We will create an API built on top of simulated customer call data that powers a slash command. This command will allows users to view a customer status report within Slack. As shown, this status report contains customer name, total calls, date of birth, and a plot of call history for the past 20 weeks.

**Setup**

[Slack](https://slack.com/) is a commonly used communication tool that's highly customizable through various integrations. It's even possible to build your own integrations, which is what we'll be doing here. In order to build a Slack app, you need to have a Slack account. In this example, we will build an app that includes a slash command that users can access by typing / into a Slack message.

**The Slack request**

In this scenario, we're building an API that will interact with a known request. This means that we need to understand the nature of the incoming request so that we can appropriately handle it within the API. Slack provides [some documentation](https://api.slack.com/slash-commands#app_command_handling) about the request that is sent when a slash command is invoked. In short, an HTTP POST request is made that contains a URL-encoded data payload. An example data payload looks like:

token=gIkuvaNzQIHg97ATvDxqgjtO

&team\_id=T0001

&team\_domain=example

&enterprise\_id=E0001

&enterprise\_name=Globular%20Construct%20Inc

&channel\_id=C2147483705

&channel\_name=test

&user\_id=U2147483697

&user\_name=Steve

&command=/weather

&text=94070

&response\_url=https://hooks.slack.com/commands/1234/5678

&trigger\_id=13345224609.738474920.8088930838d88f008e0

There's a lot of detail included in the Slack request, and the [Slack documentation](https://api.slack.com/slash-commands#app_command_handling) provides details about each field. We're mainly interested in the text field, which contains the text entered into Slack after the slash command. In the above example, the user entered /weather 94070 into Slack, so the request indicates that the command was /weather and the text was 94070.

Note that this approach is different from APIs that are not being built around a known request or specification. In such instances, we are free to expose endpoints and return data in whatever method seems most beneficial to downstream consumers. In such a scenario, we would provide downstream API consumers with an understanding of how the API handles requests and what types of responses are generated so that they can appropriately interact with the API. But in this example, the design of our API is, in part, dictated by the specifications Slack provides.

**Building the API**

Now that we have an understanding of what is included in the incoming request, we can begin to build out the API using plumber. First, we need to set up the global environment for the API by loading necessary packages and global objects, including the simulated data this API is built on. In reality, this data would likely come from an external database accessed via an ODBC connection.

# Packages ----

library(plumber)

library(magrittr)

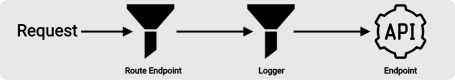
library(ggplot2)

# Data ----

# Load sample customer data

sim\_data <- readr::read\_rds("data/sim-data.rds")

The following diagram outlines what we want to build.



In essence, an incoming request will pass through two filters before reaching an endpoint. This first filter is responsible for routing incoming requests to the correct endpoint. This is done so that a single slash command can serve multiple endpoints without the need to create separate commands for each service. The second filter simply logs details about the request for future review.

**Filters**

The first filter is responsible for parsing the incoming request and ensuring it is assigned to the appropriate endpoint. This is done because when a slash command is created in Slack, there is only one endpoint defined for requests made from the command. This filter enables several endpoints to be utilized by the same slash command by parsing the incoming text of the command and treating the first value of that command as the endpoint to which the request should be routed.

#\* Parse the incoming request and route it to the appropriate endpoint

#\* @filter route-endpoint

function(req, text = "") {

# Identify endpoint

split\_text <- urltools::url\_decode(text) %>%

strsplit(" ") %>%

unlist()

if (length(split\_text) >= 1) {

endpoint <- split\_text[[1]]

# Modify request with updated endpoint

req$PATH\_INFO <- paste0("/", endpoint)

# Modify request with remaining commands from text

req$ARGS <- split\_text[-1] %>%

paste0(collapse = " ")

}

# Forward request

forward()

}

This filter requires an understanding of the req object. It's important to note that a few things happen in this filter. First, we parse the text argument and use the first part of text as the req$PATH\_INFO, which tells plumber where to route the request. Second, we take anything remaining from text and attach it to the request in req$ARGS. This means that any downstream filters or endpoints will have access to req$ARGS.

#\* Log information about the incoming request

#\* @filter logger

function(req){

cat(as.character(Sys.time()), "-",

req$REQUEST\_METHOD, req$PATH\_INFO, "-",

req$HTTP\_USER\_AGENT, "@", req$REMOTE\_ADDR, "\n")

# Forward request

forward()

}

**Endpoints**

There are a few endpoints we need to define. First, we need to define an endpoint that provides a response Slack can understand and interpret into a message. In this case, we're going to return a [JSON object](https://www.json.org/) that Slack interprets into a message with attachments. Slack provides [detailed documentation](https://api.slack.com/docs/message-attachments) on what fields it accepts in a response. Also note that Slack expects unboxed JSON, while the plumber default is to return boxed JSON. In order to ensure that Slack understands the response, we set the serializer for this response to be unboxedJSON.

#\* Return a message containing status details about the customer

#\* @serializer unboxedJSON

#\* @post /status

function(req, res) {

# Check req$ARGS and match to customer - if no customer match is found, return

# an error

customer\_ids <- unique(sim\_data$id)

customer\_names <- unique(sim\_data$name)

if (!as.numeric(req$ARGS) %in% customer\_ids & !req$ARGS %in% customer\_names) {

res$status <- 400

return(

list(

response\_type = "ephemeral",

text = paste("Error: No customer found matching", req$ARGS)

)

)

}

# Filter data to customer data based on provided id / name

if (as.numeric(req$ARGS) %in% customer\_ids) {

customer\_id <- as.numeric(req$ARGS)

customer\_data <- dplyr::filter(sim\_data, id == customer\_id)

customer\_name <- unique(customer\_data$name)

} else {

customer\_name <- req$ARGS

customer\_data <- dplyr::filter(sim\_data, name == customer\_name)

customer\_id <- unique(customer\_data$id)

}

# Simple heuristics for customer status

total\_customer\_calls <- sum(customer\_data$calls)

customer\_status <- dplyr::case\_when(total\_customer\_calls > 250 ~ "danger",

total\_customer\_calls > 130 ~ "warning",

TRUE ~ "good")

# Build response

list(

# response type - ephemeral indicates the response will only be seen by the

# user who invoked the slash command as opposed to the entire channel

response\_type = "ephemeral",

# attachments is expected to be an array, hence the list within a list

attachments = list(

list(

color = customer\_status,

title = paste0("Status update for ", customer\_name, " (", customer\_id, ")"),

fallback = paste0("Status update for ", customer\_name, " (", customer\_id, ")"),

# History plot

image\_url = paste0("localhost:5762/plot/history/", customer\_id),

# Fields provide a way of communicating semi-tabular data in Slack

fields = list(

list(

title = "Total Calls",

value = sum(customer\_data$calls),

short = TRUE

),

list(

title = "DoB",

value = unique(customer\_data$dob),

short = TRUE

)

)

)

)

)

}

There are three main things that happen in this endpoint. First, we check to ensure that the provided customer name or ID appear in the dataset. Next, we create a subset of the data for only the identified customer and use a simple heuristic to determine the customer's status. Finally, we put a list together that will be serialized into JSON in response to requests made to this endpoint. This list conforms to the standards outlined by Slack.

The second endpoint is used to provide the history plot that is referenced in the first endpoint. When an image\_url is provided in a Slack attachment, Slack uses a GET request to fetch the image from the URL. So, this endpoint responds to incoming GET requests with an image.

#\* Plot customer weekly calls

#\* @png

#\* @param cust\_id ID of the customer

#\* @get /plot/history/

function(cust\_id, res) {

# Throw error if cust\_id doesn't exist in data

if (!cust\_id %in% sim\_data$id) {

res$status <- 400

stop("Customer id" , cust\_id, " not found.")

}

# Filter data to customer id provided

plot\_data <- dplyr::filter(sim\_data, id == cust\_id)

# Customer name (id)

customer\_name <- paste0(unique(plot\_data$name), " (", unique(plot\_data$id), ")")

# Create plot

history\_plot <- plot\_data %>%

ggplot(aes(x = time, y = calls, col = calls)) +

ggalt::geom\_lollipop(show.legend = FALSE) +

theme\_light() +

labs(

title = paste("Weekly calls for", customer\_name),

x = "Week",

y = "Calls"

)

# print() is necessary to render plot properly

print(history\_plot)

}

Once these pieces are together, you can run the API either through the UI as described in the previous post, or by running plumber::plumb("plumber.R")$run(port = 5762) from the directory containing the API.

**Testing the API**

Once the API is up and running, we can test it to make sure it's behaving as we expect. Since the main point of contact is making a POST request to the /status endpoint, it's easiest to interact with the API through curl.

$ curl -X POST --data '{"text":"status 1"}' localhost:5762 | jq '.'

{

"response\_type": "ephemeral",

"attachments": [

{

"color": "good",

"title": "Status update for Rahul Wilderman IV (001)",

"fallback": "Status update for Rahul Wilderman IV (001)",

"image\_url": "localhost:5762/plot/history/001",

"fields": [

{

"title": "Total Calls",

"value": 27,

"short": true

},

{

"title": "DoB",

"value": "2004-04-01",

"short": true

}

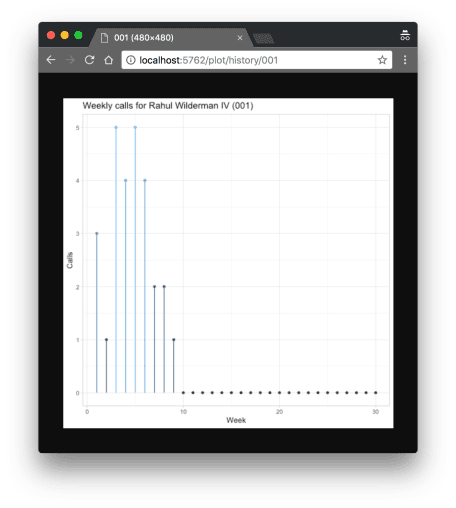
]

}

]

}

Success! Our API successfully routed our request to the appropriate endpoint and returned a valid JSON response. As a final check, we can visit the image\_url in our browser to see if the plot is properly rendered.



Everything appears to be running as expected!

**Conclusion**

In this post, we used plumber to create an API that can properly interact with the Slack slash command interface. In the next post, we will explore API security and deployment. Continuing with this example, we will secure our API using [Slack's guidelines](https://api.slack.com/docs/verifying-requests-from-slack), deploy the API, and finally connect Slack so that we can use our new slash command. At the conclusion of the next post, we will have a fully functioning Slack slash command, all built using R.

*James Blair is a solutions engineer at RStudio who focusses on tools, technologies, and best practices for using R in the enterprise.*