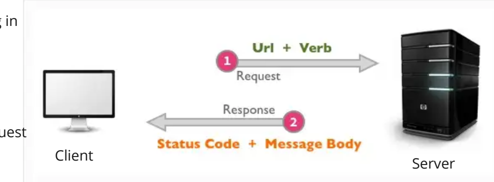
HTTP Verbs
Status codes
Routes
Headers
Body
Request/Response cycle

What are the steps from typing in https://bbc.co.uk into the web browser to getting content on your screen?

1. Type in url
2. Browser builds a HTTP Request
3. Request sent to the server
4. Server reads request
5. Server generates response
6. Response sent to browser
7. Browser reads response
8. Information displayed on screen



URL to IP conversion

TLS handshake

What is the url and the verb in the request?

What is the status code and body in the response?

https://http.cats

Use the networking tab to explore headers

- Python support for HTTP
- Starting the HTTP server
- Defining an HTTP request handler class
- Servicing HTTP requests
- Running the HTTP server
- Dynamic content example
- Static content example

Python provides a set of APIs that enable you to implement an HTTP Web server in Python
- Using classes in the http.server module

Here's the big picture:
- Create an HTTPServer object to listen on a particular port
- Define a subclass of BaseHTTPRequestHandler, to handle incoming requests from clients
- Start the server

The complete example of how to do this is in the code folder, see 1-HttpServer/webserver.py

```
from os import curdir, sep
from http.server import BaseHTTPRequestHandler,
    HTTPServer
import mimetypes
…

def main():
    try:
        server = HTTPServer(('', 8001), MyHandler)
        print('Started HTTP server...')
        server.serve_forever()

    except KeyboardInterrupt:
        print('Ctrl+C received, shutting down server')
        server.socket.close()

if __name__ == '__main__':
    main()
```

This code shows how to create an HTTP server in Python. Note the following points:
- At the bottom of the code, we test the name of this module. If the name is __main__, it means this is the top-level module in the application. In this case, we call a helper function named main() to bootstrap the HTTP server.
- The main() function creates an HTTPServer object. As mentioned previously, HTTPServer is defined in the http.server module, so we've imported it at the top of the code.
- When we create the HTTPServer object, we specify the port number it'll listen on, plus the name of our custom handler class MyHandler. We'll describe MyHandler in detail in the next few slides.
- When we're ready, we invoke serve_forever() to start the HTTP server and to keep it running continuously.
- If the user hits Ctrl+C, we call close() on the HTTP server's socket, to stop listening.

```
class MyHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        …

    def do_POST(self):
        …
```

To define an HTTP request handler class, to handle incoming requests from the client:
- Define a class that inherits from BaseHTTPRequestHandler
- Implement do_GET() if you want to handle HTTP GET requests
- Implement do_POST() if you want to handle HTTP POST requests

```
    def do_GET(self):
        …

        if self.path.endswith(".zzz"):   # Our made-up
            dynamic content.

            self.send_response(200)
            self.send_header('Content-type', 'text/html')
            self.end_headers()

            result = "You requested {0} on day {1} in {2}"
                \
                        .format(self.path,
                                time.localtime()[7],
                                time.localtime()[0])

            self.wfile.write(result.encode('utf-8'))

        else:
            f = open(curdir + sep + self.path)
            self.send_response(200)
            mimeType = mimetypes.guess_type(self.path)[0]
            self.send_header('Content-type', mimeType)
            self.end_headers()
            self.wfile.write(f.read().encode('utf-8'))
            f.close()
```

Here we handle HTTP GET requests.
- First, we test the path requested by the user. We've implemented a special rule in our web server: if the URL ends in .zzz, we generate the content dynamically. Specifically, we return a web page that echoes the name of the path requested, plus information about the date. In doing all this, we use quite a few capabilities of the BaseHTTPRequestHandler base class:
  - send_response() sends an HTTP status code back to the browser (code 200 means "OK").
  - send_header() sends an HTTP header back to the browser (we're telling the browser that we'll be returning HTML content).
  - end_headers() tells the browser we won't be sending any more HTTP headers. Therefore, what comes next will be the actual HTTP body content.
  - wfile.write() writes content to the HTTP body. We encode this text in UTF-8 encoding.
- If the user requested anything other than a .zzz resource, we assume the path is a real file name. In this case, we use the Python File API to open the file (as specified in the request URL), and copy the file contents to the HTTP response body (note we also send an HTTP 200 status code and an appropriate CONTENT-TYPE header too).
- If any errors occur, we send a 404 status code. An alternative status code might be 500, to indicate a general server error. This is handled in the try/except that wraps this conditional.

- The name "Rest"
- What is a Rest service?
- HTTP verbs
- HTTP response codes
- Key principles of Rest services
- Implementing a Rest service in Python
- Calling a Rest service in Python

The name "Representational State Transfer" is intended to evoke an image of how a well-designed Web application behaves: a network of Web pages forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user.

Fielding & Taylor 2002

Rest services are resource-centric services
- Endpoints (URIs) represent resources
- Endpoints are accessible via standard HTTPS
- Endpoints can be represented in a variety of formats (e.g. XML, JSON, HTML, plain text)

They use the same verbs and response codes that we use for all other servers.

Rest services are based on standard technologies
- HTTP, URIs, XML, JSON, etc.
- But not SOAP!

HTTP verbs specify CRUD operations
- POST, GET, PUT, DELETE

Focus on resources
- Resource-centric vs. API-centric
- Resources are identified using URIs (name everything)
- Resources are connected through links (reveal gradually)
- Resources may have different representations (XML, JSON, (X)HTML, plain text, ATOM, etc.)

Ex2 Server

Install two packages:
- flask
- flask_restful

Ex3 Client

Install one package:
- requests

- Web sockets to the rescue
- How Web sockets work
- Introducing the Python Web sockets API
- Implementing a Web sockets server
- Implementing a Web sockets client
- Running the server and client(s)

Traditionally, when a browser visits a web page:
- An HTTP request is sent to the web server that hosts that page
- The web server acknowledges this request and sends back the response

In some cases, the response could be stale by the time the browser renders the page
- E.g. stock prices, news reports, ticket sales, etc.

How can you ensure you get up-to-date information?
- Polling
- Long polling

Web sockets are a powerful communication feature in the HTML5 specification

Web sockets defines a full-duplex communication channel between browser and server
- Simultaneous 2-way data exchange between browser and server
- A large advance in HTTP capabilities
- Extremely useful for real-time, event-driven Web applications

To support real-time full-duplex communication between a client and server:
- The client and server upgrade from the HTTP protocol to the Web sockets protocol during their initial handshake

Thereafter, client and the server can communicate in full-duplex mode over the open connection
- Allows the server to push information to the client, when the data becomes available
- Allows the client and server to communicate simultaneously

You can define a Web sockets server in Python code
- Via the websockets standard module

You must implement the server to support asynchronous calls from multiple clients
- So you'll need the asyncio standard module too

The full code is in 3-WebSockets.

```
import asyncio
import websockets

async def onconnect(websocket, uri):

    while True:
        datain = await websocket.recv()
        print("From client: %s" % datain)

        dataout = "ECHO! " + datain
        print("To client:   %s" % dataout)

        await websocket.send(dataout)

start_server = websockets.serve(onconnect, 'localhost', 8002)

async def start_server():
    async with websockets.serve(onconnect, 'localhost', 8002):
        await asyncio.Future()

asyncio.run(start_server())
```

Let's discuss the code in the slide. We'll begin with the onconnect() function:
- Perhaps the first thing that strikes you is the async keyword on the onconnect() function. This annotation means the function will contain asynchronous portions of code - i.e. the function will occasionally kick off tasks that will run in the background, and the function will pause until these tasks are complete.
- The onconnect() function will be invoked when a client establishes a Web sockets connection to the server (more on this shortly). The function receives a websocket parameter, which is a live connection back to the client. The function uses this websocket to communicate with to the client.
- Inside the onconnect() function, we call websocket.recv() to wait to receive data from the client. The await keyword is due to asynchronous execution. If and when the client does decide to send us some data, we scoop up the data and put it into the local variable, datain. We do a bit of gentle massaging of the data, and then bounce it back immediately to the client via the websocket.send() call.

Outside of the onconnect() function, note the following points:
- We call the websockets.serve() method to define onconnect as the handler function for Web sockets connections on port 8002 on localhost.
- We return a Future (which is similar to a Promise in JavaScript) - it's declaring there will be some future event that will be processable.
- The asyncio.run() will run forever. It can be good to wrap this in a try/except to gracefully handle a KeyboardInterrupt

```
import asyncio
import websockets

async def client():

    websocket = await websockets.connect('ws://localhost:8002/')

    while True:
        name = input("Enter some data: ")

        print("To server: %s" % name)
        await websocket.send(name)

        resp = await websocket.recv()
        print("From server: %s" % resp)

asyncio.run(client())
```

This code shows a Web sockets client, implemented in Python.

The key point is the websockets.connect() call, which connects to the server on port 8002 on localhost (notice that the URL uses the ws:// protocol, rather than the http:// protocol). We get back a websocket object, which will allow us to communicate with the server.

The client code runs continuously. As soon as the user enters some text, the client sends the data to the Web sockets server by calling send() on the websocket object. We then await a response from the server, by calling recv() on the websocket object.