COMP30680
Web Application Development

HTTP and RESTful apis

David Coyle
d.coyle@ucd.ie

HTTP

“The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.”

w3 and the Internet Society

HTTP has been in use by the World-Wide Web global information initiative since 1990.

Until recently it was specified in a document called **RFC2616**.

This has now been replaced by a series of more managable specification documents:

http://httpwg.org/

https://www.mnot.net/blog/2014/06/07/rfc2616_is_dead

HTTP

"The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, **stateless**, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred."

w3 and the Internet Society

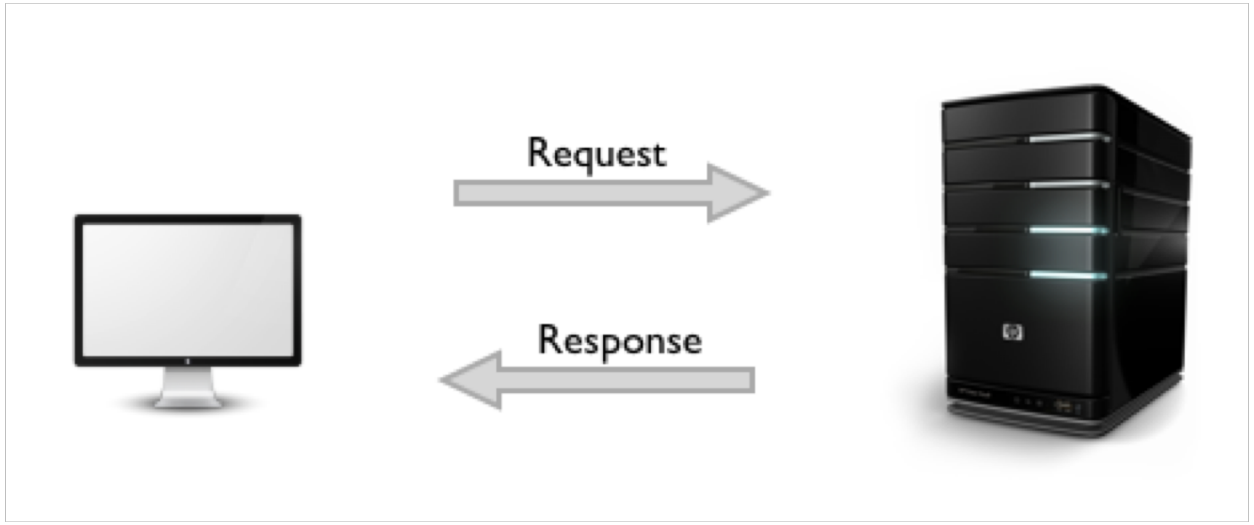
In computing, a **stateless protocol** is a communications protocol that treats each request as an independent transaction that is unrelated to any previous request so that the communication consists of independent pairs of ***request and response***.

- A stateless protocol does not require the server to retain session information or status about each communications partner for the duration of multiple requests. In contrast, a protocol which requires keeping of the internal state on the server is known as a stateful protocol.

HTTP Basics

HTTP allows for communication between a variety of hosts and clients, and supports a mixture of network configurations.

To make this possible, it assumes very little about a particular system, and does not keep state between different message exchanges.



Communication between a host and a client occurs, via a **request/response pair**. The client initiates an HTTP request message, which is serviced through a HTTP response message in return.

URLs

At the heart of web communications is the request message, which are sent via Uniform Resource Locators (URLs).

URLs have a simple structure that consists of the following components:

The protocol is typically **http**, but it can also be **https** for secure communications. The default **port** is 80, but one can be set explicitly, as illustrated in the above image. The resource path is the ***local path*** to the resource on the server.

Verbs

URLs reveal the identity of the particular host with which we want to communicate, but the action that should be performed on the host is specified via HTTP verbs.

HTTP has formalized several request verbs, including:

GET: *fetch* an existing resource. The URL contains all the necessary information the server needs to locate and return the resource.

POST: *create* a new resource. POST requests usually carry a payload that specifies the data for the new resource.

PUT: *update* an existing resource. The payload may contain the updated data for the resource.

DELETE: *delete* an existing resource.

The above four verbs are the most popular, and most tools and frameworks explicitly expose these request verbs. PUT and DELETE are sometimes considered specialized versions of the POST verb, and they may be packaged as POSTrequests with the payload containing the exact action: *create*, *update* or *delete*.

Status Codes

With URLs and verbs, the client can initiate requests to the server.

In return, the server responds with **status codes** and **message** payloads.

The status code is important and tells the client how to interpret the server response. The HTTP spec defines certain number ranges for specific types of responses:

1xx: Informational Messages

2xx: Successful

3xx: Redirection

4xx: Client Error (e.g. 401 Unauthorised, 404 Not Found)

5xx: Server Error

Full list: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

HTTP Status Codes in 60 Seconds: http://code.tutsplus.com/tutorials/http-status-codes-in-60-seconds--cms-24317

HTTP Status Codes

For great REST services the correct usage of the correct HTTP status code in a response is vital.

1xx – Informational

This class of status code indicates a provisional response, consisting only of the Status-Line and optional headers, and is terminated by an empty line

- 100 – Continue
- 101 – Switching Protocols
- 102 – Processing

2xx – Successful

This class of status code indicates that the client's request was successfully received, understood, and accepted.

- 200 – OK
- 201 – Created
- 202 – Accepted
- 203 – Non-Authoritative Information
- 204 – No Content
- 205 – Reset Content
- 206 – Partial Content
- 207 – Multi-Status

3xx – Redirection

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request.

- 300 – Multiple Choices
- 301 – Moved Permanently
- 302 – Found
- 303 – See Other
- 304 – Not Modified
- 305 – Use Proxy
- 307 – Temporary Redirect

4xx – Client Error

The 4xx class of status code is intended for cases in which the client seems to have erred.

- 400 – Bad Request
- 401 – Unauthorised
- 402 – Payment Required
- 403 – Forbidden
- 404 – Not Found
- 405 – Method Not Allowed
- 406 – Not Acceptable
- 407 – Proxy Authentication Required
- 408 – Request Timeout
- 409 – Conflict
- 410 – Gone
- 411 – Length Required
- 412 – Precondition Failed
- 413 – Request Entity Too Large
- 414 – Request URI Too Long
- 415 – Unsupported Media Type
- 416 – Requested Range Not Satisfiable
- 417 – Expectation Failed
- 422 – Unprocessable Entity
- 423 – Locked
- 424 – Failed Dependency
- 425 – Unordered Collection
- 426 – Upgrade Required

5xx – Server Error

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request.

- 500 – Internal Server Error
- 501 – Not Implemented
- 502 – Bad Gateway
- 503 – Service Unavailable
- 504 – Gateway Timeout
- 505 – HTTP Version Not Supported
- 506 – Variant Also Negotiates
- 507 – Insufficient Storage
- 510 – Not Extended

Examples of using HTTP Status Codes in REST

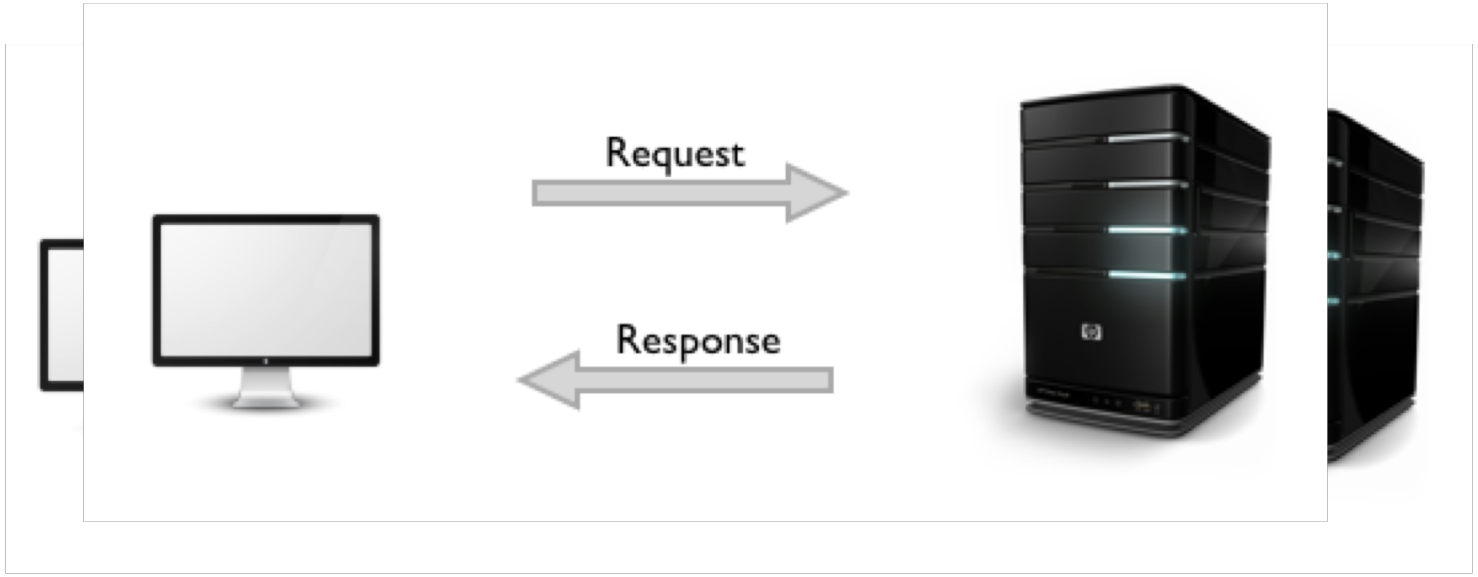
201 – When doing a POST to create a new resource it is best to return 201 and not 200.
204 – When deleting a resources it is best to return 204, which indicates it succeeded but there is no body to return.
301 – If a 301 is returned the client should update any cached URI's to point to the new URI.
302 – This is often used for temporary redirect's, however 303 and 307 are better choices.
409 – This provides a great way to deal with conflicts caused by multiple updates.
501 – This implies that the feature will be implemented in the future.

Special Cases

306 – This status code is no longer used. It used to be for switch proxy.
418 – This status code from RFC 2324. However RFC 2324 was submitted as an April Fools' Joke. The message is *I am a teapot*.

Key	Description
Black	HTTP version 1.0
Blue	HTTP version 1.1
Aqua	Extension RFC 2295
Green	Extension RFC 2518
Yellow	Extension RFC 2774
Orange	Extension RFC 2817
Purple	Extension RFC 3648
Red	Extension RFC 4918

HTTP



URLs, verbs and status codes **make up the fundamental pieces of an HTTP request/response pair**

Not required reading, but useful if you want to understand HTTP in more detail:
HTTP Succinctly: http://code.tutsplus.com/series/http-succinctly--net-33683

See http://www.w3schools.com/xml/dom_http.asp

XMLHttpRequest

The **XMLHttpRequest** object is used to exchange data with a server behind the scenes. It is available to web browser scripting languages such as JavaScript. It is used to send HTTP or HTTPS requests to a web server and load the server response data back into the script.

It allows you to:

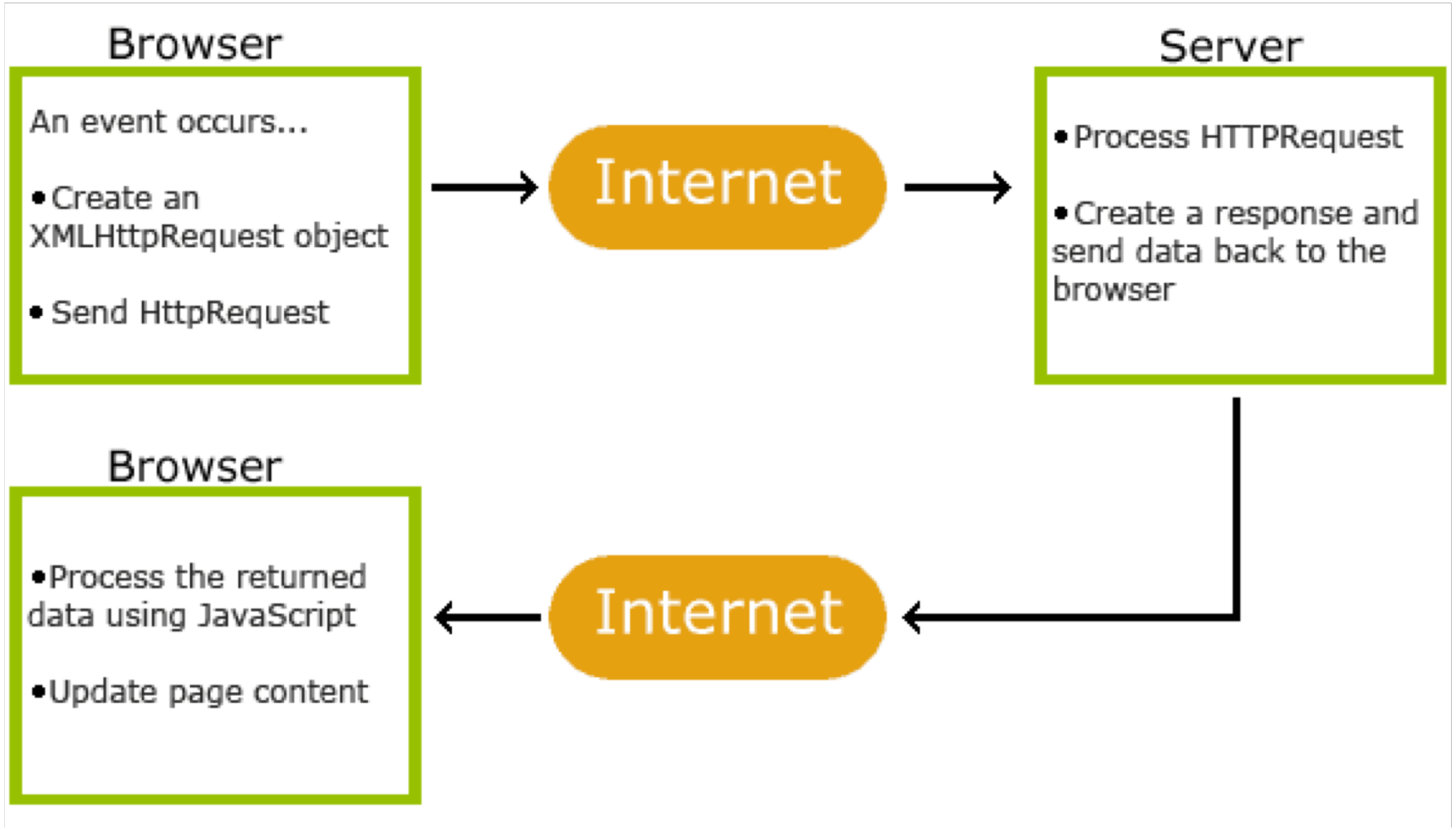
- Update a web page without reloading the page
- Request data from a server after the page has loaded
- Receive data from a server after the page has loaded
- Send data to a server in the background

Data from the response can be used to alter the current document in the browser window without loading a new web page. --> This ability is the basis of development techniques such as **Ajax**.

Despite the name, data can be in the form of not only XML, but also JSON, HTML or plain text.

The response data can also be evaluated by client-side scripting. For example, if it was formatted as JSON by the web server, it can be converted into a client-side data object for further use.

XMLHttpRequest overview



Using XMLHttpRequest

```
 9   var xmlhttp = new XMLHttpRequest();
10   var url = "myTutorials.txt";
11
12   xmlhttp.onreadystatechange = function() {
13       if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
14           var myArr = JSON.parse(xmlhttp.responseText);
15           myFunction(myArr);
16       }
17   };
18
19   xmlhttp.open("GET", url, true);
20   xmlhttp.send();
```

Reading from a local file.

```
var xmlhttp = new XMLHttpRequest();
var url = "http://api.openweathermap.org/data/2.5/forecast?
q=London,us&mode=json&appid=44db6a862fba0b067b1930da0d769e98";

xmlhttp.onreadystatechange=function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
        myFunction(xmlhttp.responseText);
    }
}
xmlhttp.open("GET", url, true);
xmlhttp.send();
```

Reading from a server.

XMLHttpRequest Object Methods

Method	Description
abort()	Cancels the current request
getAllResponseHeaders()	Returns header information
getResponseHeader()	Returns specific header information
open(method,url,async,uname,pswd)	Specifies the type of request, the URL, if the request should be handled asynchronously or not, and other optional attributes of a request method: the type of request: GET or POST url: the location of the file on the server async: true (asynchronous) or false (synchronous)
send(string)	send(string) Sends the request off to the server. string: Only used for POST requests
setRequestHeader()	Adds a label/value pair to the header to be sent

XMLHttpRequest Object Properties

Property	Description
onreadystatechange	Stores a function (or the name of a function) to be called automatically each time the readyState property changes
readyState	Holds the status of the XMLHttpRequest. Changes from 0 to 4: 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
responseText	Returns the response data as a string
responseXML	Returns the response data as XML data
status	Returns the status-number (e.g. "404" for "Not Found" or "200" for "OK")
statusText	Returns the status-text (e.g. "Not Found" or "OK")

See http://rest.elkstein.org/

REST

REST stands for **Re**presentational **S**tate **T**ransfer. (It is sometimes spelled "ReST".)

It relies on a stateless, client-server communications protocol -- and in virtually all cases, the HTTP protocol is used.

The idea is that, rather than using complex mechanisms to connect between machines, simple HTTP is used to make calls between machines.

RESTful applications use HTTP requests to send data (create and/or update), read data (e.g., make queries), and delete data.

- REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

REST

As a programming approach, REST is a lightweight alternative to Web Services and Remote Procedure Calls (RPC).

A REST service is:

- Platform-independent (you don't care if the server is Unix, the client is a Mac, or anything else),
- Language-independent (C# can talk to JavaScript, etc.),
- Standards-based (runs on top of HTTP), and
- Can easily be used in the presence of firewalls.

REST (like web services) offers no built-in security features, encryption, session management, QoS guarantees, etc. But as with Web Services, these can be added by building on top of HTTP:

- For security, username/password tokens are often used.
- For encryption, REST can be used on top of HTTPS (secure sockets).... etc.

How simple is REST?

This example queries a phonebook application for the details of a given user. All we have is the user's ID. Using Web Services and SOAP, the request would look something like this:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
 <soap:body pb="http://www.acme.com/phonebook">
  <pb:GetUserDetails>
   <pb:UserID>12345</pb:UserID>
  </pb:GetUserDetails>
 </soap:Body>
</soap:Envelope>
```

The entire request is then sent (using an HTTP POST request) to the server. The result is probably an XML file, but it will be embedded, as the "payload", inside a SOAP response envelope.

How simple is REST?

And with REST? The query will probably look like this:

```
http://www.acme.com/phonebook/UserDetails/12345
```

It's just a URL. This URL is sent to the server using a simpler GET request, and the HTTP reply is the raw result data -- not embedded inside anything, just the data you need in a way you can directly use, e.g. JSON data.

REST can easily handle more complex requests, including multiple parameters. In most cases, you'll just use HTTP GET parameters in the URL.

```
http://www.acme.com/phonebook/UserDetails?firstName=John&lastName=Doe
```

As a rule, GET requests should be for read-only queries; they should not change the state of the server and its data. For creation, updating, and deleting data, use POST requests.

How simple is REST?

And with REST? The query will probably look like this:

```
http://www.acme.com/phonebook/UserDetails/12345
```

It's just a URL. This URL is sent to the server using a simple GET request, and the HTTP reply is the raw result data -- not er y use, e.g. JSON data.

REST can easily han s, you'll just use HTTP GET para

```
http://www.acme.com/phonebook/UserDetails?firstName=John&lastName=Doe
```

As a rule, GET requests should be for read-only queries; they should not change the state of the server and its data. For creation, updating, and deleting data, use POST requests.

Some REST examples

Twitter: https://developer.twitter.com/en/docs/api-reference-index

Flickr: http://www.flickr.com/services/api/

NY Times: http://developer.nytimes.com/docs/read/article_search_api_v2

Google Maps Geocoding api: https://developers.google.com/maps/documentation/geocoding/intro

OpenWeatherMap: http://openweathermap.org/api

A large directory, updated regularly: http://www.programmableweb.com/apis/directory

A nice introductory video: https://www.youtube.com/watch?v=7YcW25PHnAA

Current weather data

Access current weather data for any location on Earth including over 200,000 cities! Current weather is frequently updated based on global models and data from more than 40,000 weather stations. Data is available in JSON, XML, or HTML format.

http://openweathermap.org/api

Example requests for current weather:

By city name:
api.openweathermap.org/data/2.5/weather?q={city name}
http://api.openweathermap.org/data/2.5/weather?q=Dublin

By geographic coordinates
api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}
http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139

Google Maps Geocoding API

GUIDES SUPPORT SEND FEEDBACK

Developer's Guide
Get a Key
Usage Limits

Google Maps Web Services
Overview
Client Library

The Google Maps Geocoding API

★ This service is also available as part of the Google Maps JavaScript API, or the Java and Python client libraries.

Contents

https://developers.google.com/maps/documentation/geocoding/intro

A Google Maps Geocoding API request must be of the following form:

```
https://maps.googleapis.com/maps/api/geocode/output?parameters
```

Example requests location information for Dublin:
https://maps.googleapis.com/maps/api/geocode/json?address=Dublin

Questions, Suggestions?

Next:

JSON examples