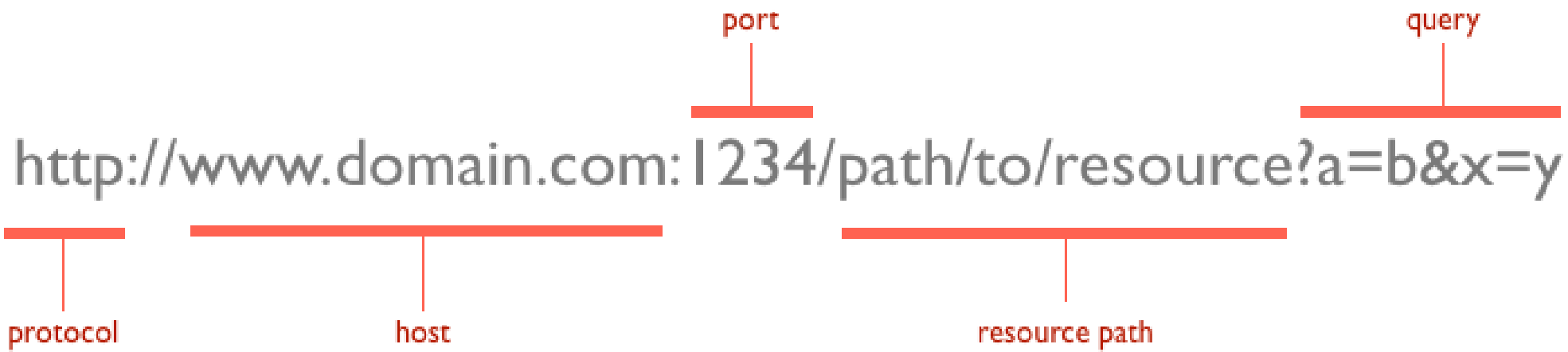
Web APIs

Lecture 13

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 parameter(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://lmgtfy.com/?q=hello%20world`
- `http://maps.googleapis.com/maps/api/geocode/json?sensor=false&address=1600+Amphitheatre+Parkway`
- `https://nomnom-prod-api.dennys.com/mapbox/geocoding/v5/mapbox.places/raleigh,%20nc.json?types=country,region,postcode,place&country=us,pr,vi,gu,mp,ca`

URL encoding

This is 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 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

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

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

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

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

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

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

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

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

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

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

```
URLencode("Σ")
```

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

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

```
[1] "Σ"
```

RESTful APIs

REST

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

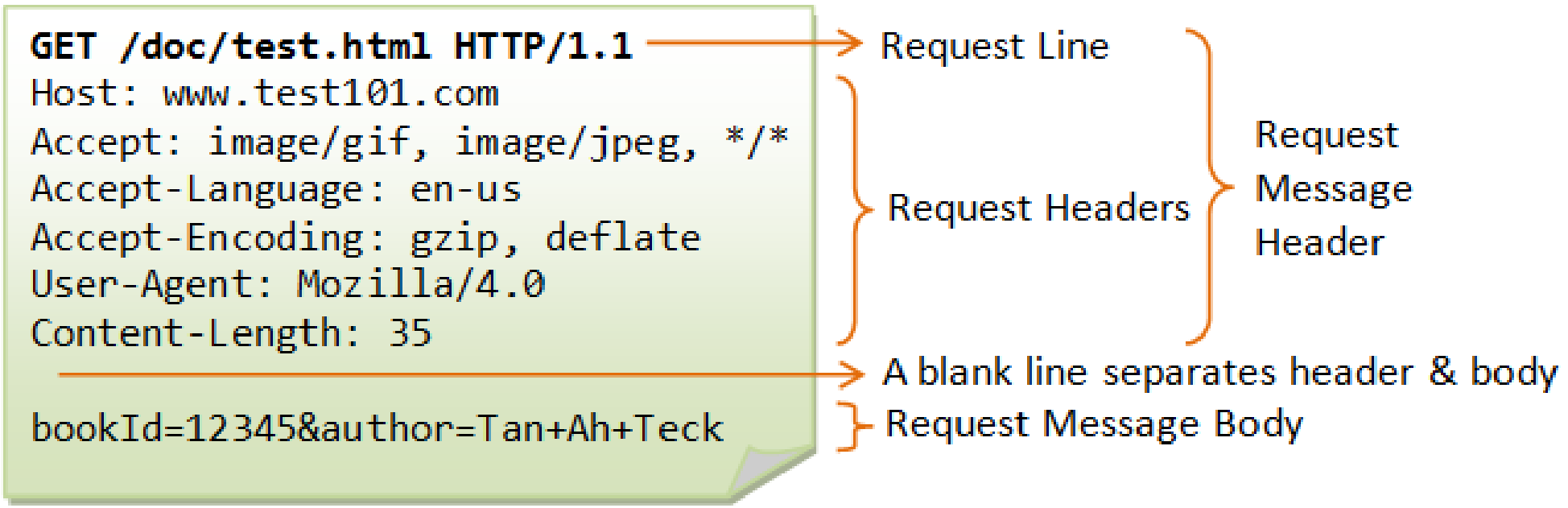
- describes an architectural style for web services (not a standard)
- all communication via HTTP requests
- Key features:
  - client–server architecture
  - addressible (specific URL endpoints)
  - stateless (no client information stored between requests)
  - layered / hierarchical
  - cacheability

HTTP Methods / Verbs

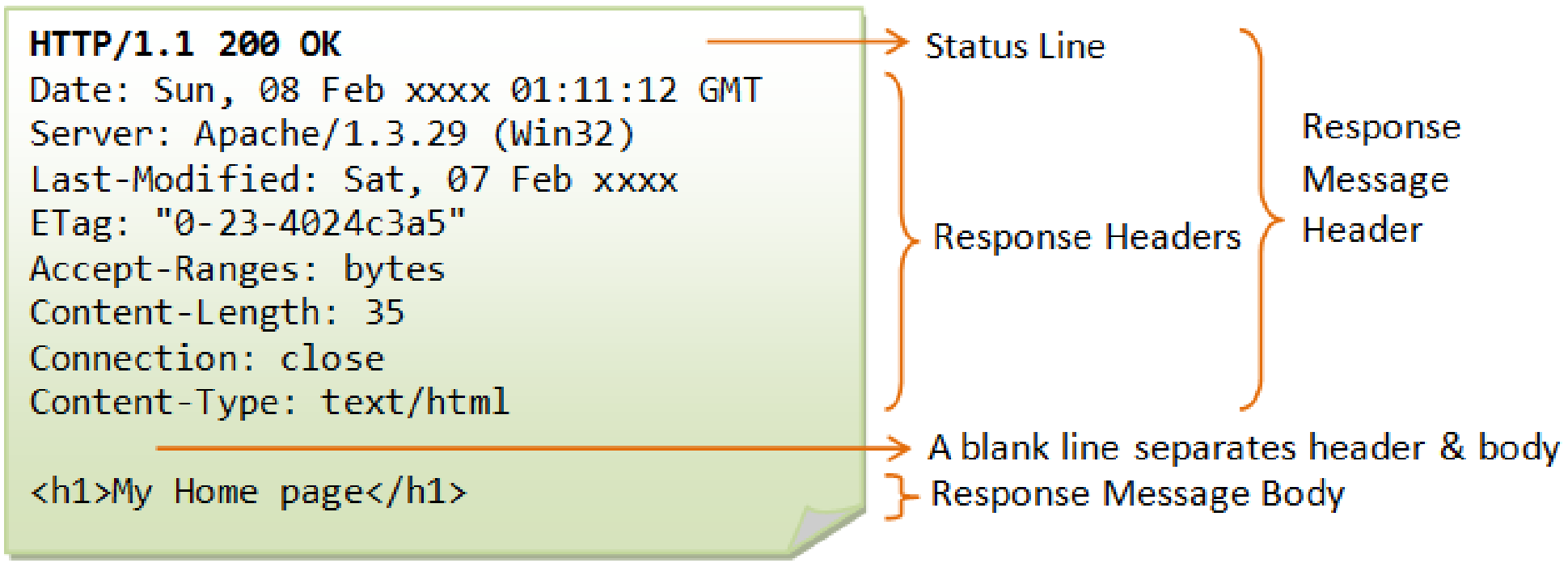
- *GET* - fetch a resource
- *POST* - create a new resource
- *PUT* - full update of a resource
- *PATCH* - partial update of a resource
- *DELETE* - delete a resource.

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

Structure of an HTTP Request



Structure of an HTTP Response



Status Codes

- 1xx: Informational Messages
- 2xx: Successful
- 3xx: Redirection
- 4xx: Client Error
- 5xx: Server Error

Example 1:
An API of Ice And Fire

Documentation

While there is a lot of standardization, every API is different and you will need to review the documentation of each.

See documentation here for AAOIF.

Resources / endpoints:

- Root - `https://www.anapioficeandfire.com/api`
- List books - `https://www.anapioficeandfire.com/api/books`
- Specific book - `https://www.anapioficeandfire.com/api/books/1`
- ...

Pagination

> An API of Ice And Fire provides a lot of data about the world of Westeros. To prevent our servers from getting cranky, the API will automatically paginate the responses. You will learn how to create requests with pagination parameters and consume the response.
>
> ***Things worth noting***
> Information about the pagination is included in the Link header Page numbering is 1-based You can specify how many items you want to receive per page, the maximum is 50
>
> ***Constructing a request with pagination***
> You specify which page you want to access with the `?page` parameter, if you don't provide the `?page` parameter the first page will be returned. You can also specify the size of the page with the `?pageSize` parameter, if you don't provide the `?pageSize` parameter the default size of 10 will be used.

Demo 1 - Basic access & pagination

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
  - Secrure secret storage
- Explicit response object, with support for
  - error codes / reporting
  - common body encoding (e.g. json, etc.)

request objects

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

Some useful `req_*()` functions:

- `req_method()` - set HTTP method
- `req_url_query()` - add query parameters to URL
- `req_url_*()` - add or modify URL
- `req_body_*()` - set body content (various formats and sources)
- `req_user_agent()` - set user-agent
- `req_dry_run()` - shows the exact request that will be made

response objects

A request is made via `req_perform()` which then returns a response object (the most recent response can also be retrieved via `last_response()`). Content of the response are accessed via the `resp_*()` functions

Some useful `resp_*()` functions:

- `resp_status()` - extract HTTP status code (`resp_status_desc()` for a text description)
- `resp_content_type()` - extract content type and encoding
- `resp_body_*()` - extract body from a specific format (json, html, xml, etc.)
- `resp_headers()` - extract response headers

Example 2 - rottentomatoes.com

```
read_html("https://www.rottentomatoes.com")
```

```
{html_document}
<html lang="en" dir="ltr" xmlns="http://www.w3.org/1999/xhtml" prefix="fb:
http://www.facebook.com/2008/fbml og: http://opengraphprotocol.org/schema/">
[1] <head prefix="og: http://ogp.me/ns# flixstertomatoes: http://ogp.me/ns/ap ...
[2] <body class="body no-touch js-mptd-layout">\n        <user-activity-manag ...
```

```
library(httr2)
req = request("https://www.rottentomatoes.com")
```

```
req
```

```
<httr2_request>
GET https://www.rottentomatoes.com
Body: empty
```

```
req |> req_user_agent()
```

```
<httr2_request>
GET https://www.rottentomatoes.com
Body: empty
Options:
• useragent: 'httr2/1.0.0 r-curl/5.2.0 libcurl/8.4.0'
```

Response

```
(req_good = req |>
  req_user_agent())
```

```
<httr2_request>
GET https://www.rottentomatoes.com
Body: empty
Options:
• useragent: 'httr2/1.0.0 r-curl/5.2.0
libcurl/8.4.0'
```

```
(req_bad = req |>
  req_user_agent(options()$HTTPUserAgent
```

```
<httr2_request>
GET https://www.rottentomatoes.com
Body: empty
Options:
• useragent: 'RStudio Desktop (2023.9.0.463); R
(4.3.1
aarch64-apple-darwin23.0.0 aarch64 darwin23.0.0)'
```

```
(res_good = req_good |> req_perform())
```

```
<httr2_response>
GET https://www.rottentomatoes.com/
Status: 200 OK
Content-Type: text/html
Body: In memory (368742 bytes)
```

```
req_bad |> req_perform()
```

```
Error in `req_perform()`:
! HTTP 403 Forbidden.
```

Response body

```
1 res_good |> resp_body_html()
```

```
{html_document}
<html lang="en" dir="ltr"
xmlns="http://www.w3.org/1999/xhtml" prefix="fb:
http://www.facebook.com/2008/fbml og:
http://opengraphprotocol.org/schema/">
[1] <head prefix="og: http://ogp.me/ns#
flixstertomatoes: http://ogp.me/ns/ap ...
[2] <body class="body no-touch js-mptd-layout">\n
<user-activity-manag ...
```

```
1 res_good |> resp_body_string()
```

```
[1] "<!DOCTYPE html>\n<html lang=\"en\" dir=\"ltr\"
xmlns=\"http://www.w3.org/1999/xhtml\" prefix=\"fb:
http://www.facebook.com/2008/fbml og:
http://opengraphprotocol.org/schema/\">\n    <head
prefix=\"og: http://ogp.me/ns# flixstertomatoes:
http://ogp.me/ns/apps/flixstertomatoes#\">\n
\n        \n            <script\n
charset=\"UTF-8\"\n
crossorigin=\"anonymous\"\n                data-
domain-script=\"7e979733-6841-4fce-9182-
515fac69187f\"\n                integrity=\"sha384-
TKdmlzVmoD70HzftTw4WtOzIBL5mNx8mXSRzEvwrWjpIJ7FZ/EuX
src=\"https://cdn.cookielaw.org/consent/7e979733-
6841-4fce-9182-515fac69187f/otSDKStub.js\"\n
type=\"text/javascript\"\n            >\n
</script>\n            <script
type=\"text/javascript\">\n                function
```

Demo 2 - httr2 + headers

```r
aaoif = function(
  resource = c("root", "books", "characters", "houses"), ...,
  base_url = "https://www.anapioficeandfire.com/api/", verbose = TRUE
) {
  resource = match.arg(resource)

  get_links = function(resp) {
    resp |>
      resp_header("link") |>
      str_match_all('<(.*?)>; rel="([a-zA-Z]+)"') |>
      (\(x) (setNames(as.list(x[[1]][,2]), x[[1]][,3])))()
  }

  if (resource == "root")
    resource = ""

  resp = request(base_url) |>
    req_url_path_append(resource) |>
    req_url_query(...) |>
    req_perform()

  full = list()
  page = 1
```

Exercise 1

Using the AAOIF API answer the following questions:

1. How many characters are included in this API?
2. What percentage of the characters are dead?
3. How many houses have an ancestral weapon?

05:00

Demo 3 - GitHub API

GitHub API(s)

GitHub provides two APIs for accessing the website and its data:

- A REST api - Getting started
- A GraphQL api - About the GraphQL API

The REST api is more mature and provides access / interact with most of the data available on the website. The GraphQL api is more flexible and efficient, but is still under development and does not provide access to all the data available on the website.

To do almost anything useful with either API you will need to authenticate. This can be done via a personal access token (PAT) which is then passed as part of the http request header.