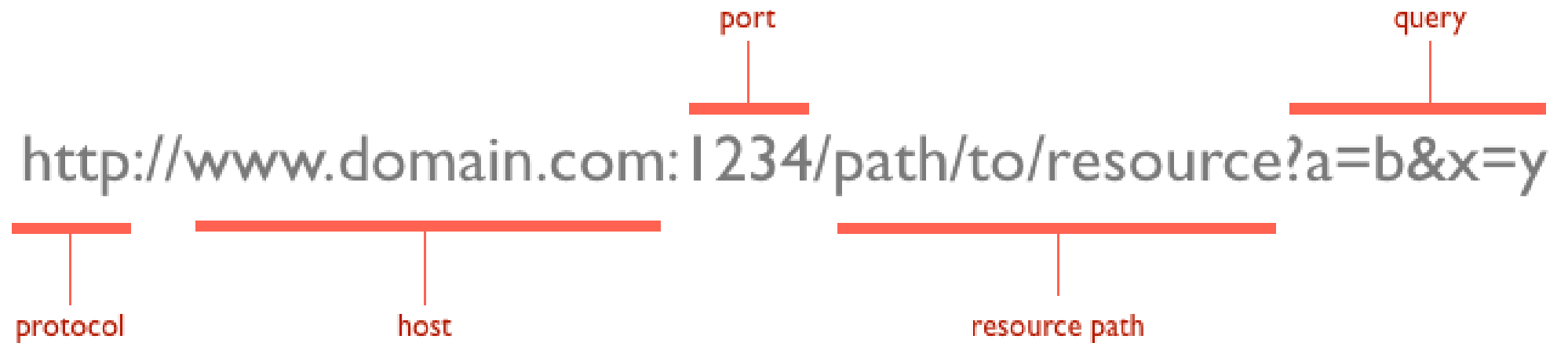


# Web APIs

## Lecture 13

Dr. Colin Rundel

# URLs



# 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://lmgty.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  $\Sigma$  would be encoded as `&#931;` )

```
1 urlencode("http://lmgty.com/?q=hello world")
```

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

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

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

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

```
[1] "!%20#%20$%20&%20'%20(%20)%20*%20+%20,%20/%20:%20;%20=%20?%20@%20[%20]"
```

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

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

```
1 urlencode("Σ")
```

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

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

```
[1] "Σ"
```

# RESTful APIs

# REST

## *RE*presentational State Transfer

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

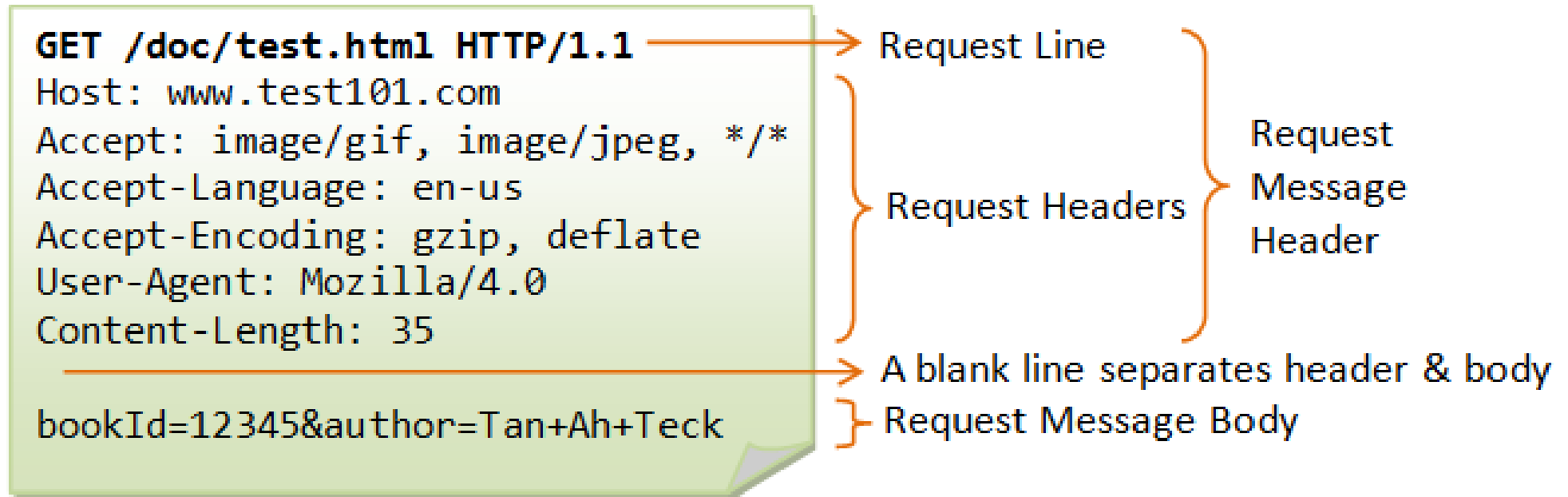
# HTTP Methods / Verbs

- *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.

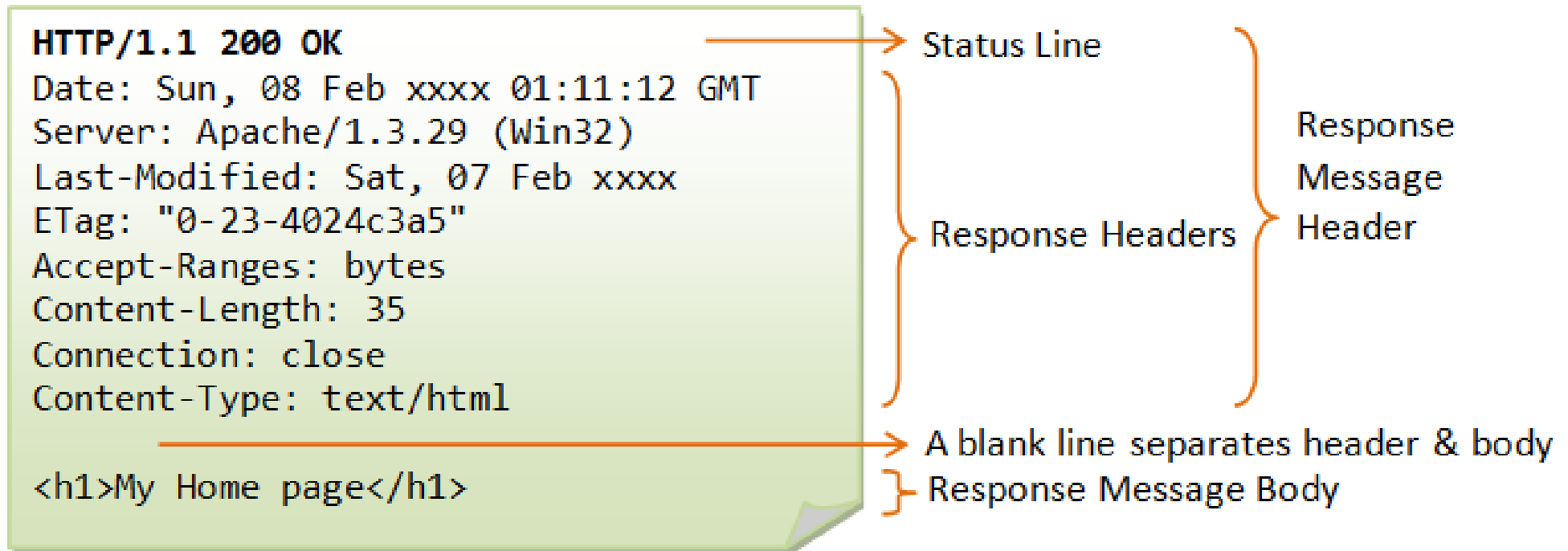
Less common verbs: *HEAD*, *TRACE*, *OPTIONS*.



# 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.anapiofficeandfire.com/api>
- List books - <https://www.anapiofficeandfire.com/api/books>
- Specific book - <https://www.anapiofficeandfire.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

`http2` is a package designed around the construction and handling of HTTP requests and responses. It is a rewrite of the `http` 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.)

# request objects

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

# Demo 2 - httr2 + headers

# 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?

# Demo 3 - GitHub API