



Introduction to Plumber APIs.



WORKSHOP 2.

Prerequisites.

Required

- Download workshop repo:
 - <https://github.com/MangoTheCat/plumber.workshop>
- R (programming language) and Rtools (Windows):
 - <https://cran.r-project.org/bin/windows/base/>
 - <https://cran.r-project.org/bin/windows/Rtools/>
- Rstudio (R IDE: free version):
 - <https://www.rstudio.com/products/rstudio/download/>
- R Packages:
 - **plumber (1.1.0)**
 - jsonlite
 - httr
 - CVrisk
 - readr
 - devtools
 - remotes



plumber

Optional (advanced)

Postman (application for sending HTTP requests):

<https://www.postman.com/downloads/>



Docker (application for building and running containers):

<https://www.docker.com/get-started/>



Schedule.

Section	Description	Time
Background	What are APIs and what is plumber?	10:00 – 10:30am
Introductory exercises	An easy API	10:30-11:20am
	Running the package API	
	Creating simple endpoints	
	Using serializers	
	Sending GET requests	
Break		11:20-11:40am
Intermediate exercises	Sending POST requests	11:40-12:30pm
	Using global data	
	Plumber request and response objects	
	Authorization filter	
Advanced exercises	Dockerizing API	12:30-13:00pm
	Asynchronous endpoints	

Background.

What is an API?

- APIs provide an *interface* for two *applications* to communicate, and share data and functionality *programmatically*
- Every website is an API, but not all APIs are websites, for example:
 - Google Maps API
 - Python Tensorflow API
- API is a general term, here we are talking about:

*Making R functions available to other applications via a *URI* that can accept *HTTP* requests*

Definitions:

HTTP = Protocol for sending messages in a structured format

API = Application programming interface

GET = get some data (e.g. websites)

POST = post some data, get some data back (e.g. ML models)

What is plumber?

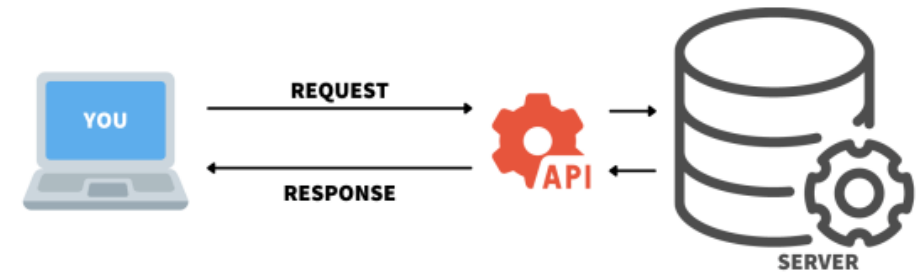
- Plumber allows you to create a web API by simply decorating R code with roxygen2-style annotations.
- Example use-cases:
 - Serve statistical models
 - Serve machine learning models as endpoints
 - Integrate R visualizations into other applications
 - Share data

POST Request

- Header: key-value pairs (auth-key=[password])
- **Body: contains data**
- (Some other ~default info like source of request) {

"height":180
"weight":67

}



{

"bmi":20.6

}

Response

- HTTP status code (200: Success, 404: Not found)
- Header: key-value pairs (content-type=JSON)
- **Body: contains data**

Example plumber endpoint.

These are **block annotations** and are used to specify the attributes of an endpoint, filter or static file handler in plumber. They start with **#***

```
1  /* CVD risk calculator
2  /* Returns risk of CVD event in the next 10-years
3  /*
4  /* @param gender:str male or female
5  /* @param age:int age in years
6  /* @param bmi:dbl body-mass index
7  /* @param sbp:dbl systolic blood pressure
8  /* @param bp_med:int on blood pressure medication 1=Yes, 0=No
9  /* @param smoker:int Are you a smoker 1=Yes, 0=No
10 /* @param diabetes:int Do you have diabetes 1=Yes, 0=No
11 /*
12 /* @post /heart_disease_risk
13 /* @serializer unboxedJSON
14 cvd_event_risk|
```

Block annotations for **endpoints** are always followed by a function. If the function is available, the name is sufficient, however, the function can be specified here anonymously →

```
/* @get
\O{
  "Hello world!"
}
```

(1) **Summary field** The first comment without an **@annotation**

(2) **Description field** Additional comments without **@annotations**

(4-10) **Parameters** Endpoint parameter names, data-types and descriptions

(12) **HTTP method and path**
Annotation used to generate an endpoint (e.g: **@get** and **@post**) followed by the path


(13) **Serializer** Output/return type

(14) **Endpoint function** This is the function which is called from a request to the endpoint

Introductory exercises.

0 – An easy API.

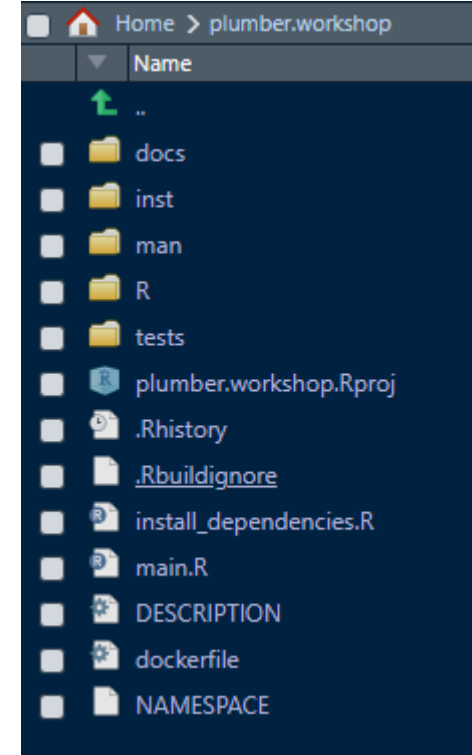
- ❑ Open Rstudio
- ❑ Create a file called **plumber.R**
- ❑ Enter the following code and **SAVE**:

```
#* @get /hello  
function() "Hello world!"
```
- ❑ In the top right corner of the script pane – click Run API 
- ❑ Voila, your first API!
- ❑ In the swagger UI that pops up, click **Try it out** and then **Execute**.
- ❑ You can now **delete** this file! As we will be using another plumber.R file for the workshop

1 – Setting up the project.

In the repo there is an API that is setup like an **R package** in a similar way **golem** is used for shiny apps.

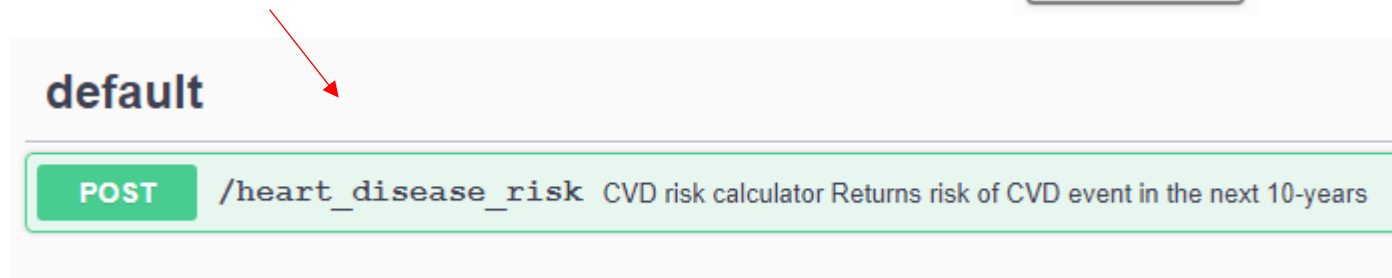
In the first exercise (next slide) we will run and interact with the existing API in this package.



- **docs**: documents for this workshop
- **inst**: package file, contains API specification files
- **man**: help for the R/ functions
- **R**: R functions
- **tests**: tests for scripts in R/

1 – Instructions.

- ❑ Download workshop repo: <https://github.com/adamwaring/plumber.workshop>
- ❑ Open the project in Rstudio
- ❑ Run the code `devtools::install()` then `devtools::load_all()`
- ❑ Install dependencies `./install_dependencies.R`
- ❑ Open the file `./inst/api/plumber.R`
- ❑ In the top right corner of the script pane – click Run API 
- ❑ A new window should open with the [swagger interface](#) to the API. Click on the endpoint `/heart_disease_risk` and click Try it out 



2–Creating Endpoints using block annotations.

- Plumber allows you to specify endpoints using header comments called **block annotations**.
- Plumber annotations start with **#*** and can be followed by **@[keyword]**
- In the plumber.R file from the previous exercise, annotations are already used:
 - **global annotations** **#* @apiTitle** and **#* @apiDescription**
 - **block annotations** preceding the filter and the endpoint.
- In exercise 2, let's create some more endpoints!

See here for all available annotations

<https://www.rplumber.io/articles/annotations.html>.

2–Instructions.

Endpoint 1

- ❑ Summary field: Check the API is working
- ❑ Method and path: *#{@get /ok}*
- ❑ Endpoint function: `function() "OK"`

Endpoint 2

- ❑ Summary field: Greetings {name}
- ❑ Method and path: *#{@get /greetings}*
- ❑ Param: *#{@param name:string User name}*
- ❑ Endpoint function: `function(name) paste0("Hello ", name, "!!")`

Check your new endpoints in the auto-generated Swagger interface using the Run API button at the top of the script!

3 – Serializers.

- **Serializers** specify the response body format
- The default is **JSON** but many more are available including tables (csv, tsv) and images (png)
- See here for all available serializers
<https://www.rplumber.io/reference/serializers.html>
- In exercise 3, let's create an endpoint with a non-default serializer!

3 – Instructions (choose 1 to implement).

Endpoint 1

- ❑ **Summary field:** Returns a random normal histogram
- ❑ **Method and path:** *## @get /histogram*
- ❑ **Serializer:** *## @serializer png*
- ❑ **Endpoint function:** `function()` `hist(rnorm(1000))`

Endpoint 2

- ❑ **Summary field:** Returns the sum of two numbers
- ❑ **Method and path:** *## @get /add*
- ❑ **Param 1:** *## @param x:int number 1*
- ❑ **Param 2:** *## @param y:int number 2*
- ❑ **Serializer:** *## @serializer unboxedJSON*
- ❑ **Endpoint function:** `function(x, y)` `as.numeric(x) + as.numeric(y)`

Endpoint 3

- ❑ **Summary field:** Returns the Iris dataset in tsv format
- ❑ **Method and path:** *## @get /iris*
- ❑ **Serializer:** *## @serializer tsv*
- ❑ **Endpoint function:** `function()` `iris`

NOTE: Swagger sends arguments as strings even if you specify *## @param x:int* use `as.numeric()` to convert them

4 – Sending HTTP requests (GET).

- So far we used the swagger API to send HTTP requests
- There are many ways to send these HTTP requests, including any common programming languages
- In R it can be with the httr package
- In the next exercise, we will see how to send GET requests from a web browser, from within R and from the command line

4 – Instructions.

Firstly, let's run the API with a specific port.

- ❑ Instead of using the **Run API** button – open the file called `./main.R`
- ❑ In the top-right of the script panel source the script as a local job (this runs the API in another session)

1. Web browser

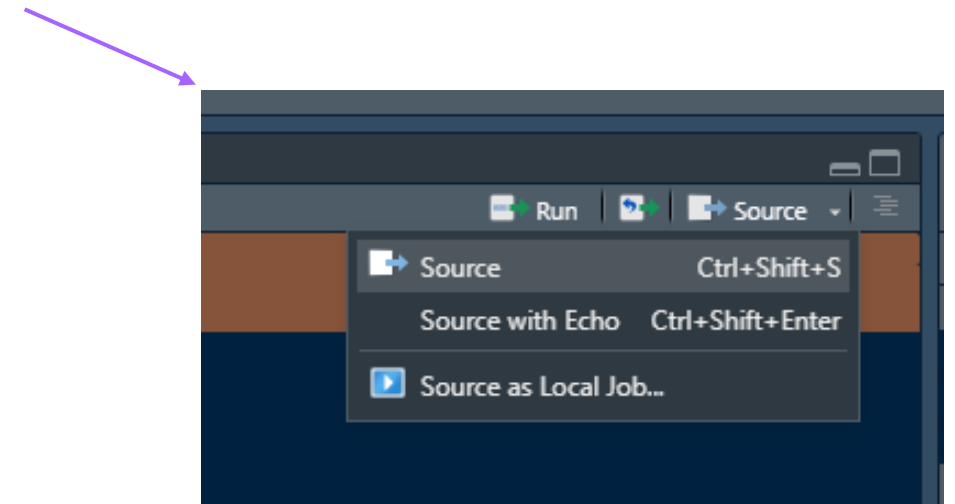
- ❑ <http://localhost:80/greetings?name=Gandalf>
- ❑ <http://localhost:80/histogram>

2. Within R (httr)

- ❑ Open the file `./tests/GET_requests_with_httr.R`
- ❑ Run the code for `/greetings` and `/add`
- ❑ Test the `/iris` endpoint yourself

3. Command prompt (curl)

- ❑ Open a command prompt, and type:
- ❑ `curl "http://localhost/add?x=1&y=2"`



Intermediate exercises.


1 – Sending HTTP requests (POST).

- Sending GET requests is easy as we can send the parameters as **query arguments**
- For POST requests we usually want to send a more structured request with a **body** containing data
- As well as a **body** you can send **header** attributes as key-value pairs which will come in handy later (auth)
- The next exercise demonstrates how to send POST requests with **curl**, **httr** and **Postman** <https://www.postman.com/downloads/>

1 – Instructions.

- ❑ If not still running, run the API with a specific port as a local job.

Can also do it a cleaner way with a JSON file containing the data



1. Command prompt (curl)

- ❑ Open a command prompt, and type:
- ❑ `curl -X POST "http://localhost:80/heart_disease_risk" -H "Content-Type: application/json" -d "{\"sex\": \"male\", \"age\": 50, \"bmi\": 24, \"sbp\": 125, \"bp_med\": 0, \"smoker\": 0, \"diabetes\": 1}"`

2. Within R (httr)

- ❑ Open the file `./tests/POST_requests_with_httr.R`
- ❑ Run the code for the `/heart_disease_risk` endpoint

2–Global data.

- Any data declared at the top of the `./plumber.R` script is global data that can be accessed by subsequent filters and endpoints
- This is useful in terms of performance as it is loaded once to serve many HTTP requests
- In this exercise, we will create an endpoint that returns expected lifespan (UK) given age and sex based on a table of data stored in `./inst/extdata/lifespan_data_UK.tsv`

2–Instructions.

Global data

Reading the file this way works for both local development and after installation of the package in a deployment environment

```
lifespan_path = system.file("extdata", "lifespan_data_UK.tsv", package = "plumber.workshop")
lifespan = read.table(lifespan_path, sep="\t", h=T)
```

Endpoint

- ❑ **Summary field:** Returns average lifespan (UK)
- ❑ **Method and path:** *## @post /average_lifespan*
- ❑ **Param 1:** *## @param age:int User age 0-100*
- ❑ **Param 2:** *## @param sex:string User sex male or female*
- ❑ **Serializer:** *## @serializer unboxedJSON*
- ❑ **Endpoint function:** `function(age, sex){
 plumber.workshop::average_lifespan(age, sex, lifespan)
}`

This is not a parameter for the endpoint, it is taken from the global environment

3 – The request and response objects.

- Plumber conveniently extracts the parameters in the **request object** and matches them to the endpoint functions
- However there are situations where you might want to directly access the request object and do the extraction yourself – for example filters, or when the number of parameters becomes unmanageable
- Both the request and response objects are **environments**
- The next exercise will demonstrate how to interact with these objects

3 – Instructions.

- ❑ Create and run the following endpoint

Endpoint

- ❑ **Summary field:** Debugging endpoint
- ❑ **Method:** *#{@post /browser}*
- ❑ **Endpoint function:** `function(req, res) browser()`

- ❑ Open and command line, and type:

```
curl -X POST "http://localhost:80/browser" -H "Content-Type: application/json" -d "{\"x\":1, \"y\":2}"
```

- ❑ Once the `browser()` breakpoint is activated you will be directed to the console, explore the object `req` (request) and `res` (response):

```
mode(req)
names(req)
as.list(req)
req$args
req$postBody
mode(res)
names(res)
res$status
```


4 – Filters.

- Now we are familiar with the request and response objects we will look at **filters**
- Filters modify the incoming request before it reaches the endpoints
- A **logging** filter has already been created, inspect its code in R/
- In the next exercise we will implement an **authorization** filter

4 – Instructions.

Filter

- ❑ Filter name: *.* auth*
- ❑ Filter function: `plumber.workshop::authorizer`

Inspect the
function in
the .R/ folder



- ❑ Create an environment variable with the authorization password
`Sys.setenv(plumber_auth_key="test123")` – this is not persistent across sessions
- ❑ Any requests without the authorization header will now fail – use *.* @preempt auth* in the block annotations of select endpoints to skip authorization
- ❑ To make a request with the authorization key use http, curl or Postman - See next slide for more detail

4 – More detail.

□ httptr

- Open the file
./tests/POST_requests_with_httptr.R
- Create the AUTHORIZATION header with the following line:

```
headers = httptr::add_headers(AUTHORIZATION = "test123")
```

- In the POST request add the argument:
config=headers

□ curl

- Open a command prompt, and type:

```
curl -X POST "http://localhost:80/heart_disease_risk" -H  
"Content-Type: application/json" -H  
"Authorization:test123" -d '{"sex": "male", "age": 50,  
"bmi": 24, "sbp": 125, "bp_med": 0, "smoker": 0,  
"diabetes": 1}'
```

Sidenote: Authentication and Swagger

When running the API using RunAPI – you will now get an authentication error because swagger sends requests (without headers) to build the interface.

To avoid this you can drop the swagger or use a separate helper function to skip authentication for swagger requests:

```
is_swagger_request = function(path_info){  
  
  swagger_urls = c("/__docs__/", "/__swagger__/",  
"/openapi.json")  
  
  any(sapply(swagger_urls, function(x) grepl(x,  
path_info)))  
  
}  
**path_info = req$PATH_INFO
```

However, you cannot use endpoints with the swagger interface as you cannot provide the key, to allow certain endpoints to skip this authentication process, use annotation: **##**
@preempt auth

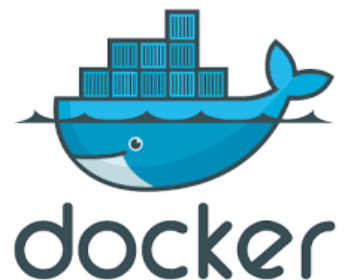
Advanced exercises.

1 – Dockerize your application.

Running plumber APIs from a Docker container is easy and if you can do this you can deploy it pretty much anywhere including Azure, AWS, Google Cloud

- ☐ Download and install Docker <https://www.docker.com/get-started/>
- ☐ Start Docker
- ☐ Inspect the dockerfile in the project root (see next slide)
- ☐ Open a command prompt:
 - ☐ Navigate to the project directory (`cd`)
 - ☐ Run the following commands:
 - ☐ `docker build -t plumber_workshop .`
 - ☐ `docker run --rm -p 80:80 plumber_workshop`
- ☐ Test the API running at localhost:80 using a method of your choice (easiest way is typing <http://localhost:80/ok> into the browser)

Congratulations, your API is running in a Docker container!



The Dockerfile.

Environmental variable:

Here the environmental variable for the authorization filter is declared. **WARNING! This is not a secure way to store passwords but functions only for this demonstration**

Copy and install: Add R package to the container, all files will be included in the image including data files. It is possible to create a .dockerignore file that functions in the same way as .gitignore, this way files only required for development can be excluded from this copy command

```
# pull a docker image for a baseline image
# contains linux libraries, R and plumber
FROM rstudio/plumber

# create the ENV object for authentication (demo only)
# WARNING! this is not a secure way to store passwords
ENV plumber_auth_key="[enter password here]"

# install packages
RUN Rscript -e "install.packages(c('remotes', 'CVrisk', 'readr'))"

# copy everything into a build directory and install package
RUN mkdir/build_zone
ADD ./build_zone
WORKDIR /build_zone
RUN Rscript -e 'remotes::install_local(upgrade="never")'

# open port 80 to traffic
EXPOSE 80

# when the container starts, start the main.R script
ENTRYPOINT ["Rscript", "main.R"]
```

Base Docker Image: Dockerfiles often start with a base image for simplicity, here it contains the R language, plumber package and potentially some other things

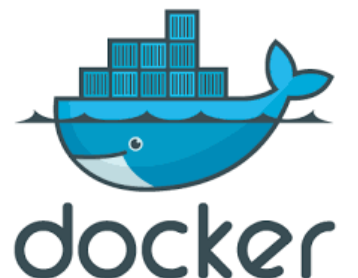
Install required R packages:

Occasionally you will see errors on build that will indicate further package requirement e.g. here we do not call `library(readr)` in our project but its installation is necessary for `@serializer tsv`

Open port: necessary to open a connection between the API and the end-user

Start the application:

uses the Rscript command to run the main.R script when the container starts



2–Performance, futures and promises.

- R is a single-threaded language with no native support for parallel computation like Julia or C++
- This means, without special consideration, the performance of your API is only as efficient as your code
- Each new HTTP request will form a queue and will execute only when the previous request has been served
- One way to overcome this is to use a load balancer, this is usually part of a deployment setup, which distributes incoming traffic over multiple instances of your API
- However, it is also possible to achieve asynchronous code using the `future` package in conjunction with plumber

<https://www.rstudio.com/resources/rstudioglobal-2021/plumber-and-future-async-web-apis/>

- In the next exercise we will see this in action.

2–Instructions.

- Navigate to the file `inst/api_future/plumber_future.R`
- Notice the slow function has commented out the future expression
- Run the API and open two command prompts
- In the first send a request to the slow endpoint, in the second send a request to the fast endpoint

```
curl "http://127.0.0.1:[enter-port]/slow"
```

```
curl "http://127.0.0.1:[enter-port]/fast"
```

- Notice the fast endpoint waits for the slow endpoint to finish
- Now uncomment the future expression in `/slow` and run again
- If all goes well, your fast expression should execute instantly!

Further information.

Postman – instructions.

Postman

- ❑ Download and install Postman
- ❑ On startup you can skip creating an account
- ❑ Navigate to the new HTTP request section (while in a workspace see the + for a new tab)
- ❑ Enter the URL including endpoint path
- ❑ Add data into the body in JSON format →
- ❑ Send the request
- ❑ More details next slide...

JSON input data

```
{  
  "sex" : "male",  
  "age" : 50,  
  "bmi" : 22,  
  "sbp" : 134,  
  "bp_med" : 1,  
  "smoker" : 1,  
  "diabetes" : 0  
}
```

POSTMAN HTTP request using JSON.

The image shows a screenshot of the Postman application interface with several annotations pointing to specific features:

- Collection name:** group of related requests (points to the 'plumber_demo / cvd event risk' breadcrumb).
- Request name:** identifier for request (points to the 'POST cvd event risk' header).
- API URL:** base API address + endpoint path (points to the 'localhost:80/cvd_event_risk' URL).
- Request component:** Body = input data (points to the 'Body' tab in the request editor).
- Request method:** GET, POST, PUT etc. (HTTP verb) (points to the 'POST' dropdown menu).
- Request body:** data to send to API (points to the JSON body text area).
- Body data format (JSON):** JSON is most common – it is a named list of inputs (points to the 'JSON' dropdown in the body type selection).
- Body type (raw):** Further define with the body data format drop-down tab (points to the 'raw' radio button).
- Send:** send the HTTP request to the API (points to the 'Send' button).
- Response:** Your results (points to the 'Your 10-year CVD risk is 26.55%' response text).

The JSON body data shown is:

```
{  "gender": "male",  "age": 45,  "bmi": 37,  "sbp": 134,  "bp_med": 1,  "smoker": 1,  "diabetes": 0}
```

The response status is 200 OK, and the response text is: "Your 10-year CVD risk is 26.55%".

Things I didn't cover.

- There is much I didn't cover here that is worth knowing – here are some examples:
 - **Parsers:** The @parser filter can be used to define the format of the post body
 - <https://www.rplumber.io/reference/parsers.html>
 - **File upload:** APIs can be configured to accept files as inputs (use in conjunction with parsers)
 - <https://www.rplumber.io/articles/annotations.html>
 - **Static files:** Static file can be served with your API
 - <https://www.rplumber.io/reference/PlumberStatic.html>
 - **Programmatic usage:** instead of using `#*` comments, you can explicitly code your API
 - <https://www.rplumber.io/articles/programmatic-usage.html>

Further information.

- Official documentation
 - <https://www.rplumber.io/>
- Cheat sheet
 - <https://raw.githubusercontent.com/rstudio/cheatsheets/main/plumber.pdf>
- Code organization and packaging
 - <https://github.com/ozean12/plungr> - still under development
 - <https://community.rstudio.com/t/plumber-api-and-package-structure/18099/11>
- Authentication and authorization
 - <https://github.com/jandix/sealr>

Other R API frameworks

- Rest R serve:
<https://restrserve.org/>
- BeakR:
<https://github.com/MazamaScience/beakr>
- Ambiorix:
<https://github.com/devOpifex/ambiorix>
- Fiery:
<https://github.com/thomasp85/fiery>

REST APIs with plumber: : CHEAT SHEET



Introduction to REST APIs

Web APIs use **HTTP** to communicate between **client** and **server**.

HTTP



HTTP is built around a **request** and a **response**. A **client** makes a request to a **server**, which handles the request and provides a response. Requests and responses are specially formatted text containing details and data about the exchange between client and server.

REQUEST

```
curl -v "http://httpbin.org/get"

#> GET / get HTTP/1.1
#> Host: httpbin.org
#> User-Agent: curl/7.55.1
#> Accept: */*
#
# Request Body
```

Annotations: Path, HTTP Method, Headers, HTTP Version, Message body

RESPONSE

```
#< HTTP/1.1 200 OK
#< Connection: keep-alive
#< Date: Thu, 02 Aug 2018 18:22:22 GMT
#
# Response Body
```

Annotations: HTTP Version, Status code, Reason phrase, Headers, Message body

Plumber: Build APIs with R

Plumber uses special comments to turn any arbitrary R code into API endpoints. The example below defines a function that takes the **msg** argument and returns it embedded in additional text.

```
library(plumber)

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

Annotations: Plumber comments begin with #*, @decorators define API characteristics, HTTP Method, /<path> is used to define the location of the endpoint

Plumber pipeline

Plumber endpoints contain R code that is executed in response to an HTTP request. Incoming requests pass through a set of mechanisms before a response is returned to the client.

FILTERS

Filters can forward requests (after potentially mutating them), throw errors, or return a response without forwarding the request. Filters are defined similarly to endpoints using the **@filter [name]** tag. By default, filters apply to all endpoints. Endpoints can opt out of filters using the **@preempt** tag.

PARSER

Parsers determine how Plumber parses the incoming request body. By default Plumber parses the request body as JavaScript Object Notation (JSON). Other parsers, including custom parsers, are identified using the **@parser [parser name]** tag. All registered parsers can be viewed with **registered_parsers()**.

ENDPOINT

Endpoints define the R code that is executed in response to incoming requests. These endpoints correspond to HTTP methods and respond to incoming requests that match the defined method.

METHODS

- **@get** - request a resource
- **@post** - send data in body
- **@put** - store / update data
- **@delete** - delete resource
- **@head** - no request body
- **@options** - describe options
- **@patch** - partial changes
- **@use** - use all methods

SERIALIZER

Serializers determine how Plumber returns results to the client. By default Plumber serializes the R object returned into JavaScript Object Notation (JSON). Other serializers, including custom serializers, are identified using the **@serializer [serializer name]** tag. All registered serializers can be viewed with **registered_serializers()**.

```
library(plumber)

#* @filter log
function(req, res) {
  print(req$HTTP_USER_AGENT)
  forward()
}

#* Convert request body to uppercase
#* @preempt log
#* @parser json
#* @post /uppercase
#* @serializer json
function(req, res) {
  toupper(req$body)
}
```

Annotations: Identify as filter, Forward request, Endpoint description, Parser, HTTP Method, Filter name, Opt out of the log filter, Endpoint path, Serializer

Running Plumber APIs

Plumber APIs can be run programmatically from within an R session.

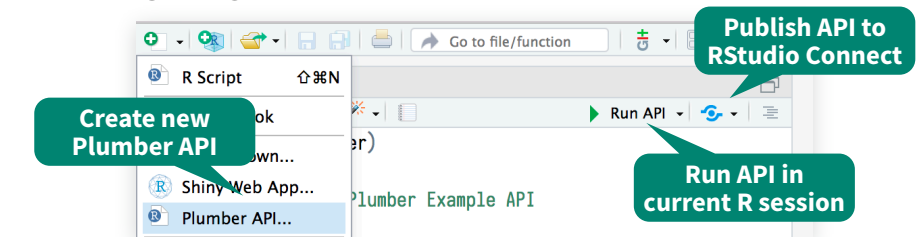
```
library(plumber)

plumb("plumber.R") %>%
  pr_run(port = 5762)
```

Annotations: Path to API definition, Specify API port

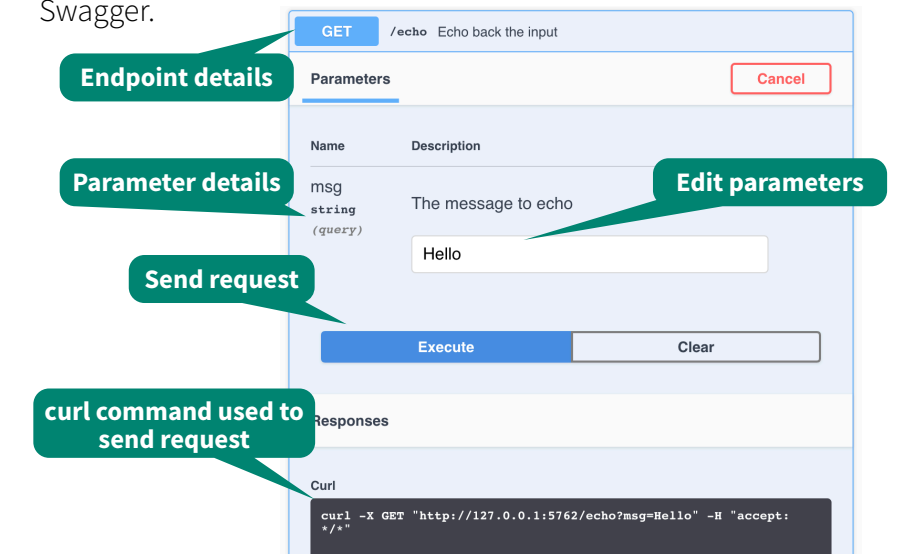
This runs the API on the host machine supported by the current R session.

IDE INTEGRATION



Documentation

Plumber APIs automatically generate an OpenAPI specification file. This specification file can be interpreted to generate a dynamic user-interface for the API. The default interface is generated via Swagger.



Interact with the API

Once the API is running, it can be interacted with using any HTTP client. Note that using **httr** requires using a separate R session from the one serving the API.

```
(resp <- httr::GET("localhost:5762/echo?msg=Hello"))
#> Response [http://localhost:5762/echo?msg=Hello]
#>   Date: 2018-08-07 20:06
#>   Status: 200
#>   Content-Type: application/json
#>   Size: 35 B
httr::content(resp, as = "text")
#> [1] "{\\\"msg\\\": [\\\"The message is: 'Hello'\\\"]}"
```

Programmatic Plumber

Tidy Plumber

Plumber is exceptionally customizable. In addition to using special comments to create APIs, APIs can be created entirely programmatically. This exposes additional features and functionality. Plumber has a convenient “tidy” interface that allows API routers to be built piece by piece. The following example is part of a standard `plumber.R` file.

```
library(plumber)

#* @plumber
function(pr) {
  pr %>%
    pr_get(path = "/echo",
            handler = function(msg = "") {
              list(msg = paste0(
                "The message is: ",
                msg,
                ""
              ))
            }) %>%
    pr_get(path = "/plot",
            handler = function() {
              rand <- rnorm(100)
              hist(rand)
            },
            serializer = serializer_png()) %>%
    pr_post(path = "/sum",
            handler = function(a, b) {
              as.numeric(a) + as.numeric(b)
            })
}
```

Use @plumber tag

Function that accepts and modifies a plumber router

“Tidy” functions for building out Plumber API

OpenAPI

Plumber automatically creates an OpenAPI specification file based on Plumber comments. This file can be further modified using `pr_set_api_spec()` with either a function that modifies the existing specification or a path to a `.yaml` or `.json` specification file.

```
library(plumber)

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

#* @plumber
function(pr) {
  pr %>%
    pr_set_api_spec(function(spec) {
      spec$paths[["/echo"]]$get$summary <-
        "Echo back the input"
      spec
    })
}
```

Function that receives and modifies the existing specification

Return the updated specification

By default, Swagger is used to interpret the OpenAPI specification file and generate the user interface for the API. Other interpreters can be used to adjust the look and feel of the user interface via `pr_set_docs()`.



Advanced Plumber

REQUEST and RESPONSE

Plumber provides access to special `req` and `res` objects that can be passed to Plumber functions. These objects provide access to the request submitted by the client and the response that will be sent to the client. Each object has several components, the most helpful of which are outlined below:

Name	Example	Description
req		
<code>req\$pr</code>	<code>plumber::pr()</code>	The Plumber router processing the request
<code>req\$body</code>	<code>list(a=1)</code>	Typically the same as <code>argsBody</code>
<code>req\$argsBody</code>	<code>list(a=1)</code>	The parsed body output
<code>req\$argsPath</code>	<code>list(c=3)</code>	The values of the path arguments
<code>req\$argsQuery</code>	<code>list(e=5)</code>	The parsed output from <code>req\$QUERY_STRING</code>
<code>req\$cookies</code>	<code>list(cook = "a")</code>	A list of cookies
<code>req\$REQUEST_METHOD</code>	"GET"	The method used for the HTTP request
<code>req\$PATH_INFO</code>	"/"	The path of the incoming HTTP request
<code>req\$HTTP_*</code>	"HTTP_USER_AGENT"	All of the HTTP headers sent with the request
<code>req\$bodyRaw</code>	<code>charToRaw("a=1")</code>	The <code>raw()</code> contents of the request body
res		
<code>res\$headers</code>	<code>list(header = "abc")</code>	HTTP headers to include in the response
<code>res\$setHeader()</code>	<code>setHeader("foo", "bar")</code>	Sets an HTTP header
<code>res\$setCookie()</code>	<code>setCookie("foo", "bar")</code>	Sets an HTTP cookie on the client
<code>res\$removeCookie</code>	<code>removeCookie("foo")</code>	Removes an HTTP cookie
<code>res\$body</code>	<code>"{\"a\": [1]}"</code>	Serialized output
<code>res\$status</code>	200	The response HTTP status code
<code>res\$toResponse()</code>	<code>toResponse()</code>	A list of status, headers, and body

ASYNCH PLUMBER

Plumber supports asynchronous execution via the `future` R package. This pattern allows Plumber to concurrently process multiple requests.

```
library(plumber)
future::plan("multisession")

#* @get /slow
function() {
  promises::future_promise({
    slow_calc()
  })
}
```

Set the execution plan

Slow calculation

MOUNTING ROUTERS

Plumber routers can be combined by mounting routers into other routers. This can be beneficial when building routers that involve several different endpoints and you want to break each component out into a separate router. These separate routers can even be separate files loaded using `plumb()`.

```
library(plumber)

route <- pr() %>%
  pr_get("/foo", function() "foo")

#* @plumber
function(pr) {
  pr %>%
    pr_mount("/bar", route)
}
```

Create an initial router

Mount one router into another

In the above example, the final route is `/bar/foo`.

RUNNING EXAMPLES

Some packages, like the Plumber package itself, may include example Plumber APIs. Available APIs can be viewed using `available_apis()`. These example APIs can be run with `plumb_api()` combined with `pr_run()`.

```
library(plumber)

plumb_api(package = "plumber",
          name = "01-append",
          edit = TRUE) %>%
  pr_run()
```

Identify the package name and API name

Run the example API

Optionally open the file for editing

Deploying Plumber APIs

Once Plumber APIs have been developed, they often need to be deployed somewhere to be useful. Plumber APIs can be deployed in a variety of different ways. One of the easiest way to deploy Plumber APIs is using RStudio Connect, which supports push button publishing from the RStudio IDE.

Run API

Publish API...

Manage Accounts...

Thank you!