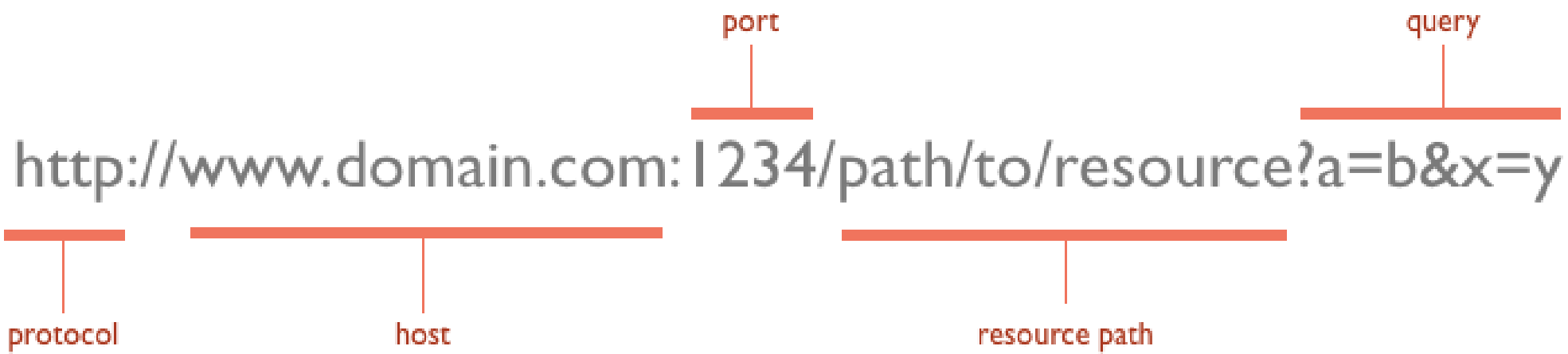
# Web APIs

## Lecture 14

Dr. Colin Rundel

# URLs


port
query
http://www.domain.com:1234/path/to/resource?a=b&x=y
protocol
host
resource path

# Query Strings

Provides named argument(s) and value(s) that modify the behavior of the resulting page.

Format generally follows:

```
?arg1=value1&arg2=value2&arg3=value3
```

Some quick examples,

- http://maps.googleapis.com/maps/api/geocode/json?sensor=false&address=1600+Amphitheatre+Parkway
- https://swapi.dev/api/people/?search=r2

# URL encoding

This will often be handled automatically by your web browser or other tool, but it is useful to know a bit about what is happening

- Spaces will be encoded as '+' or '%20'
- Certain characters are reserved and will be replaced with the percent-encoded version within a URL

| ! | # | $ | & | ' | ( | ) |
|---|---|---|---|---|---|---|
| %21 | %23 | %24 | %26 | %27 | %28 | %29 |
| * | + | , | / | : | ; | = |
| %2A | %2B | %2C | %2F | %3A | %3B | %3D |
| ? | @ | [ | ] | | | |
| %3F | %40 | %5B | %5D | | | |

- Characters that cannot be converted to the correct charset are replaced with HTML numeric character references (e.g. a Σ would be encoded as Σ )

# Examples

```
1 URLencode("http://lmgtfy.com/?q=hello world")
```

```
[1] "http://lmgtfy.com/?q=hello%20world"
```

```
1 URLdecode("http://lmgtfy.com/?q=hello%20world")
```

```
[1] "http://lmgtfy.com/?q=hello world"
```

```
1 URLencode("!#$&'()*+,/:;=?@[]")
```

```
[1] "!#$&'()*+,/:;=?@[]"
```

```
1 URLencode("!#$&'()*+,/:;=?@[]", reserved = TRUE)
```

```
[1] "%21%23%24%26%27%28%29%2A%2B%2C%2F%3A%3B%3D%3F%40%5B%5D"
```

```
1 URLencode("!#$&'()*+,/:;=?@[]", reserved = TRUE) |>
2   URLdecode()
```

```
[1] "!#$&'()*+,/:;=?@[]"
```

```
1 URLencode("Σ")
```

```
[1] "%CE%A3"
```

```
1 URLdecode("%CE%A3")
```

```
[1] "Σ"
```

# RESTful APIs

# REST

**RE**presentational **S**tate **T**ransfer

- Describes an architectural style for web services (not a standard)
- All communication via HTTP requests and responses
- Key features:
  - **Stateless** - each request is self-contained; no session state stored on server
  - **Addressable** - resources identified by URLs (endpoints)
  - **Uniform interface** - standard HTTP methods (GET, POST, PUT, DELETE)
  - **Cacheable** - responses can be cached to improve performance
- Resources are represented in standard formats (typically JSON or XML) and delivered in the response body

# GitHub API

GitHub provides a REST API that allows you to interact with most of the data available on the website.

There is extensive documentation and a huge number of endpoints to use - almost anything that can be done on the website can also be done via the API.

GitHub REST API

# Demo 1 - GitHub API - Basic access

For this demo we will be using the GitHub REST API to access public data about users and repositories.

We will be making unauthenticated requests, which are subject to stricter rate limits but do not require credentials.

- Get a user
- List organization repositories

# Listing Repos

```r
z = jsonlite::read_json("https://api.github.com/orgs/sta323-sp26/repos")
length(z)
```

```
[1] 2
```

```r
str(z, max.level = 2)
```

```
List of 2
 $ :List of 83
  ..$ id                          : int 1129149160
  ..$ node_id                     : chr "R_kgDOQ01y6A"
  ..$ name                        : chr "sta323-sp26.github.io"
  ..$ full_name                   : chr "sta323-sp26/sta323-sp26.github.io"
  ..$ private                     : logi FALSE
  ..$ owner                       :List of 19
  ..$ html_url                    : chr "https://github.com/sta323-sp26/sta323-sp26.github.io"
  ..$ description                 : chr "Duke Sta 323 - Spring 2026 - Course Website"
  ..$ fork                        : logi FALSE
  ..$ url                         : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ forks_url                   : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ keys_url                    : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ collaborators_url           : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ teams_url                   : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ hooks_url                   : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ issue_events_url            : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ events_url                  : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ assignees_url               : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ branches_url                : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
  ..$ tags_url                    : chr "https://api.github.com/repos/sta323-sp26/sta323-sp26.gith
```

```
1 z |> map_chr("full_name")
```

```
[1] "sta323-sp26/sta323-sp26.github.io" "sta323-sp26/exercises"
```

```r
z = jsonlite::read_json(
  "https://api.github.com/orgs/tidyverse/repos"
)
length(z)
```

```
[1] 30
```

```r
z |> map_chr("full_name")
```

```
 [1] "tidyverse/ggplot2"         "tidyverse/lubridate"
 [3] "tidyverse/stringr"         "tidyverse/dplyr"
 [5] "tidyverse/readr"           "tidyverse/magrittr"
 [7] "tidyverse/tidyr"           "tidyverse/nycflights13"
 [9] "tidyverse/rvest"           "tidyverse/purrr"
[11] "tidyverse/haven"           "tidyverse/readxl"
[13] "tidyverse/reprex"          "tidyverse/tibble"
[15] "tidyverse/multidplyr"      "tidyverse/dtplyr"
[17] "tidyverse/hms"             "tidyverse/modelr"
[19] "tidyverse/forcats"         "tidyverse/tidyverse"
[21] "tidyverse/tidytemplate"    "tidyverse/blob"
[23] "tidyverse/ggplot2-docs"    "tidyverse/glue"
[25] "tidyverse/style"           "tidyverse/dbplyr"
[27] "tidyverse/googledrive"     "tidyverse/googlesheets4"
[29] "tidyverse/tidyverse.org"   "tidyverse/datascience-box"
```

# Pagination

Many REST APIs limit the number of results returned in a single response to manage server load and improve performance. When working with large(r) datasets, you'll need to make multiple requests to retrieve all results.

Common pagination approaches:

- Offset-based - specify starting position and number of items (`?offset=20&limit=10`)
- Page-based - specify page number and page size (`?page=2&per_page=30`)
- Cursor-based - use a token/cursor pointing to next set of results
- Link header - server provides URLs to next/previous pages in response headers

# GitHub API Pagination

GitHub uses page-based and link header pagination:

- Query parameters:
  - `per_page` - number of items per page (default: 30, max: 100)
  - `page` - page number to retrieve (default: 1)
- Link header: GitHub includes a `Link` header in responses with URLs for:
  - `next` - next page
  - `prev` - previous page
  - `first` - first page
  - `last` - last page

```r
z = jsonlite::read_json(
  "https://api.github.com/orgs/tidyverse/repos?page=2"
)
length(z)
```

```
[1] 15
```

```r
z |> map_chr("full_name")
```

```
 [1] "tidyverse/tidyversedashboard" "tidyverse/dsbox"
 [3] "tidyverse/design"             "tidyverse/tidyeval"
 [5] "tidyverse/tidy-dev-day"       "tidyverse/funs"
 [7] "tidyverse/vroom"              "tidyverse/website-analytics"
 [9] "tidyverse/tidyups"            "tidyverse/duckplyr"
[11] "tidyverse/code-review"        "tidyverse/ellmer"
[13] "tidyverse/ragnar"             "tidyverse/vitals"
[15] "tidyverse/ggbot2"
```

```r
z = jsonlite::read_json(
  "https://api.github.com/orgs/tidyverse/repos?per_page=100"
)
length(z)
```

```
[1] 45
```

```r
z |> map_chr("full_name")
```

```
 [1] "tidyverse/ggplot2"           "tidyverse/lubridate"
 [3] "tidyverse/stringr"           "tidyverse/dplyr"
 [5] "tidyverse/readr"             "tidyverse/magrittr"
 [7] "tidyverse/tidyr"             "tidyverse/nycflights13"
 [9] "tidyverse/rvest"             "tidyverse/purrr"
[11] "tidyverse/haven"             "tidyverse/readxl"
[13] "tidyverse/reprex"            "tidyverse/tibble"
[15] "tidyverse/multidplyr"        "tidyverse/dtplyr"
[17] "tidyverse/hms"               "tidyverse/modelr"
[19] "tidyverse/forcats"           "tidyverse/tidyverse"
[21] "tidyverse/tidytemplate"      "tidyverse/blob"
[23] "tidyverse/ggplot2-docs"      "tidyverse/glue"
[25] "tidyverse/style"             "tidyverse/dbplyr"
[27] "tidyverse/googledrive"       "tidyverse/googlesheets4"
[29] "tidyverse/tidyverse.org"     "tidyverse/datascience-box"
[31] "tidyverse/tidyversedashboard" "tidyverse/dsbox"
[33] "tidyverse/design"            "tidyverse/tidyeval"
[35] "tidyverse/tidy-dev-day"      "tidyverse/funs"
[37] "tidyverse/vroom"             "tidyverse/website-analytics"
[39] "tidyverse/tidyups"           "tidyverse/duckplyr"
[41] "tidyverse/code-review"       "tidyverse/ellmer"
[43] "tidyverse/ragnar"            "tidyverse/vitals"
[45] "tidyverse/ggbot2"
```

# Demo 2 - Authenticated Endpoint(s)

The information we've been retrieving is all public, but the GitHub API also has endpoints that require authentication (e.g. to access private user data, create repositories, etc.). This is why when requesting the `sta323-sp26` repos we get only 2 results (all of the private repos are hidden).

One example of this is the `/user` endpoint which returns information about the *authenticated* user. If we try to access this endpoint without authentication, we get an error:

```
1 jsonlite::read_json("https://api.github.com/user")
```

```
Warning in open.connection(con, "rb"): cannot open URL
'https://api.github.com/user': HTTP status was '401 Unauthorized'
```

```
Error in `open.connection()`:
! cannot open the connection to 'https://api.github.com/user'
```

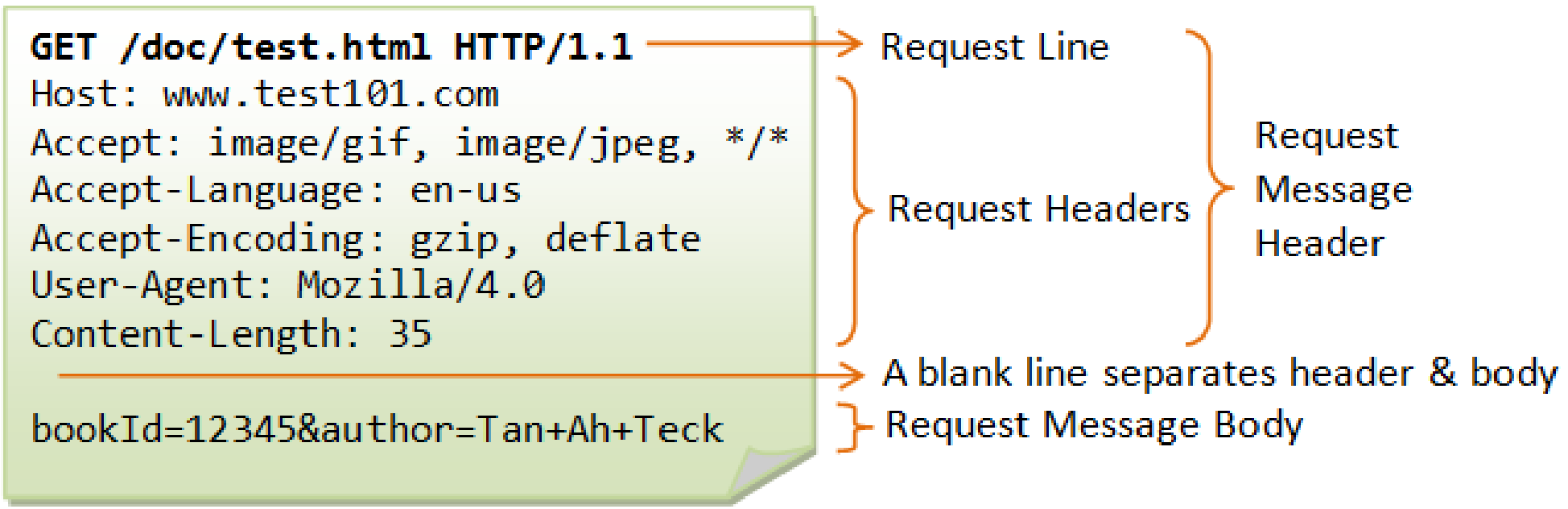

httr2

# Background

`httr2` is a package designed around the construction and handling of HTTP requests and responses. It is a rewrite of the `httr` package and includes the following features:

- Pipeable API
- Explicit request object, with support for
  - rate limiting
  - retries
  - OAuth
  - Secure secret storage
- Explicit response object, with support for
  - error codes / reporting
  - common body encoding (e.g. json, etc.)

# Structure of an HTTP Request



# HTTP Methods / Verbs

| Verb | Purpose | GitHub API example |
|---|---|---|
| `GET` | Fetch a resource | Get a user, list org repos |
| `POST` | Create a new resource | Create a gist, open an issue |
| `PUT` | Full update of a resource | Replace file contents in a repo |
| `PATCH` | Partial update of a resource | Update a repo's description |
| `DELETE` | Remove a resource | Delete a gist |

`GET` and `POST` are by far the most common when consuming APIs.

Less common verbs: `HEAD`, `TRACE`, `OPTIONS`.

# httr2 request objects

A new request object is constructed via `request()` which is then modified via `req_*()` functions

Some useful functions:

- `request()` - initialize a request object
- `req_method()` - set HTTP method
- `req_url_query()` - add query parameters to URL
- `req_body_json()`, `req_body_raw()`, etc. - set body content (various formats and sources)
- `req_user_agent()` - set request user-agent header
- `req_dry_run()` - shows the exact request that will be made

# Example

```
req = request("https://api.github.com/orgs/tidyverse/repos") |>
  req_url_query(per_page=100) |>
  req_user_agent("Sta323 Demo Client/0.1")
```

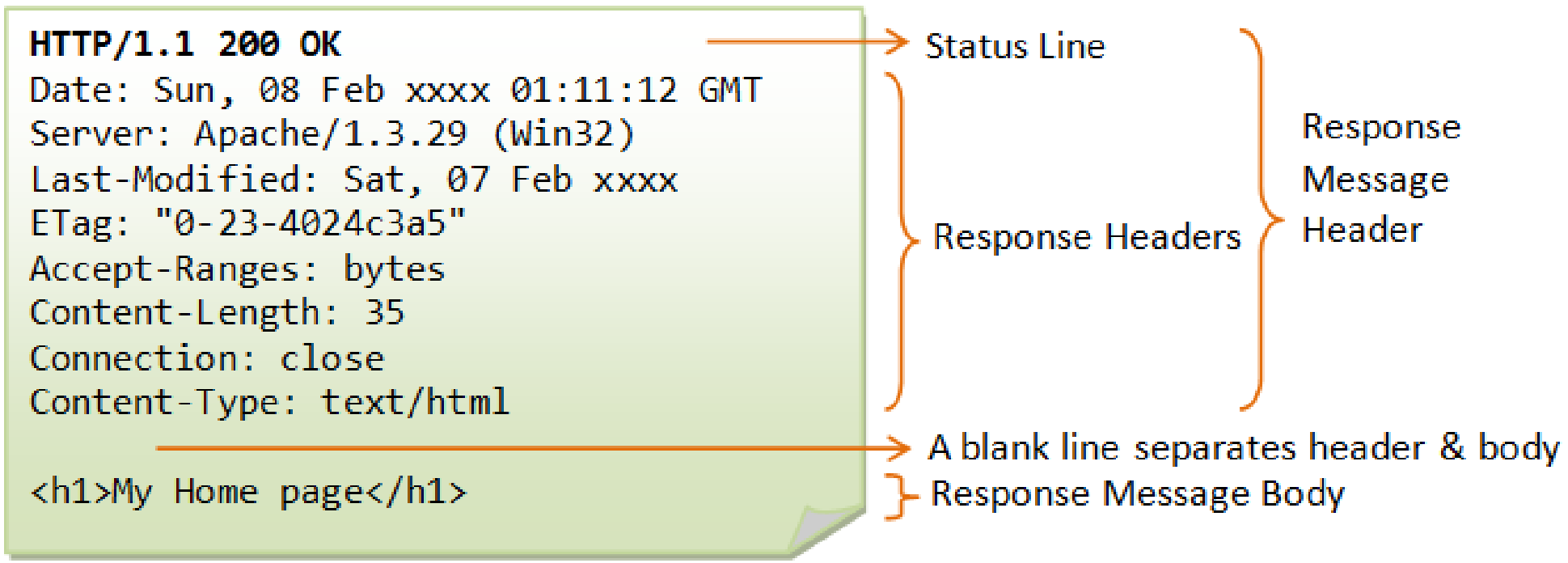
```
req
```

```
<httr2_request>
GET https://api.github.com/orgs/tidyverse/repos?per_page=100
Body: empty
Options:
* useragent: "Sta323 Demo Client/0.1"
```

```
req |> req_dry_run()
```

```
GET /orgs/tidyverse/repos HTTP/1.1
accept: */*
accept-encoding: deflate, gzip
host: api.github.com
user-agent: Sta323 Demo Client/0.1
```

# Structure of an HTTP Response



# Status Codes

In general,

- 2XX - Success
- 3XX - Redirection
- 4XX - Client error
- 5XX - Server error

Some specific examples:

| Code | Description | Meaning |
|---|---|---|
| 200 | OK | Request succeeded |
| 201 | Created | Resource created (POST/PUT success) |
| 204 | No Content | Success with no response body |
| 301 | Moved Permanently | Permanent redirect |
| 400 | Bad Request | Malformed request syntax or parameters |
| 401 | Unauthorized | Authentication required or failed |
| 403 | Forbidden | Authenticated but not permitted |
| 404 | Not Found | Resource doesn’t exist |
| 429 | Too Many Requests | Rate limit exceeded |
| 500 | Internal Server Error | Server-side failure |
| 503 | Service Unavailable | Server temporarily unavailable |

# httr2 response objects

Once a request is made via `req_perform()`, a response object will be returned (the most recent response can also be retrieved via `last_response()`). Contents of the response are accessed via the `resp_*()` functions

Some useful functions:

- `resp_status()` - extract HTTP status code
- `resp_status_desc()` - text description of the status code
- `resp_content_type()` - extract content type and encoding
- `resp_body_json()`, `resp_body_html()`, etc. - extract body using specified format
- `resp_headers()` - extract response headers

# Demo 3 - httr2 + GitHub

# Basic Request

```
resp = request("https://api.github.com/users/rundel") |>
  req_perform()
```

```
resp |> resp_status()
```

```
[1] 200
```

```
resp |> resp_status_desc()
```

```
[1] "OK"
```

```
resp |> resp_content_type()
```

```
[1] "application/json"
```

```
resp |> resp_body_json() |> str()
```

```
List of 33
 $ login             : chr "rundel"
 $ id                : int 273926
 $ node_id           : chr "MDQ6VXNlcjI3MzkyNg=="
 $ avatar_url        : chr "https://avatars.githubusercontent.com/u/273926?v=4"
 $ gravatar_id       : chr ""
 $ url               : chr "https://api.github.com/users/rundel"
 $ html_url          : chr "https://github.com/rundel"
 $ followers_url     : chr "https://api.github.com/users/rundel/followers"
 $ following_url     : chr "https://api.github.com/users/rundel/following{/other_user
```

```
$ gists_url          : chr "https://api.github.com/users/rundel/gists{/gist_id}"
$ starred_url        : chr "https://api.github.com/users/rundel/starred{/owner}{/repo
$ subscriptions_url  : chr "https://api.github.com/users/rundel/subscriptions"
$ organizations_url  : chr "https://api.github.com/users/rundel/orgs"
$ repos_url          : chr "https://api.github.com/users/rundel/repos"
$ events_url         : chr "https://api.github.com/users/rundel/events{/privacy}"
$ received_events_url: chr "https://api.github.com/users/rundel/received_events"
$ type               : chr "User"
$ user_view_type     : chr "public"
$ site_admin         : logi FALSE
```

# Pagination w/ httr2

```
request("https://api.github.com/orgs/tidyverse/repos") |>
  req_auth_bearer_token(gitcreds::gitcreds_get()$password) |>
  req_perform() |>
  resp_body_json() |>
  map_chr("full_name")
```

```
 [1] "tidyverse/ggplot2"          "tidyverse/lubridate"
 [3] "tidyverse/stringr"          "tidyverse/dplyr"
 [5] "tidyverse/readr"            "tidyverse/magrittr"
 [7] "tidyverse/tidyr"            "tidyverse/nycflights13"
 [9] "tidyverse/rvest"            "tidyverse/purrr"
[11] "tidyverse/haven"            "tidyverse/readxl"
[13] "tidyverse/reprex"           "tidyverse/tibble"
[15] "tidyverse/multidplyr"       "tidyverse/dtplyr"
[17] "tidyverse/hms"              "tidyverse/modelr"
[19] "tidyverse/forcats"          "tidyverse/tidyverse"
[21] "tidyverse/tidytemplate"     "tidyverse/blob"
[23] "tidyverse/ggplot2-docs"     "tidyverse/glue"
[25] "tidyverse/style"            "tidyverse/dbplyr"
[27] "tidyverse/googledrive"      "tidyverse/googlesheets4"
[29] "tidyverse/tidyverse.org"    "tidyverse/datascience-box"
```

```
request("https://api.github.com/orgs/t
  req_url_query(page=2) |>
  req_perform() |>
  resp_body_json() |>
  map_chr("full_name")
```

```
 [1] "tidyverse/tidyversedashboard"
 [2] "tidyverse/dsbox"
 [3] "tidyverse/design"
 [4] "tidyverse/tidyeval"
 [5] "tidyverse/tidy-dev-day"
 [6] "tidyverse/funs"
 [7] "tidyverse/vroom"
 [8] "tidyverse/website-analytics"
 [9] "tidyverse/tidyups"
[10] "tidyverse/duckplyr"
[11] "tidyverse/code-review"
[12] "tidyverse/ellmer"
[13] "tidyverse/ragnar"
[14] "tidyverse/vitals"
[15] "tidyverse/ggbot2"
```

```
request("https://api.github.com/orgs/t
  req_url_query(per_page=100) |>
  req_perform() |>
  resp_body_json() |>
  map_chr("full_name")
```

```
 [1] "tidyverse/ggplot2"
 [2] "tidyverse/lubridate"
 [3] "tidyverse/stringr"
 [4] "tidyverse/dplyr"
 [5] "tidyverse/readr"
 [6] "tidyverse/magrittr"
 [7] "tidyverse/tidyr"
 [8] "tidyverse/nycflights13"
 [9] "tidyverse/rvest"
[10] "tidyverse/purrr"
[11] "tidyverse/haven"
[12] "tidyverse/readxl"
[13] "tidyverse/reprex"
[14] "tidyverse/tibble"
[15] "tidyverse/multidplyr"
[16] "tidyverse/dtplyr"
[17] "tidyverse/hms"
[18] "tidyverse/modelr"
[19] "tidyverse/forcats"
[20] "tidyverse/tidyverse"
```

# `req_perform_iterative()`

Pagination bookkeeping is annoying - this can (sometimes) be handled automatically with `req_perform_iterative()` - see `?iterate_with_offset` for pre-built helpers.

```
1 resps = request("https://api.github.com/orgs/tidyverse/repos") |>
2   req_url_query(per_page=15) |>
3   req_perform_iterative(next_req = iterate_with_link_url())
```

```
iterating ■■■■■                                10% | ETA:  9s
iterating ■■■■■■                               15% | ETA:  8s
```

```
1 resps
```

```
[[1]]
<httr2_response>
GET https://api.github.com/orgs/tidyverse/repos?per_page=15
Status: 200 OK
Content-Type: application/json
Body: In memory (90375 bytes)

[[2]]
<httr2_response>
GET https://api.github.com/organizations/22032646/repos?per_page=15&page=2
Status: 200 OK
Content-Type: application/json
Body: In memory (91573 bytes)
```

```
[[3]]
<httr2_response>
GET https://api.github.com/organizations/22032646/repos?per_page=15&page=3
Status: 200 OK
Content-Type: application/json
Body: In memory (89916 bytes)
```

```r
resps |>
  map(resp_body_json) |>
  purrr::list_flatten() |>
  map_chr("full_name")
```

```
 [1] "tidyverse/ggplot2"            "tidyverse/lubridate"
 [3] "tidyverse/stringr"            "tidyverse/dplyr"
 [5] "tidyverse/readr"              "tidyverse/magrittr"
 [7] "tidyverse/tidyr"              "tidyverse/nycflights13"
 [9] "tidyverse/rvest"              "tidyverse/purrr"
[11] "tidyverse/haven"              "tidyverse/readxl"
[13] "tidyverse/reprex"             "tidyverse/tibble"
[15] "tidyverse/multidplyr"         "tidyverse/dtplyr"
[17] "tidyverse/hms"                "tidyverse/modelr"
[19] "tidyverse/forcats"            "tidyverse/tidyverse"
[21] "tidyverse/tidytemplate"       "tidyverse/blob"
[23] "tidyverse/ggplot2-docs"       "tidyverse/glue"
[25] "tidyverse/style"              "tidyverse/dbplyr"
[27] "tidyverse/googledrive"        "tidyverse/googlesheets4"
[29] "tidyverse/tidyverse.org"      "tidyverse/datascience-box"
[31] "tidyverse/tidyversedashboard" "tidyverse/dsbox"
[33] "tidyverse/design"             "tidyverse/tidyeval"
[35] "tidyverse/tidy-dev-day"       "tidyverse/funs"
[37] "tidyverse/vroom"              "tidyverse/website-analytics"
[39] "tidyverse/tidyups"            "tidyverse/duckplyr"
```

# Error Handling in httr2

By default, `httr2` throws an R error for HTTP error responses (4xx and 5xx):

```
request("https://api.github.com/user") |>
  req_perform()
```

```
Error in `req_perform()`:
! HTTP 401 Unauthorized.
```

```
resp = request("https://api.github.com/user") |>
  req_error(is_error = function(resp) FALSE) |>
  req_perform()
```

```
resp |> resp_status()
```

```
[1] 401
```

```
resp |> resp_status_desc()
```

```
[1] "Unauthorized"
```

```
resp |> resp_body_json() |> str()
```

```
List of 3
 $ message          : chr "Requires authentication"
 $ documentation_url: chr "https://docs.github.com/rest"
 $ status           : chr "401"
```

# Authentication

Most APIs have rate limits and access restrictions:

- Unauthenticated requests - limited rate limits, restricted access
- Authenticated requests - higher rate limits, access to private resources

GitHub API rate limits (per hour):

- Unauthenticated: 60 requests
- Authenticated: 5,000 requests (personal access token)

Authentication is done via HTTP headers, typically:

- `Authorization: Bearer <token>`

Here `Authorization` is the request header and `Bearer <token>` is the value. With `httr2` `req_auth_bearer_token()` can

# GitHub Personal Access Tokens (PATs)

A PAT is a secure alternative to using passwords for API authentication:

- Generated on GitHub: Settings → Developer settings → Personal access tokens
- Choose scopes (permissions) carefully - only grant what's needed
- Two types:
  - Fine-grained tokens - repository-specific access (recommended)
  - Classic tokens - broader access patterns

Best practices:

- Never commit tokens to git repositories
- Store in environment variables (e.g., `.Renviron`)
- Use packages like `gitcreds` or `credentials` to manage tokens securely
- Set expiration dates and rotate tokens regularly

For all of the following examples, I have created a PAT and added via `gitcreds::gitcreds_set()` to store it securely.

# Demo 4 - Using Authentication

```
request("https://api.github.com/user") |>
  req_auth_bearer_token(gitcreds::gitcreds_get()$password) |>
  req_perform() |>
  resp_body_json() |>
  str()
```

```
List of 41
 $ login                   : chr "rundel"
 $ id                      : int 273926
 $ node_id                 : chr "MDQ6VXNlcjI3MzkyNg=="
 $ avatar_url              : chr "https://avatars.githubusercontent.com/u/273926?v=4"
 $ gravatar_id             : chr ""
 $ url                     : chr "https://api.github.com/users/rundel"
 $ html_url                : chr "https://github.com/rundel"
 $ followers_url           : chr "https://api.github.com/users/rundel/followers"
 $ following_url           : chr "https://api.github.com/users/rundel/following{/other_user}"
 $ gists_url               : chr "https://api.github.com/users/rundel/gists{/gist_id}"
 $ starred_url             : chr "https://api.github.com/users/rundel/starred{/owner}{/repo}"
 $ subscriptions_url       : chr "https://api.github.com/users/rundel/subscriptions"
 $ organizations_url       : chr "https://api.github.com/users/rundel/orgs"
 $ repos_url               : chr "https://api.github.com/users/rundel/repos"
 $ events_url              : chr "https://api.github.com/users/rundel/events{/privacy}"
 $ received_events_url     : chr "https://api.github.com/users/rundel/received_events"
 $ type                    : chr "User"
 $ user_view_type          : chr "private"
 $ site_admin              : logi FALSE
 $ name                    : chr "Colin Rundel"
 $ company                 : chr "Duke University"
```

```r
request("https://api.github.com/user") |>
  req_headers(
    Authorization = paste("Bearer", gitcreds::gitcreds_get()$password)
  ) |>
  req_perform() |>
  resp_body_json() |>
  str()
```

```
List of 41
 $ login                   : chr "rundel"
 $ id                      : int 273926
 $ node_id                 : chr "MDQ6VXNlcjI3MzkyNg=="
 $ avatar_url              : chr "https://avatars.githubusercontent.com/u/273926?v=4"
 $ gravatar_id             : chr ""
 $ url                     : chr "https://api.github.com/users/rundel"
 $ html_url                : chr "https://github.com/rundel"
 $ followers_url           : chr "https://api.github.com/users/rundel/followers"
 $ following_url           : chr "https://api.github.com/users/rundel/following{/other_user}"
 $ gists_url               : chr "https://api.github.com/users/rundel/gists{/gist_id}"
 $ starred_url             : chr "https://api.github.com/users/rundel/starred{/owner}{/repo}"
 $ subscriptions_url       : chr "https://api.github.com/users/rundel/subscriptions"
 $ organizations_url       : chr "https://api.github.com/users/rundel/orgs"
 $ repos_url               : chr "https://api.github.com/users/rundel/repos"
 $ events_url              : chr "https://api.github.com/users/rundel/events{/privacy}"
 $ received_events_url     : chr "https://api.github.com/users/rundel/received_events"
 $ type                    : chr "User"
 $ user_view_type          : chr "private"
 $ site_admin              : logi FALSE
 $ name                    : chr "Colin Rundel"
 $ company                 : chr "Duke University"
 $ blog                    : chr "rundel.github.io"
```

# Demo 5 - POST Request

```r
gist = request("https://api.github.com/gists") |>
  req_auth_bearer_token(gitcreds::gitcreds_get()$password) |>
  req_body_json(list(
    description = "Testing 1 2 3 ...",
    files = list("test.R" = list(content = "print('hello world')\n")),
    public = TRUE
  ))

gist |> req_dry_run()
```

```
POST /gists HTTP/1.1
accept: */*
accept-encoding: deflate, gzip
authorization: <REDACTED>
content-length: 105
content-type: application/json
host: api.github.com
user-agent: httr2/1.2.2 r-curl/7.0.0 libcurl/8.14.1

{
  "description": "Testing 1 2 3 ...",
  "files": {
    "test.R": {
      "content": "print('hello world')\n"
    }
  },
  "public": true
}
```

```
1 resp = gist |> req_perform()
2 resp |> resp_status()
```

```
[1] 201
```

```
1 resp |> resp_status_desc()
```

```
[1] "Created"
```

```
1 resp |> resp_body_json() |> pluck("html_url")
```

```
[1] "https://gist.github.com/rundel/c6ff9d3e96cfe56ece328ed710c3be71"
```

# Other Useful httr2 Features

# Rate Limiting - `req_throttle()`

APIs enforce rate limits — exceeding them results in `429 Too Many Requests` or similar errors. `req_throttle()` automatically inserts delays between requests to stay within a limit:

```
# At most 30 requests per minute
req = request("https://api.github.com") |>
  req_throttle(rate = 30 / 60)

# Each req_perform() call on this request will wait if needed
req |> req_url_path("/users/rundel") |> req_perform()
req |> req_url_path("/users/hadley")  |> req_perform()
```

The `rate` argument is **requests per second**. The throttle is applied per-hostname, so all requests to the same host share the same token bucket.

# Automatic Retries - `req_retry()`

Transient failures (network blips, `429`, `503`) can be retried automatically with `req_retry()`:

```
request("https://api.github.com/users/rundel") |>
  req_retry(
    max_tries = 5,
    is_transient = function(resp) {resp_status(resp) %in% c(429, 500, 503)}
  ) |>
  req_perform()
```

Key arguments:

- `max_tries` - maximum total attempts (including the first)
- `max_seconds` - give up after this many seconds total
- `is_transient` - function deciding if a response warrants a retry
- `backoff` - function computing wait time between attempts (default: exponential backoff)

# Response Caching - `req_cache()`

`req_cache()` stores responses on disk and replays them for identical requests, respecting HTTP cache headers:

```
request("https://api.github.com/users/rundel") |>
  req_cache(path = "api_cache/") |>
  req_perform()
```

- The cache is keyed on the full request URL and headers
- Cached responses are reused until they expire (based on `Cache-Control` / `Expires` headers)
- Pass `max_age` to set a maximum cache age regardless of server headers: