done with

```
install.packages("plumber") # Version 0.4.2
install.packages("httr") # Version 1.3.1
# Note: RStudio version is 1.0.153 and R version is 3.4.1
```

Creating the server

Making the server itself is actually extremely simple and can be done in just a few lines of code! The following code should go in the server.R file.

```
# Place this code in server.R.
library(plumber)
r <- plumb("surveys.R")
r$run(port=8000)
```

Let's quickly go over what these lines of code do. First, we load the plumber library, which is what we will be using to automatically setup the REST API. Next, we set a variable to the result of calling the function "plumb" on surveys.R. Plumb is a function from the plumber library which takes an R file as input and sets up a REST API using definitions in that file, a process which will be explained shortly. Finally, we call the run function on the object we got back and specify a port to run on the server on. These few lines of code will create a server ready to handle requests and send back responses. Of course, right now it doesn't know how to handle any requests. To do that, we need to set up the API itself!

Creating the REST API

Making the REST API is more complicated than just creating a server to run it, so I'll be breaking this into several pieces and explaining each along the way. First, let's quickly go over what the API will do. We're going to create the backend for a class rating app. People can add classes that should be rated and then other users can add a rating for that class, on a scale of 1-5. Users can also get a list of every class available to rate along with its id number and can request to see a barplot of the ratings for a class with a particular id. To do all this, we will be creating REST endpoints. These are essentially URLs that accept a particular type of input and give an output based on that. Each endpoint is for a different function. For example, one endpoint will create a class and another endpoint will add a rating. REST APIs also have different "verbs" that you can use to send a request for an endpoint. This tutorial will cover only "GET" and "POST". As you might guess, GET is used to get some data from the API and POST is used to send some data to the API.

Initializing State

For our API to work, we need to keep some state around. Specifically, we need to keep all the classes and the number of each rating that class received. We'll store this data in a data frame with the columns "name", "v1", "v2", "v3", "v4", "v5". Name will be the name of the class and each v column stores the number of votes gave the class that rating. For example, v4 is the number of times a class was rated 4. For convenience, we'll also keep a vector of the rating columns to reference later. Something to keep in mind is that we're storing all of the ratings in memory. I'll explain the implications of this design choice later on.

```
# Place this code in surveys.R
# Initialize state for the API in a data frame.
surveyData <- data.frame(name = character(0),
                         v1 = integer(0),
                         v2 = integer(0),
                         v3 = integer(0),
                         v4 = integer(0),
                         v5 = integer(0),
                         stringsAsFactors = FALSE)
# Convenience vector with vote column names.
voteColumns <- c("v1", "v2", "v3", "v4", "v5")
```

Adding a Class

As mentioned before, one of the components of this application is that users will be able to add a class for others to rate. For this to work, we need to add an endpoint to our application that takes as input the name of the class to add. Additionally, since this is an example of sending data to the application it makes sense to use a POST endpoint. Here's how we can add the endpoint to surveys.R, followed by a more detailed description of what this does.

```
# Place this code in surveys.R.

# Add a class to be rated.
#* @post /surveys
addSurvey <- function(className) {
  newDataFrame <- data.frame(name = className,
                             v1 = 0, v2 = 0, v3 = 0,
                             v4 = 0, v5 = 0, stringsAsFactors = FALSE)
  surveyData <<- rbind(surveyData, newDataFrame)
  return(nrow(surveyData))
}
```

The first thing you may notice when looking at this function is the slightly strange looking comment above the function declaration. At first glance you might assume that "#*@post /surveys" is just some documentation. However, this is actually an annotation that tells Plumber to setup an endpoint! Essentially, the annotation says that if anyone tries to access the "/surveys" path with a POST request then this addSurvey function should be called with the result being shown to the caller. This is what makes Plumber so convenient - with just one comment we've made a regular R function into a useable endpoint. You might be wondering how the user can specify the name of the class to create. Again, Plumber makes things easy! POST requests can include some data with them, and in this case we'll include data with the name "className". Plumber will then automatically attach the associated data to the variable className that we have as an argument in the function. Don't worry if it's confusing how this endpoint will be called - that's part of the client side and will be explained later.

As for how the function actually works, we first create a new data frame with the same columns as surveyData and with its name set to the specified class name and all vote counts initialized to 0. We then use the rbind function to merge this data frame with the surveyData data frame, essentially adding this row to the data frame. Note that when we assign this to surveyData we use "<<-". This double arrow symbol means that it first checks if a global variable exists with that name and assigns it to that if possible. Essentially, we make sure that we assign the previously defined variable to this new data frame. If we only used "<-", we'd create a new variable that is only available in this function, which would be useless for preserving state. Finally, we return the id number of the class added, which will be sent to the client for them to use if they wish.

Getting All Class IDs and Names

The next endpoint we'll add to the API is one that takes no parameters and simply outputs a list of pairs of class id and class name. We'll use the same "/surveys" endpoint, but this time we'll make it a GET request. The reason we care about class id is because if users want to interact with the data for a specific class, they will do so using that class's id. This endpoint gives them a way to see all classes and find the id of the one they're interested in.

```
# Place this code in surveys.R.

# Returns a list of lists, where each sublist has a class's id and name.
#* @get /surveys
getSurveys <- function() {
  surveys <- list()
  for(i in 1:nrow(surveyData)) {
    surveys[[i]] <- list(i, surveyData$name[i])
  }
  return(surveys)
}
```

This function should be fairly straightforward. We use almost the same annotation as before, but instead of it having a "@post", it's now "@get". In the function, we create an empty list, and then iterate through all entries in surveyData. For each entry, we append to the list a list with the id and name of that survey. Finally, we return the list at the end. When we return a list like this, Plumber will automatically serialize it into a format that can be understood by the client.

Adding a Vote

Let's now setup an endpoint for users to add a vote for a particular rating to a class. The endpoint for this will be of the form "/surveys/id", where id will be the id number of the class. For example, to add a vote for class with id 5, a user would use the endpoint "/surveys/5". Since we're sending data, we'll make this a POST request and attach the vote as data in the request. Vote should be a number between 1 and 5 representing the user's rating.

```
# Place this code in surveys.R.

# Lets the user add a vote between 1 and 5 to the class with a specific id.
#* @post /surveys/<id:int>
addVote <- function(id, vote) {
  vote = as.numeric(vote)
  if (vote < 1 | vote > 5) {
    stop("Vote must be between 1 and 5.")
  }
  if (id < 1 | id > nrow(surveyData)) {
    stop("Survey id is not valid.")
  }
  surveyData[id, voteColumns[vote]] <<- surveyData[id, voteColumns[vote]] + 1
  return(paste0("Added a vote for ", vote, " to ", surveyData$name[id]))
}
```

The endpoint setup looks pretty similar to before, except that we now have an extra "/". This allows us to setup a dynamic endpoint. For example, this endpoint will be used for any url of the form "/surveys/id" where id is some integer. Additionally, the id in the path will be bound to the argument "id" of the function! We also include an argument for the vote itself, which will be passed as part of the POST data. The code itself is pretty straight forward. We make sure the vote and id are both valid and then increment the vote count for that vote column in the surveyData table. Then, we return a simple message to the user detailing what was done.

Getting a Vote Barplot

The final endpoint will allow the user to view a barplot of the vote distribution for a particular class. We'll again use the "/surveys/id" endpoint, but this time with a GET request.

```
# Place this code in surveys.R.

# Gets a barplot with the vote distribution for a class with a specific id.
#* @get /surveys/<id:int>
#* @png
getVotePlot <- function(id) {
  columns = surveyData[id, voteColumns]
  barplotData = c(columns[[1]], columns[[2]],
                  columns[[3]], columns[[4]], columns[[5]])
  barplot(barplotData, names.arg = c("1", "2", "3", "4", "5"),
          xlab = "Rating", ylab = "Number of Votes")
  title(paste0("Vote distribution for ", surveyData[id, "name"]))
}
```

Let's start by looking at the annotation we used - it's almost identical to the last except that we use "@" instead of "@". Another key difference is the additional annotation below it, "#* @png". This denotes that whatever the function returns should be serialized into a png image. In effect, it will take whatever graph we make and send it back to the client as a png! The function itself takes an id as an argument and retrieves the vote columns for that class. It then creates a vector with the number of votes for each rating and finally constructs a labeled barplot for that class. Plumber handles the hard part by making that barplot into a png image.

And with that, the backend REST API is complete!

Creating a Simple Test Client

Now that the backend API has been completed, let's make a simple client that will make requests to the API and use the responses. We'll make use of the "httr" library which provides functions for making GET and POST requests to URLs. All the URLs we're making requests to start with "localhost:8000" because localhost refers to the machine the client is being run on (which will be the same machine you're running the server on) and in server.R we specified that we'd use port 8000.

```
#Place this code in client.R.

library("httr")
```

```
## Warning: package 'httr' was built under R version 3.4.2
```

```
set.seed(1)
# Try adding classes
classnames = c("Math 101", "History 102", "Advanced Psychology", "CS61B")
# Add all these classes to the backend.
for (class in classnames) {
  id = content(
    POST(url = "localhost:8000/surveys",
         body = paste0("className=", class))
  )
  print(paste0("Added ", class, " with id = ", id))
}
```

```
## [1] "Added Math 101 with id = 1"
## [1] "Added History 102 with id = 2"
## [1] "Added Advanced Psychology with id = 3"
## [1] "Added CS61B with id = 4"
```

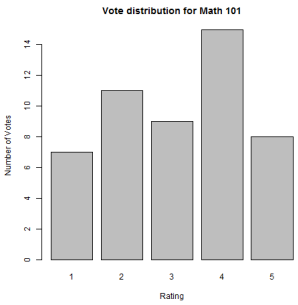
The above code simply defines a vector of classnames and then iterates through each one. For every class in the vector, we make a POST

request to the server's "/surveys" endpoint. Additionally, we include data with the post request - the name of the class. This data will then be available for the backend to use. Finally, for each one we add we get back its id and print it out.

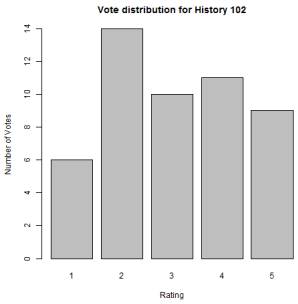
```
# Get a list of all class ids and names.
allClasses = content(GET(url = "localhost:8000/surveys"), as = "parsed")
for (classData in allClasses) {
  # classData will be a list with class id and name.
  # We use 2 brackets because id and name are stored as a list within the list.
  id = classData[[1]][[1]]
  name = classData[[2]][[1]]
  # For every class generate a random sample of votes and post them.
  if (name == "CS61B") {
    randomSample = sample(x = 1:5, prob = c(.03, .05, .3, .35, .27), replace = TRUE, size = 50)
  } else {
    randomSample = sample(x = 1:5, replace = TRUE, size = 50)
  }
  for (vote in randomSample) {
    content(
    POST(url = paste0("localhost:8000/surveys/", id),
         body = paste0("vote=", vote))
    )
  }
  # Download the graphs for each class.
  bin <- content(
    GET(paste0("localhost:8000/surveys/", id)),
    "raw")
  filename = paste0("class", id, "_plot.png")
  writeBin(bin, filename)
}
```

In the above code, we send a GET request to the "/surveys" endpoint. This gives us a list of lists, where each inner list contains a class id and class name. However, by default Plumber will wrap each element of the inner list in another list. For example, the overall list might look like [[[1], ["Math 101"]], [2], ["History 102"]]. For each class, we generate a random sample of votes and POST the votes to the "/surveys/id" endpoint. Just to test that we're getting back class names, we check if the class is "CS61B" and if we do generate samples that are more likely to have higher votes (what can I say, I just really liked 61B). After sending all the vote data, we send a GET request to the "/surveys/id" endpoint to get a graph of the votes in binary form. We then write the binary data to a png file. Finally we use knitr to display all the images we downloaded.

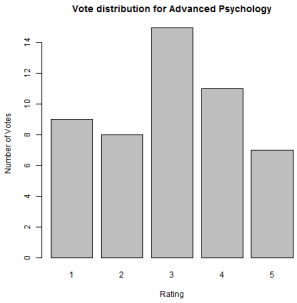
```
# Display all the graphs.
knitr::include_graphics("class1_plot.png")
```



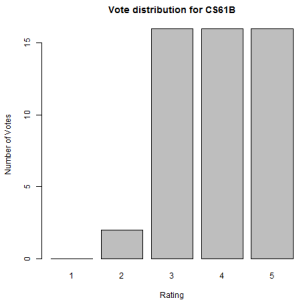

```
knitr::include_graphics("class2_plot.png")
```




```
knitr::include_graphics("class3_plot.png")
```




```
knitr::include_graphics("class4_plot.png")
```



Future Considerations

While this post covered the gist of making APIs with Plumber, there were a lot of important details that were skipped over. For example, our APIs are completely insecure and potentially could be abused. For more info on security considerations, take a look at the Plumber docs. Another problem is that R is single threaded and does not allow for multiple concurrent requests to be made to the server. Therefore, scaling up the backend may require multiple servers to be running. Another problem is how we're storing the voting data. Currently, we store it entirely in memory. However, if we restarted the server all the data would be lost. Also, if we end up deploying multiple servers they won't all have access to the data. Ideally, some sort of database should probably be used to access and store the data.

Conclusion

And there you have it, we've created a simple REST API ready to be used for websites, mobile apps, etc. The magic of Plumber is in its simplicity. With just a few annotations, we can automatically wire together the whole API just by using simple R functions. Additionally, Plumber's automatic serialization of graphs makes it really easy to get graphical data from a Plumber API. This could be a great way of adding statistics onto an already existing application in a way that makes them easy to view and analyze. R is already a powerful language for statistics, but by building a REST API on top of it we make it even more powerful by enabling even more people to use the data without having direct access to it.

References

- https://www.rplumber.io/
- https://www.knowru.com/blog/how-create-restful-api-for-machine-learning-credit-model-in-r/
- https://www.slideshare.net/sellorm/creating-apis-with-r-and-plumber-57608851
- https://www.computerworld.com/article/3171766/application-development/how-to-create-your-own-slack-bots-and-web-apis-in-r.html?page=2
- https://trestletech.com/2015/06/rapier-convert-r-code-to-a-web-api/
- http://httr.r-lib.org/articles/quickstart.html
- https://awesome-r.com/