Building APIs with R

Lecture 14

Dr. Colin Rundel


plumber
rplumber.io

plumber

Plumber allows you to create a web API by merely decorating your existing R source code with roxygen2-like comments. Take a look at an example.

```
# plumber.R

#* Echo back the input
#* @param msg The message to echo
#* @get /echo
function(msg="") {
  list(msg = paste0("The message is: '", msg, "'"))
}

#* Plot a histogram
#* @serializer png
#* @get /plot
function() {
  rand <- rnorm(100)
  hist(rand)
}
```

These comments allow plumber to make your R functions available as API endpoints.

Running your API

Your plumber API is just an R script, so you can run it just like you would any other R script (but this won't do much) since we need both the code and the metadata / decorators.

If you are using RStudio it will likely identify the script as a plumber API and provide a "Run API" button in the script editor.

However, if you are using something else or would like more control you have two options:

```
# Use an R6 object
p = plumb("Lec14_ex1.R")
p$run()
```

```
# Use the pipeable interfac
pr("Lec14_ex1.R") |>
  pr_run()
```

Decorators / Annotations

Plumber makes use of comment based decorators to determine the structure and behavior of the API.

There are a number of other packages that use a similar approach, including `roxygen2`, `Rcpp`, and others that we will be seeing later in the class.

Some useful details:

- Decorators preceed the function (endpoint) definition
- Either `#'` or `#*` can be used but the latter is preferred
- `@cmd` based annotations are used to specify the behavior of an endpoint

Endpoints

Endpoints are therefore just decorated anonymous R functions (the name/endpoint is determined by the decorator).

Every endpoint must have (at least) one of the below annotations followed by a path.

- `@get`
- `@post`
- `@put`
- `@delete`
- `@head`

these determine the HTTP method that the endpoint will respond to.

Multiple verbs

Endpoints can have a single verb or multiple verbs, in the latter case your code will need to handle the different verbs. Similarly, the same endpoint may be created multiple times with different decorator verbs.

The two examples below are equivalent in the endpoints created, but there may be advantages with either approach depending on the use case.

```
#* @get /cars
#* @post /cars
#* @put /cars
function(){
  ...
}
```

```
#* @get /cars
function(){
  ...
}

#* @post /cars
function(){
  ...
}

#* @put /cars
function(){
  ...
}
```

Query parameters

A plumber function's arguments determine the query parameters that the endpoint will accept.

Including the `@param` annotation is optional, but allows for more detailed documentation of the parameters (and shows up in the autogenerated Swagger page).

```
#* Echo back the input
#* @param msg The message to echo
#* @get /echo
function(msg="") {
  paste0("The message is: '", msg, "'")
}
```

```
#* Return the sum of two numbers
#* @param a The first number to add
#* @param b The second number to add
#* @post /sum
function(a, b) {
  as.numeric(a) + as.numeric(b)
}
```

Dynamic routes

Plumber also supports dynamic routes, which are specified by including a parameter in the path rather than as part of a query string. This is a common feature in many REST APIs (e.g. GitHub's).

One or more parameters are specified in the path by wraping the parameter name with <>.

```
#* @get /msg/<from>/<to>
function(from, to, msg="Hello!){
  paste0("From: ", from, ", to: ", to, " - ", msg)
}
```

Typed dynamic routes

By default plumber treats all parameters as strings (`character`), and your function is responsible for any type coercion necessary.

In the case of parameters from dynamic routing - the type can be specified in the path using `:` followed by `bool`, `logical`, `double`, `numeric`, or `int`.

```
#* @get /user/<id:int>
function(id){
  next <- id + 1
  # ...
}
```

```
#* @post /user/activated/<activ
function(active){
  if (!active){
    # ...
  }
}
```

Serialization

By default, the return value of a plumber endpoint is serialized to JSON (via `jsonlite`). This can be overridden by using the `@serializer` annotation.

Some of the available serializers,

Annotation	Content Type	Description/References
`@serializer html`	`text/html; charset=UTF-8`	Passes response through without
`@serializer json`	`application/json`	Object processed with `jsonlite::toJSON()`
`@serializer csv`	`text/csv`	Object processed with `readr::format_csv()`
`@serializer text`	`text/plain`	Text output processed by `as.character()`
`@serializer print`	`text/plain`	Text output captured from `print()`
`@serializer cat`	`text/plain`	Text output captured from `cat()`
`@serializer jpeg`	`image/jpeg`	Images created with `jpeg()`
`@serializer png`	`image/png`	Images created with `png()`
`@serializer svg`	`image/svg`	Images created with `svg()`
`@serializer pdf`	`application/pdf`	PDF File created with `pdf()`

`req` & `res`

Plumber endpoint functions can also optionally have `req` and `res` arguments which capture the HTTP request and response objects respectively.

These objects are R `environment` objects and are useful for more advanced use cases, such as setting custom headers, cookies, or status codes.

```
#* @get /req
function(req, arg="") {
  if (arg == "")
    ls(envir=req)
  else
    req[[arg]]
}
```

```
#* @get /res
function(res, arg="") {
  if (arg == "")
    ls(envir=res)
  else
    res[[arg]]
}
```

Named parameter collisions

There are three distinct ways that arguments / values can be passed to an endpoint function:

1. Query string parameters (e.g. `?a=1&b=2`)
2. Dynamic path parameters (e.g. `/user/<user_id>/`)
3. Body parameters (e.g. `POST` or `PUT` requests)

Only the first two are directly accessible as arguments to the endpoint function. The third is accessible via the `req` object. It is possible for these to collide, and Plumber has a specific order of precedence for how these are resolved.

All three are accessible within the `req` environment using `req$argsPath`, `req$argsBody`, and `req$argsQuery`. There is also `req$args` which is a list of all three combined.

Errors

If any of your plumber code throws an error, the error will be caught and returned as a JSON object with the error message and a `500` HTTP status code.

More specific HTTP errors can be returned by modifying the `res$status` and `res$body` fields.

```
#* @get /error
#* @serializer html
function() {
  stop("This is an error")
}
```

```
#* @get /forbidden
#* @serializer text
function(res) {
  res$status = 403
  res$body = "You are forbi
}
```

More specific modification of error responses (500, 404, etc) can be set using `pr_set_error()`, `pr_set_404()` and

State

For some apps it is useful to have a shared state between endpoints. This can be achieved in a number of different ways - from the use of global variables, to file-based storage, to databases.

A simple example of the former,

```
# plumber.R

clicks = 0

#* @get /increment
function() {
  clicks <<- clicks + 1
  list(clicks = clicks)
}

#* @get /current
function() {
  list(clicks = clicks)
}
```

Live Demo