SERVER-SIDE

WEB SERVICES

HTTP PROTOCOL

MORE HTTP: OTHER FEATURES

HTTP SERVICES

VERSION

v1.3 07 October 2021

v1.2 05 October 2021

v1.1 30 September 2021

v1.0 28 September 2021

ACKNOWLEDGMENTS

THANKS TO:

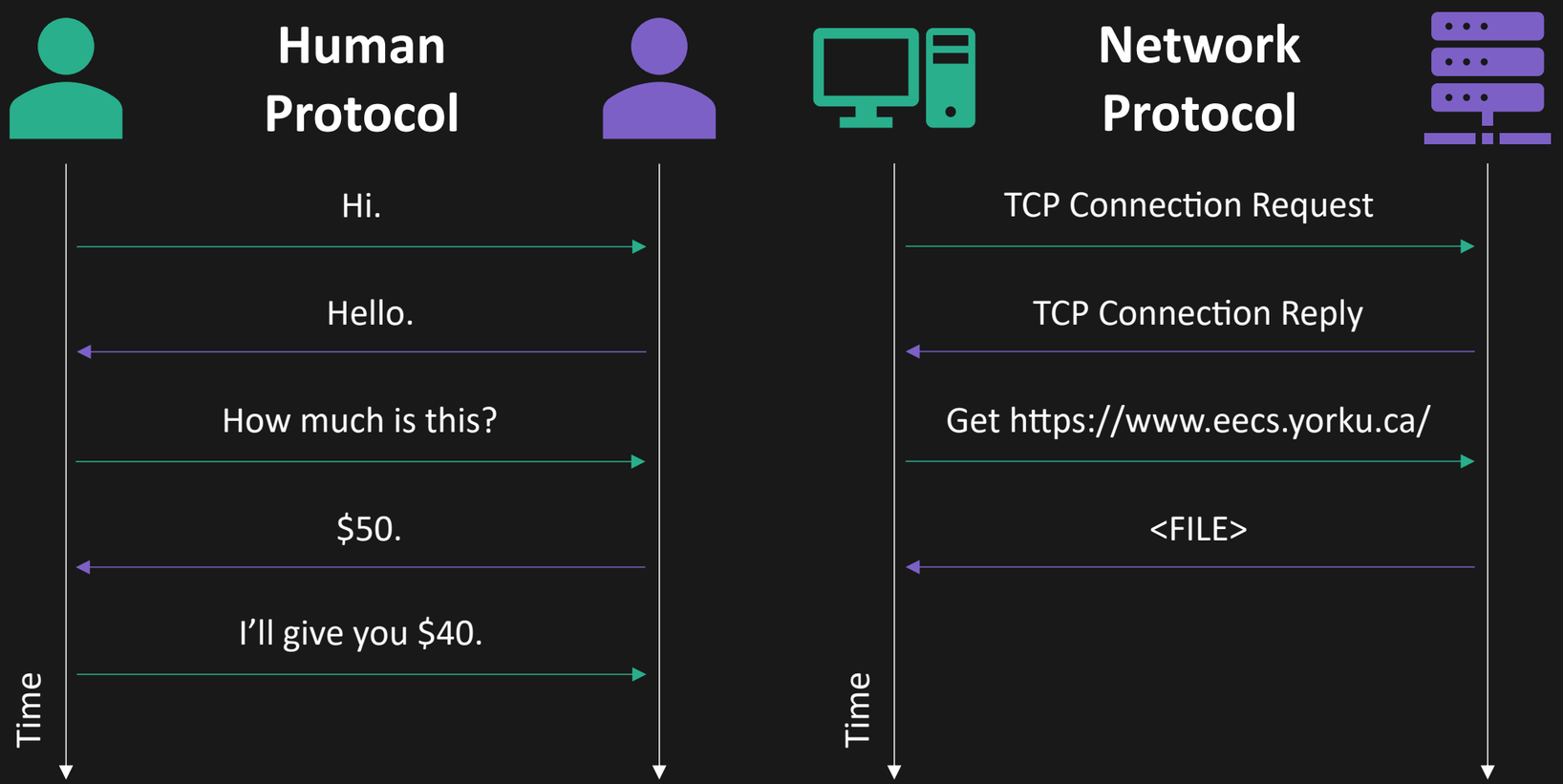
- Hamzeh Roumani, who has shaped EECS-4413 into a leading hands-on CS course at EECS and who generously shared all of his course materials and, more importantly, his teaching philosophy with me;
- Parke Godfrey, my long-suffering Master's supervisor and mentor; and
- Suprakash Datta for giving me this opportunity to teach this course.

PRINTABLE VERSION OF THE TALK

Download PDF

HTTP

THE PROTOCOL

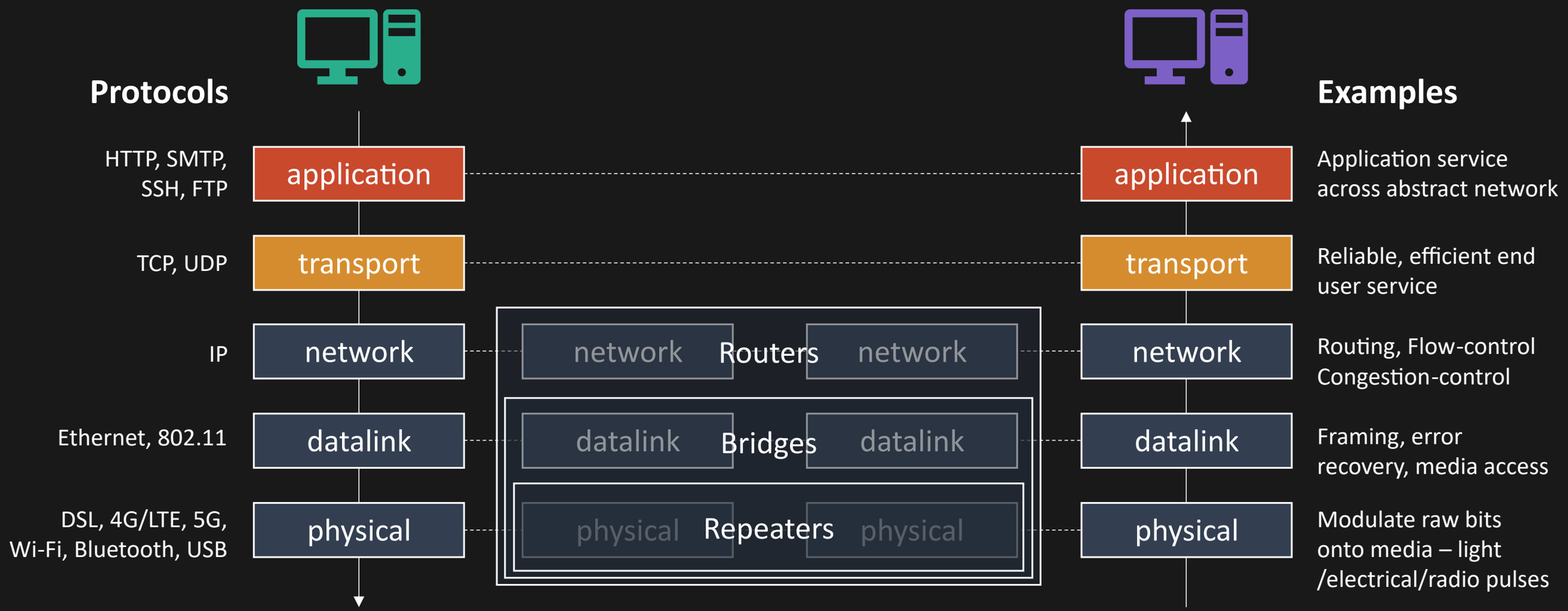


WHAT IS A PROTOCOL?

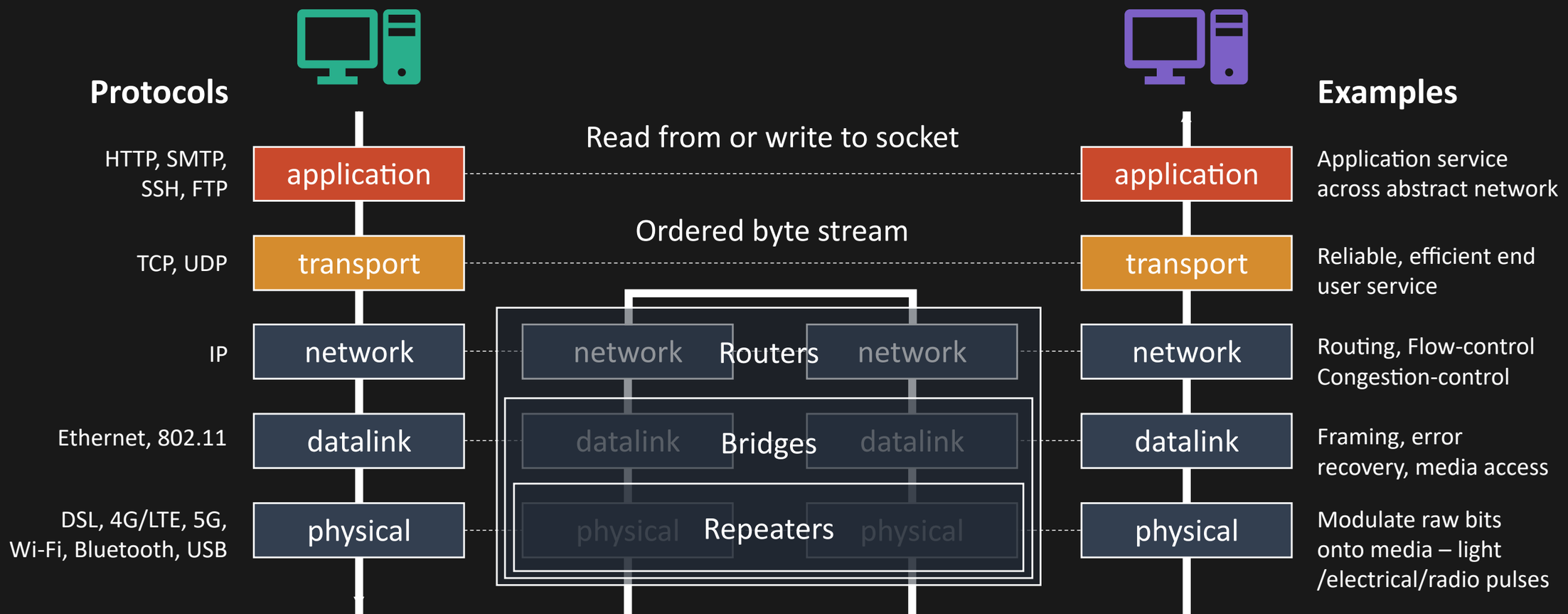
A protocol defines format & order of messages sent and received among network entities, and actions taken on message transmission, receipt.

Source: Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).

THE PROTOCOL STACK



THE PROTOCOL STACK



HYPERTEXT TRANSFER PROTOCOL (HTTP)

The **Hypertext Transfer Protocol (HTTP)** is an application layer protocol for distributed, collaborative, hypermedia information systems, used primarily with the WWW (World Wide Web) in the client-server model where a web browser is a client communicating with the webserver which is hosting the website. Since 1990, this has become the foundation for data communication. HTTP is a standard and stateless protocol that is used for different purposes as well using extensions for request methods, error codes, as well as headers.

Sources: Hypertext Transfer Protocol – HTTP/1.1 (W3C); Introduction to HTTP (w3schools.in); and Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).

- Created by Tim Berners-Lee at CERN (1991)
- Standardized and much expanded by the IETF.
- HTTP/1.0 (RFC 1945), HTTP/1.1 (RFC 2068), HTTP/2 (RFC 7540)
- Other related protocols: HTTPS and WebSocket.
- Rides on top of the TCP protocol, standard on port 80
- TCP provides: reliable, bi-directional, in-order byte stream
- Goal: transfer objects between systems
- Do not confuse with other WWW concepts:
  - HTTP is not page layout language (that is HTML)
  - HTTP is not object naming scheme (that is URLs)

HTTP IN OPERATION

http://www.eecs.yorku.ca/course_archive/2021-22/F/4413/index.html

Time

1a. HTTP client initiates TCP connection to HTTP server (process) at www.eecs.yorku.ca. Port 80 is default for HTTP server.

1b. HTTP server at host www.eecs.yorku.ca waiting for TCP connection at port 80. “accepts” connection, notifying client.

2. HTTP client sends HTTP request message (containing URL) into TCP connection socket.

3. HTTP server receives request message, forms response message containing requested object (/course_archive/2021-22/F/4413/index.html), sends message into socket.

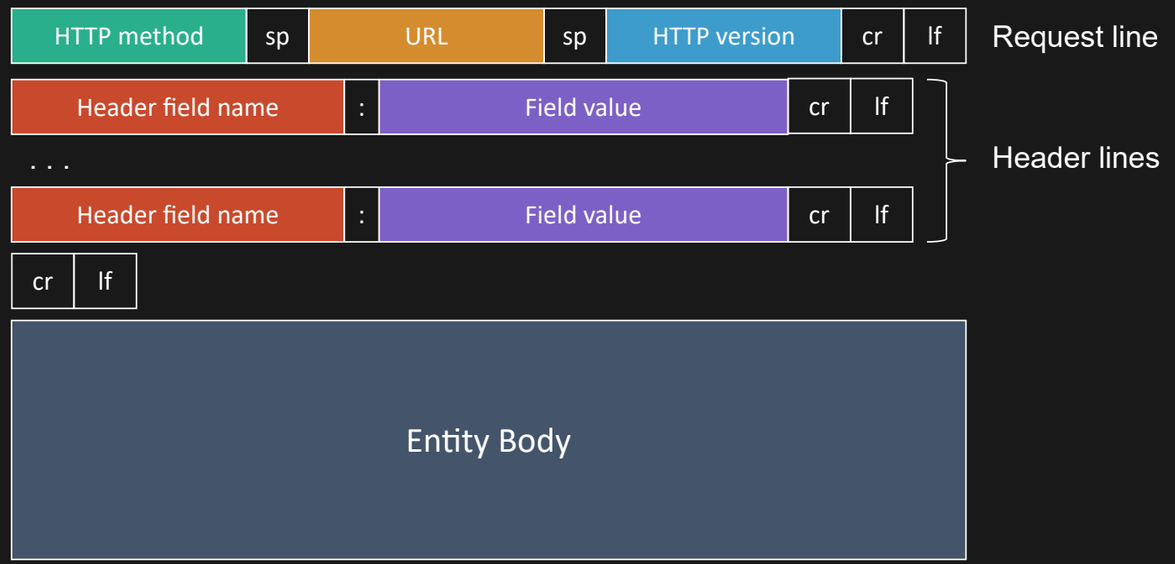
4. HTTP server closes TCP connection.

5. HTTP client receives response message containing HTML file, displays HTML. Parsing HTML file, finds references CSS, JS, and Image files.

6. Steps 1-5 repeated for each separately referenced object.

Sources: Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).

HTTP REQUEST




```
GET /course_archive/2021-22/F/4413/ HTTP/1.1
Host: www.eecs.yorku.ca
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36.
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp.
Accept-Encoding: gzip, deflate, br
Accept-Language: en-CA,en-US;q=0.9,en;q=0.8
Cookie: mayaauth="..."
```

Sources: Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).

HTTP REQUEST METHODS

GET Used to request a particular resource data from the Web server by specifying the parameters as a query string (name and value pairs) in the URL part of the request.

- Are bookmark-able as they appear in the URL.
- Cache-able.
- Saved in the browser history if it is executed using a web browser.
- Character length restrictions (2048 chars maximum).
- Cannot be used to send binary data.
- Data can only be retrieved and have no other effect.
- When communicating sensitive data such as login credentials, should not be used.
- A safe and ideal method to request data only.

HEAD Almost identical to the GET method, but the only difference is that it will not return any response body. The HEAD request becomes useful for testing whether the GET request will actually respond before making the actual GET request.

DELETE Used to delete any specific resource.

TRACE Used for performing a message loop-back, which tests the path for the target resource. It is useful for debugging purposes.

CONNECT Used for establishing a tunnel to the server recognized by a given URI.

POST Used to send data to the Web server in the request body of HTTP.

- Not be bookmark-able as they do not appear in the URL.
- Not cached.
- Not saved as history by the web browsers.
- No restriction on the amount of data to be sent.
- Can be used to send ASCII as well as binary data.
- Use when communicating sensitive data, such as when submitting an HTML form.
- Security depends on the HTTP protocol.
- By using secure HTTP (HTTPS), information is protected.

PUT Requests that the target resource creates or updates its state with the state defined by the representation enclosed in the request. Used to update existing resources with uploaded content or to create a new resource if the target resource is not found. The difference between POST and PUT is that PUT requests are static, which means calling the same PUT method multiple times will not yield a different result.

PATCH Requests that the target resource modifies its state according to the partial update defined in the representation enclosed in the request.

OPTIONS Used for describing the communication preferences for any target resource.

Sources: Introduction to HTTP (w3schools.in).

UNIVERSAL RESOURCE IDENTIFIERS (URI)

URIs are also known as WWW addresses, Uniform Resource Name (URN), and the Uniform Resource Locator (URL). These are formatted, case-insensitive strings that identify a web service, a resource, a website, etc.

```
URI = "http :" "//" host[ ":" port ][ abs_path ["?" query]]
```

- A standard way to send many name/value pairs in a single string (QUERY_STRING or Form data)
- Specified in RFC 2396 'Uniform Resource Identifiers (URI): Generic Syntax'.

RULES OF URL-ENCODING

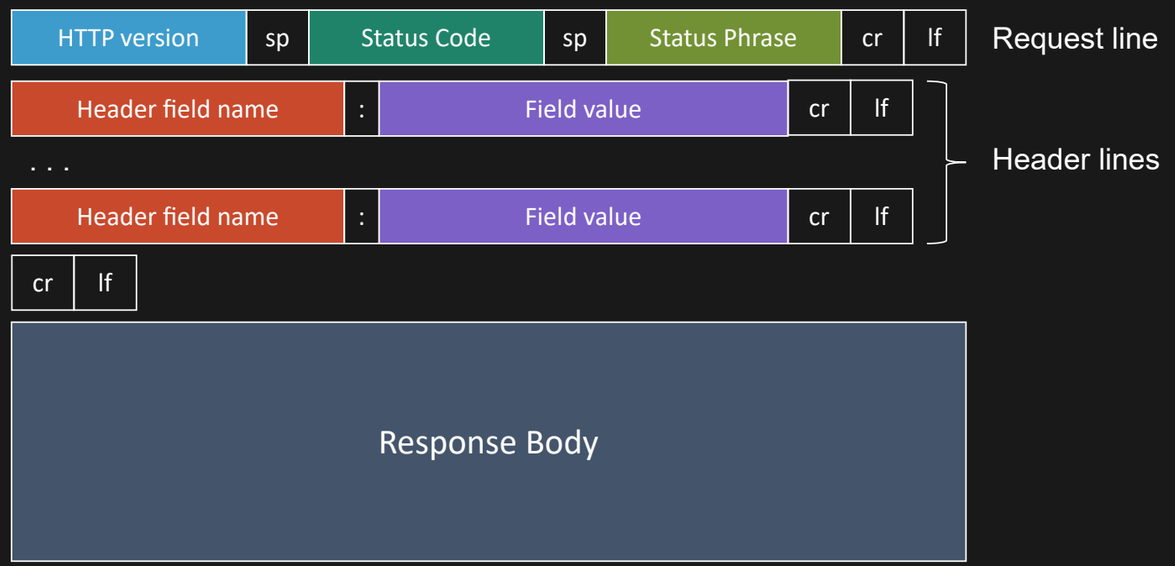
1. All submitted URLs, query strings or form data should be concatenated into single strings of ampersand (&) separated name=value pairs, one pair for each form tag or query parameter. Like this:

   ```
   form_tag_name_1=value_1&form_tag_name_2=value_2&...
   ```

2. Spaces in a name or value are replaced by a plus (+) sign or "%20" (a percent sign followed by 20). This is because url's cannot have spaces in them and under METHOD=GET, the form data is supplied in the query string in the url.
3. Other characters (ie, =, &, +) are replaced by a percent sign (%) followed by the two-digit hexadecimal equivalent of the punctuation character in the Ascii character set.
4. Otherwise, it would be hard to distinguish these characters inside a query or form variable from those between the variables in the first rule above.

Sources: Introduction to HTTP (w3schools.in) and CGI - CSC309F Programming on the Web, David A. Penny (2001)..

HTTP RESPONSE




```
HTTP/1.1 200 OK
Date: Tue, 28 Sep 2021 15:48:11 GMT
Server: Apache/2.4.48 (Unix) PHP/7.4.20 OpenSSL/1.0.2k mod_wsgi/4.7.1 Python/3.8
Strict-Transport-Security: max-age=31536000; includeSubDomains
Last-Modified: Tue, 28 Sep 2021 00:21:45 GMT
Accept-Ranges: bytes
Content-Length: 13305
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

<!DOCTYPE html>
<html lang="en-US">
  <head>
    ...
```

Sources: Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).

HTTP RESPONSE STATUS CODES

1XX	INFORMATIONAL
100	HTTP CONTINUE
101	SWITCHING PROTOCOLS

2XX	SUCCESS
200	OK
201	CREATED
202	ACCEPTED
203	NON AUTHORITATIVE INFORMATION
204	NO CONTENT
205	RESET CONTENT
206	PARTIAL CONTENT

4XX	CLIENT ERROR
400	BAD REQUEST
401	UNAUTHORIZED
402	PAYMENT REQUIRED
403	FORBIDDEN
404	NOT FOUND
405	METHOD NOT ALLOWED
406	NOT ACCEPTABLE
407	PROXY AUTHENTICATION REQUIRED
408	REQUEST TIME OUT
409	CONFLICT
410	GONE
411	LENGTH REQUIRED
412	PRECONDITION FAILED
413	REQUEST ENTITY TOO LARGE
414	REQUEST URI TOO LARGE
415	UNSUPPORTED MEDIA TYPE

3XX	REDIRECTION
300	MULTIPLE CHOICES
301	MOVED PERMANENTLY
302	MOVED TEMPORARILY
303	SEE OTHER
304	NOT MODIFIED
305	USE PROXY

5XX	SERVER ERROR
500	INTERNAL SERVER ERROR
501	NOT IMPLEMENTED
502	BAD GATEWAY
503	SERVICE UNAVAILABLE
504	GATEWAY TIME OUT
505	HTTP VERSION NOT SUPPORTED

TRY OUT HTTP (CLIENT SIDE) FOR YOURSELF

1. Telnet to your favorite Web server, e.g.:

```
$ telnet www.eecs.yorku.ca 80
```

Open TCP connection to port 80 (default http server port) at `www.eecs.yorku.ca`. Anything typed is sent to port 80 at `www.eecs.yorku.ca`.

2. Type in a GET HTTP request:

```
$ GET /course_archive/2021-22/F/4413/index.html HTTP/1.1
```

Type this at the prompt (followed by two carriage returns) to send this minimal GET request to the HTTP server.

3. Look at response message sent by the HTTP server!

MORE HTTP

OTHER FEATURES

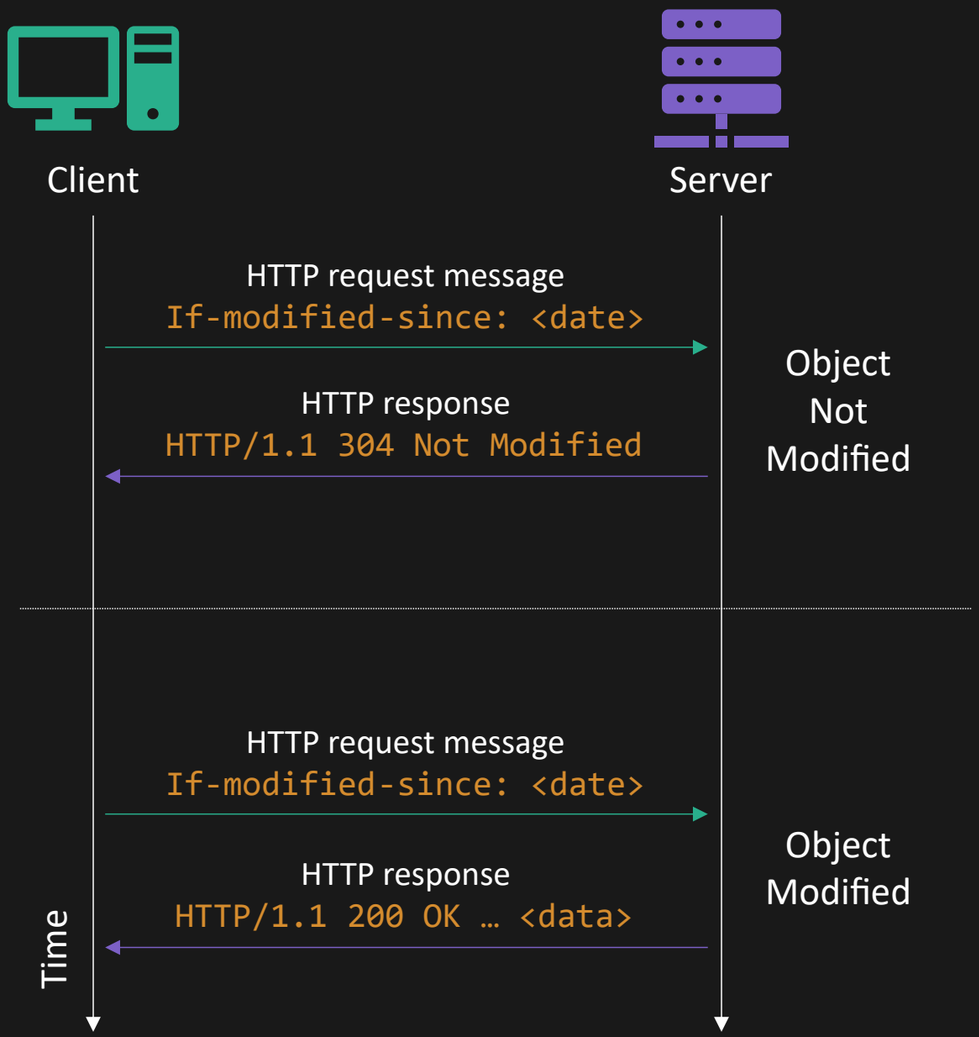
- Conditional GET
- Redirection
- Basic Authentication
- Open Authentication (OAuth)
- Persistence & Cookies
- Keep-Alive (with HTTP/1.1)
- Web Caching

CONDITIONAL GET

IF-MODIFIED-SINCE REQUEST HEADER

The client tells the server it has data and asks the server whether it has a fresher version or the client is up to date.

- Goal: Don't send object if the client has an up-to-date copy (cached).
- Client: Specify the date of cached copy in the HTTP request: `If-modified-since: <date>`.
- Server: Response contains no object if the cached copy is up to date: `HTTP/1.1 304 Not Modified`.



SESSION MANAGEMENT

COOKIES

PROBLEM: HTTP IS STATELESS

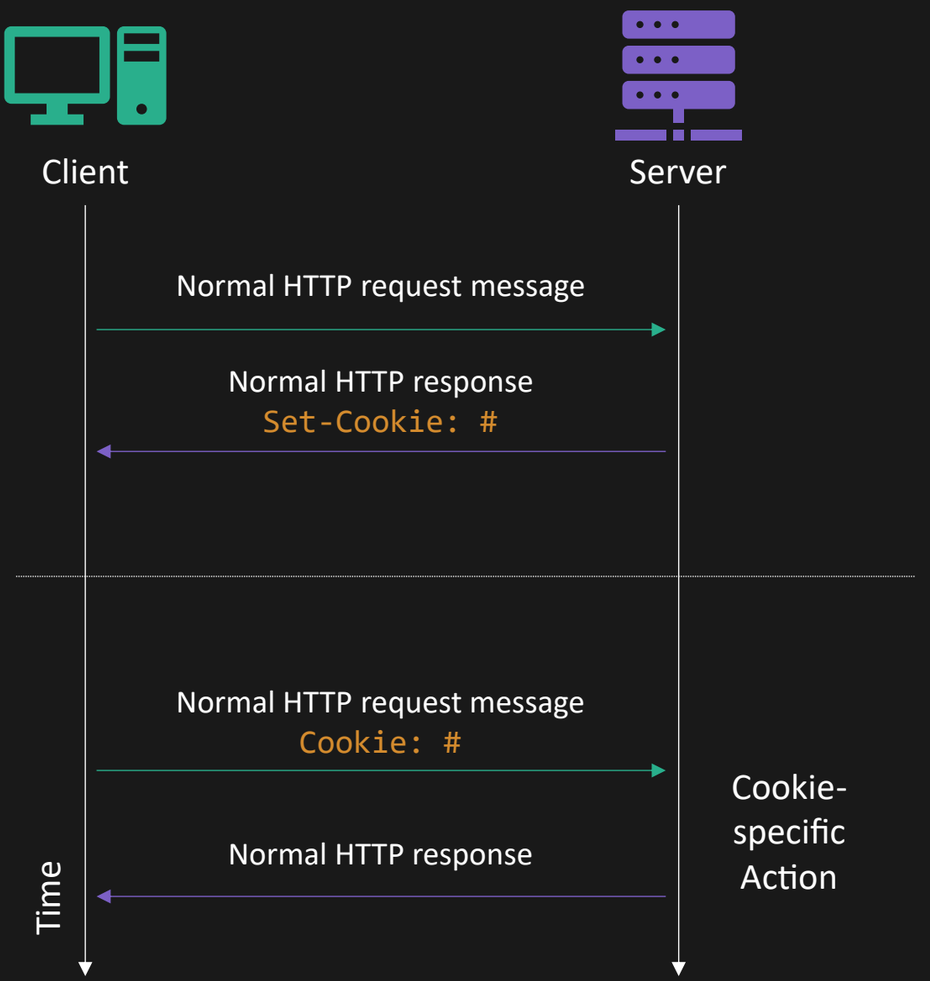
- Server does not maintain status information across client requests
- No way to correlate multiple request from same user

SOLUTION: STORE A COOKIE ON THE CLIENT-SIDE

- Small amount of information (typically server-generated user id)
- Sent by client with each request
- Updated by server with response

CLIENT-SERVER INTERACTION: COOKIES

- Server sends Cookie to client in response message: `Set-Cookie: 1678453`.
- Client presents Cookie in later requests: `Cookie: 1678453`.
- Server matches presented-cookie with server-stored info:
  - Authentication
  - Remembering user preferences, previous choices



Sources: Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).

SESSION MANAGEMENT

SESSIONS

HTTP cannot maintain state (RESTful) but we can:

CLIENT-SIDE

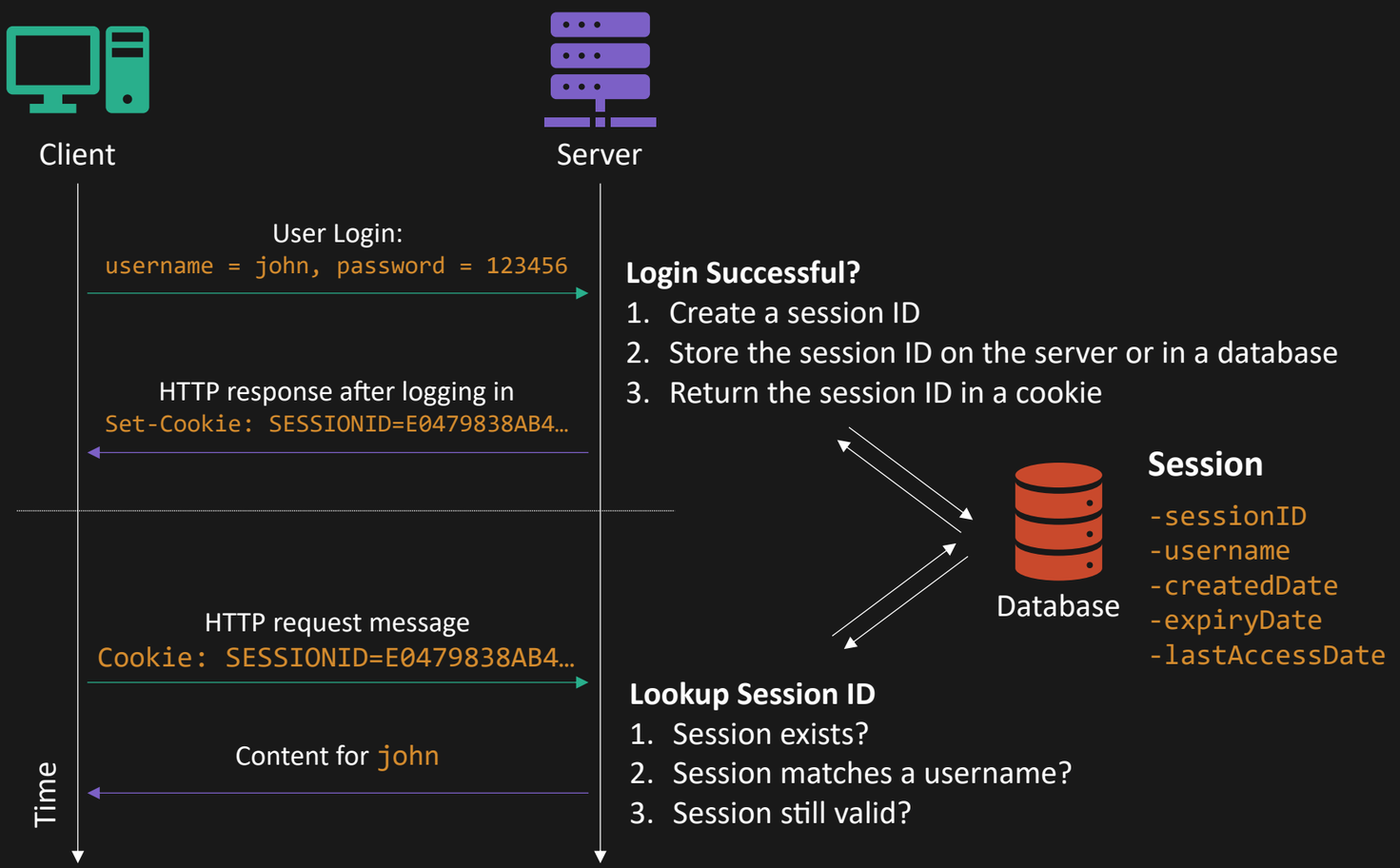
- State maintained by client and sent as needed to server.

NETWORK SIDE

- State shuffled back and forth with every request/response
- Typically through hidden fields, URL Rewriting, or Cookies

SERVER SIDE

- Server keeps it in memory or a database with a key derived from the client's credentials (known though authentication or assigned).
- The key (cookie) is stored in an HTTP header (network side).



SESSION MANAGEMENT

COOKIES VS SESSIONS

COOKIES

A cookie is a small file with the maximum size of 4KB that the web server stores on the client computer. Once a cookie has been set, all page requests that follow return the cookie name and value. A cookie can only be read from the domain that it has been issued from.

SESSIONS

A session is a global variable stored on the server. Each session is assigned a unique id which is used to retrieve stored values. Whenever a session is created, a cookie containing the unique session id is stored on the user's computer and returned with every request to the server. If the client browser does not support cookies, the unique session id is displayed in the URL. Sessions have the capacity to store relatively large data compared to cookies.

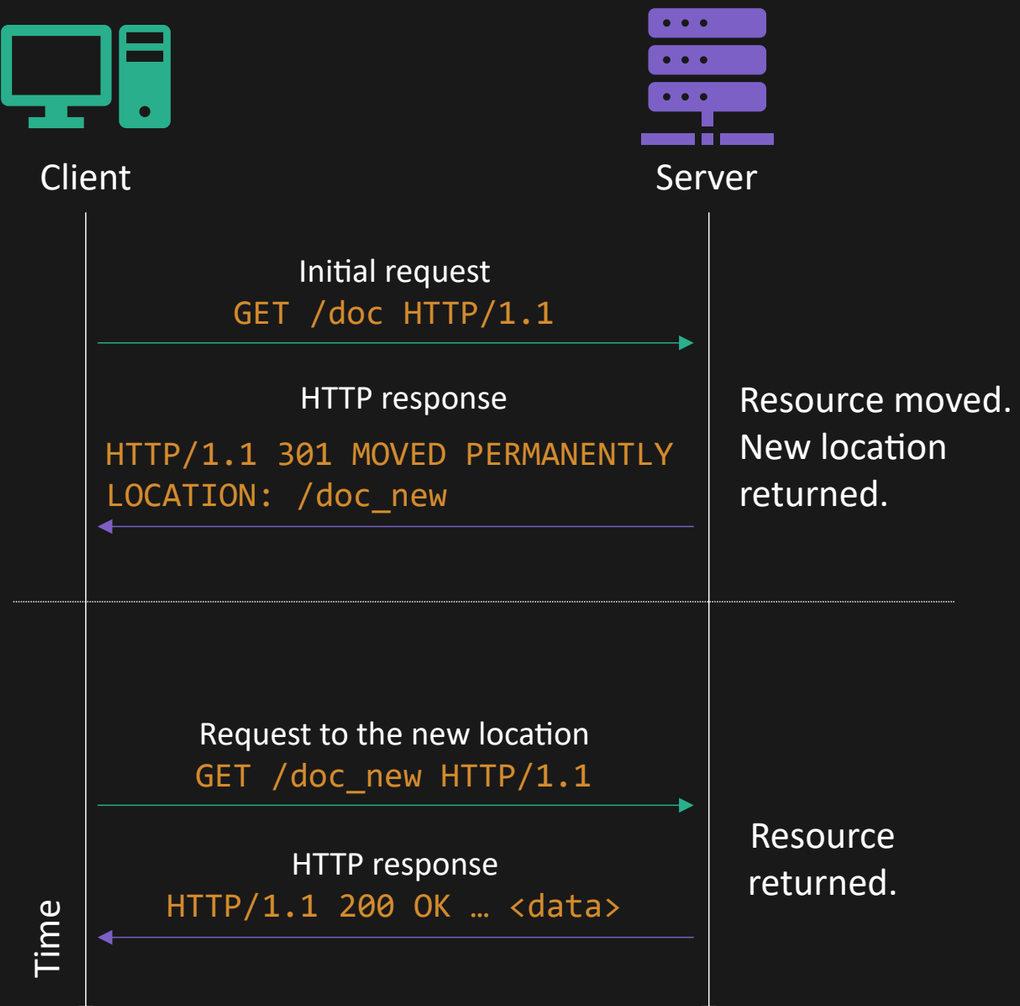
KEY DIFFERENCES

COOKIES	SESSIONS
Cookies are client-side files that contain user information	Sessions are server-side files which contain user information. More secure and tamper-proof.
Cookies are not secure, as data is stored in a text file on the client. If an unauthorized user gets access to a client's browser, they can tamper with the data.	Sessions are more secured compared to cookies, as they save data in encrypted form and are stored on the webserver.
Cookie ends depending on the lifetime you set for it	A session ends when a user closes their browser.
A cookie is not dependent on Session.	A session is dependent on Cookie.
You don't need to start cookie as it is stored on the client's machine	You need to start the session.
There is no way to delete or unset a cookie as it resides on the client.	You can destroy (invalidate) a session from the server.
The official maximum cookie size is 4KB	Within a session you can store as much data as you like. The only limits you can reach is the maximum memory consumable at one time.

REDIRECTION

In HTTP, redirection is triggered by a server sending a special redirect response to a request. Redirect responses have status codes that start with `3`, and a `Location` header holding the URL to redirect to.

When browsers receive a redirect, they immediately load the new URL provided in the `Location` header. Besides the small performance hit of an additional round-trip, users rarely notice the redirection.

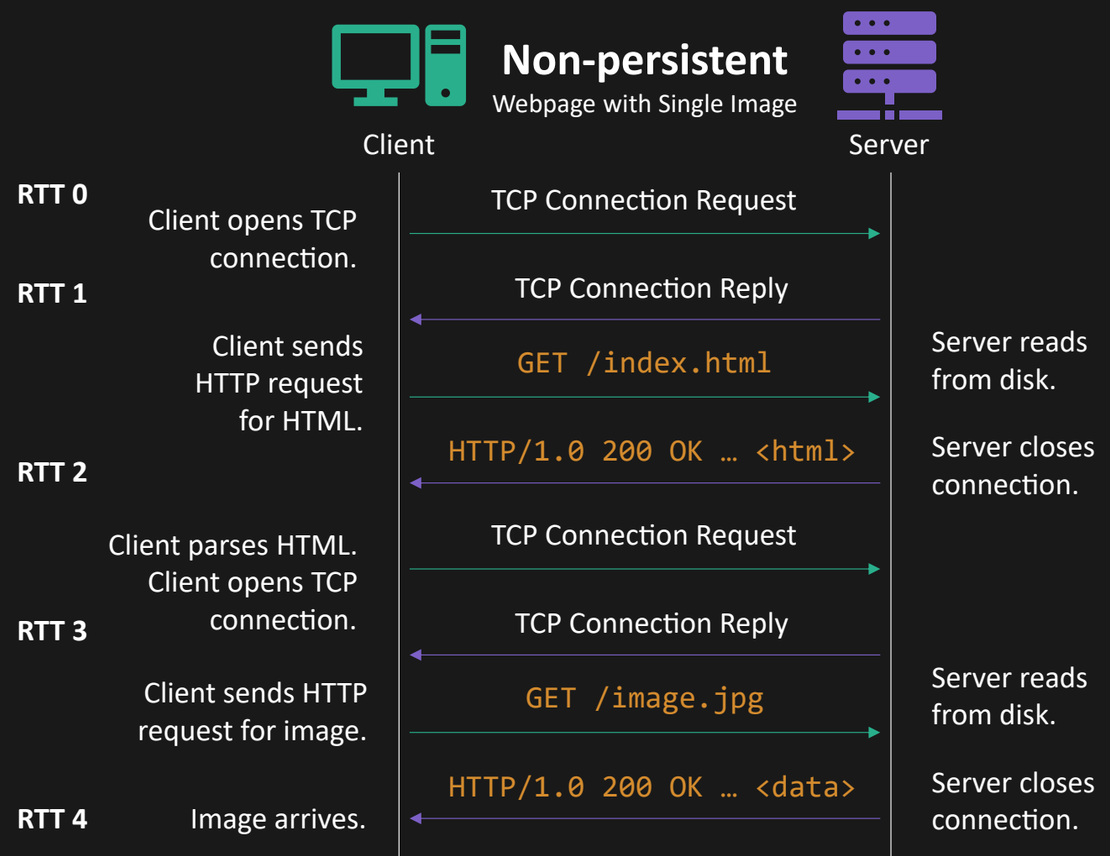


KEEP-ALIVE - PERSISTENT CONNECTIONS

HTTP 1.0 PROBLEM:

- Each request opens new connection
  - Starting up is slow
  - Takes several packets
- Short transfers are hard on TCP
  - Stuck in “slow start” phase of TCP connection
  - Loss recovery is poor when windows are small
- Lots of extra connections
  - Increases server state/processing

Sources: Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).



KEEP-ALIVE - PERSISTENT CONNECTIONS

HTTP 1.1 SOLUTION:

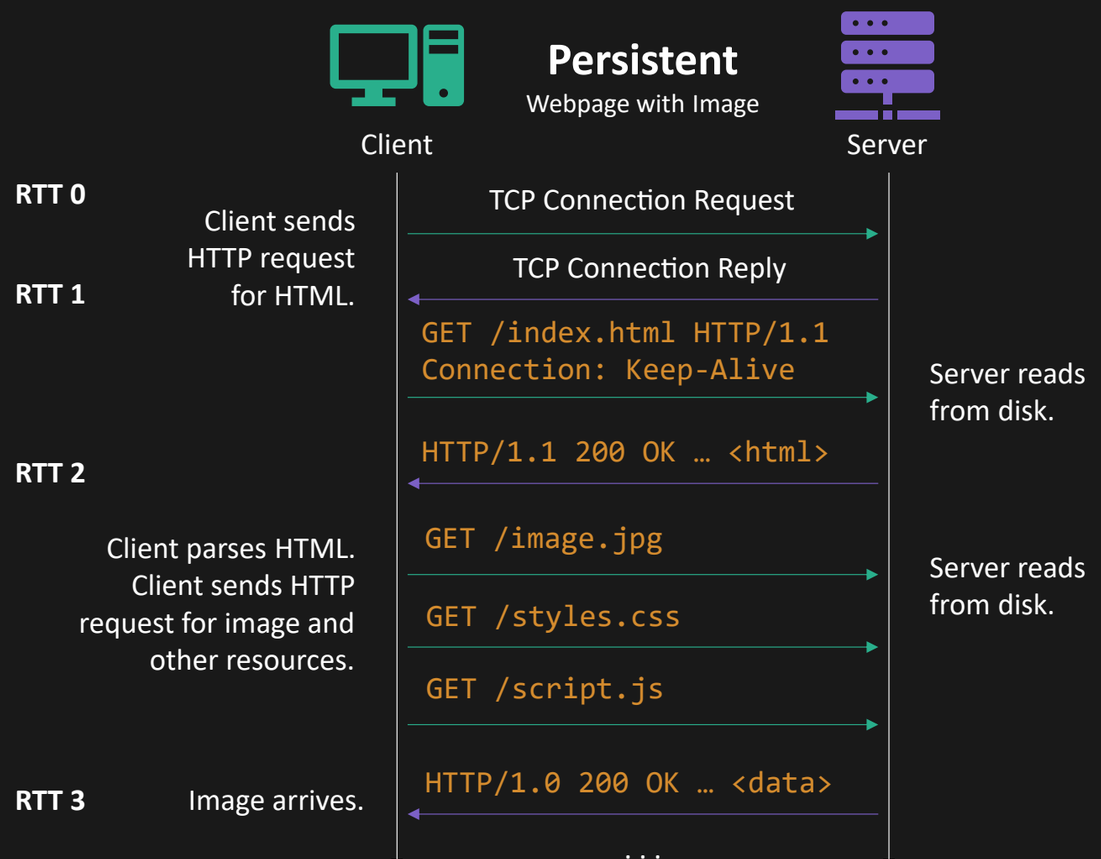
- Keeps connection open for a time after server response so that multiple requests can ride on single connection
- Reduced connection setup overhead.

```
GET /index.html HTTP/1.1
Connection: Keep-Alive

... Multiple HTTP requests ...

GET /script.js HTTP/1.1
Connection: Close
```

Sources: Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).



NON-PERSISTENT

- `HTTP/1.0`
- Server parses request, responds, and closes TCP connection
- 2 RTTs to fetch each object
- Each object transfer suffers from slow start
- Most 1.0 browsers used parallel TCP connections.

PERSISTENT

- Default for `HTTP/1.1`
- On same TCP:
  - connection: server,
  - parses request, responds, parses new request, …
- Client sends requests for all referenced objects as soon as it receives base HTML.
- Fewer RTTs and less slow start.
- Prefetching

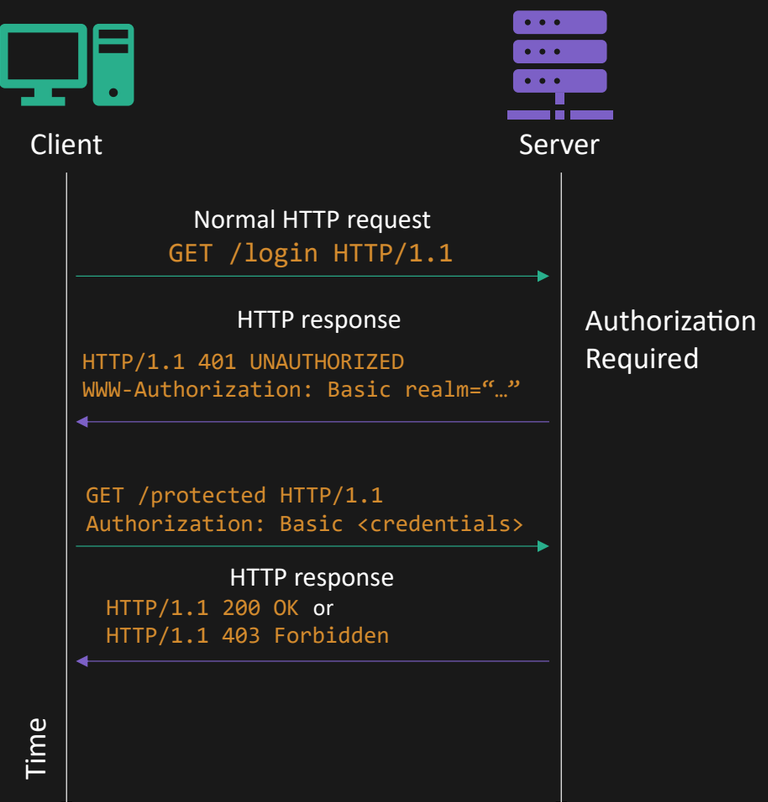
Sources: Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).

BASIC AUTHENTICATION

- When challenged, the client sends their user ID and password in the clear to the server.
- Not secure enough (snooping is easy) but useful for simple things

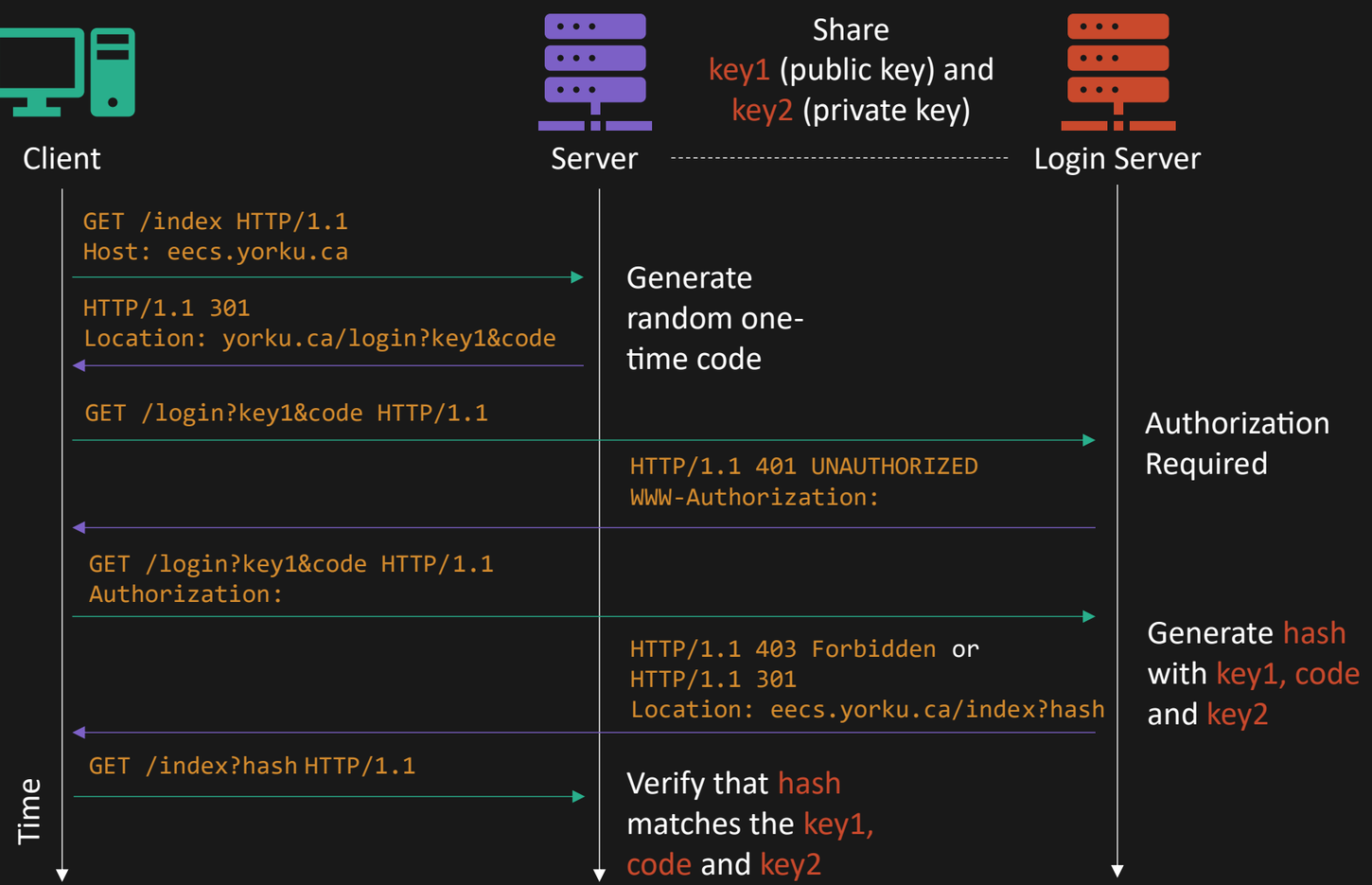
CLIENT-SERVER INTERACTION: AUTHENTICATION

- Authentication goal: Control access to server documents
- Stateless: client must present authorization in each request
- Authorization: typically name, password
  - Sends `Authorization` header line in request
  - If no authorization presented, server refuses access, sends `WWW-Authenticate` header line in response.
- Browser caches name and password so that user does not have to repeatedly enter it.



Sources: Protocols and HTTP, Web Browsers and Servers, Caching, DNS, David A. Penny (2001).

OPEN AUTHENTICATION (OAUTH)



WEB CACHES (PROXY SERVERS)

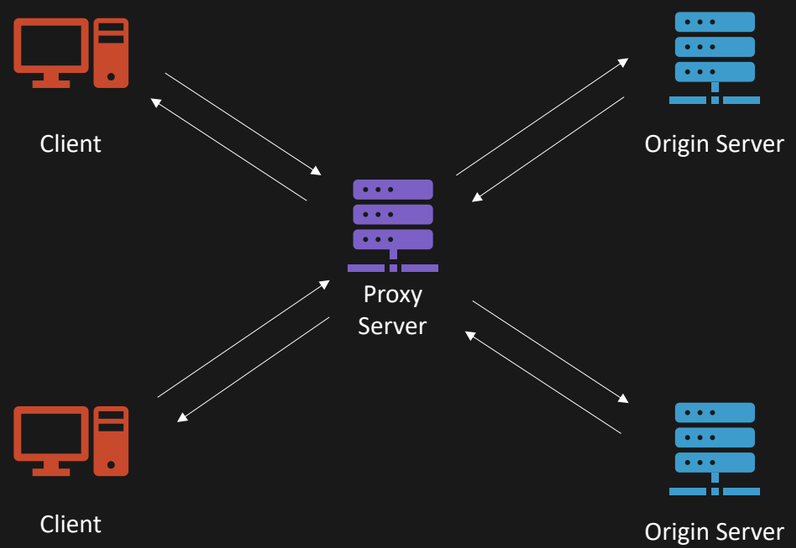
WEB CACHING

- Improve performance
  - Scalability
  - Response time
  - Load balancing
  - Availability
  - Saves network and server resources
- Proxy cache
  - Done at the client-side

GOAL

Fill client request without going to origin server.

- User sets browser: Web accesses via web cache
- Client sends all HTTP requests to web cache
  - If object at web cache, web cache immediately returns object in HTTP response
  - Else requests object from origin server, then returns HTTP response to client



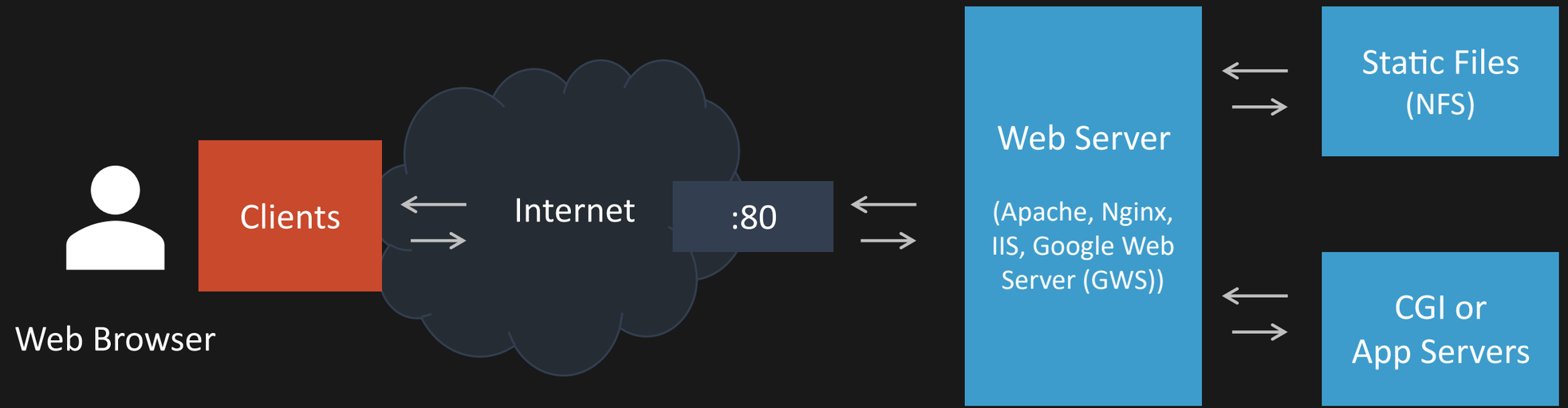
HTTP

SERVICES

HTTP CLIENT

Web browser = TCP client + HTTP + HTML/CSS/JS + DOM

```java
public class HTTPClient {
  public static void main(String[] args) throws Exception {
    PrintStream out = System.out;
    try (Scanner in = new Scanner((new URL(args[0])).openStream()))
      while (in.hasNextLine()) {
        out.println(in.nextLine());
      }
    }
  }
}
```



HTTP SERVER

Web server = TCP Server + Port 80 + HTTP.

- Built-in static file serving
- Built-in scalability
- Built-in security (HTTPS + auth) via `.htaccess`
- Built-in telemetry (logs and error logs)
- Extensibility: PHP (can violate view migration); CGI (good & language agnostic); App Servers (best): Tomcat JSP, WebSphere, WebLogic, NodeJS, ASP.NET, …).

COMMON GATEWAY INTERFACE (CGI)

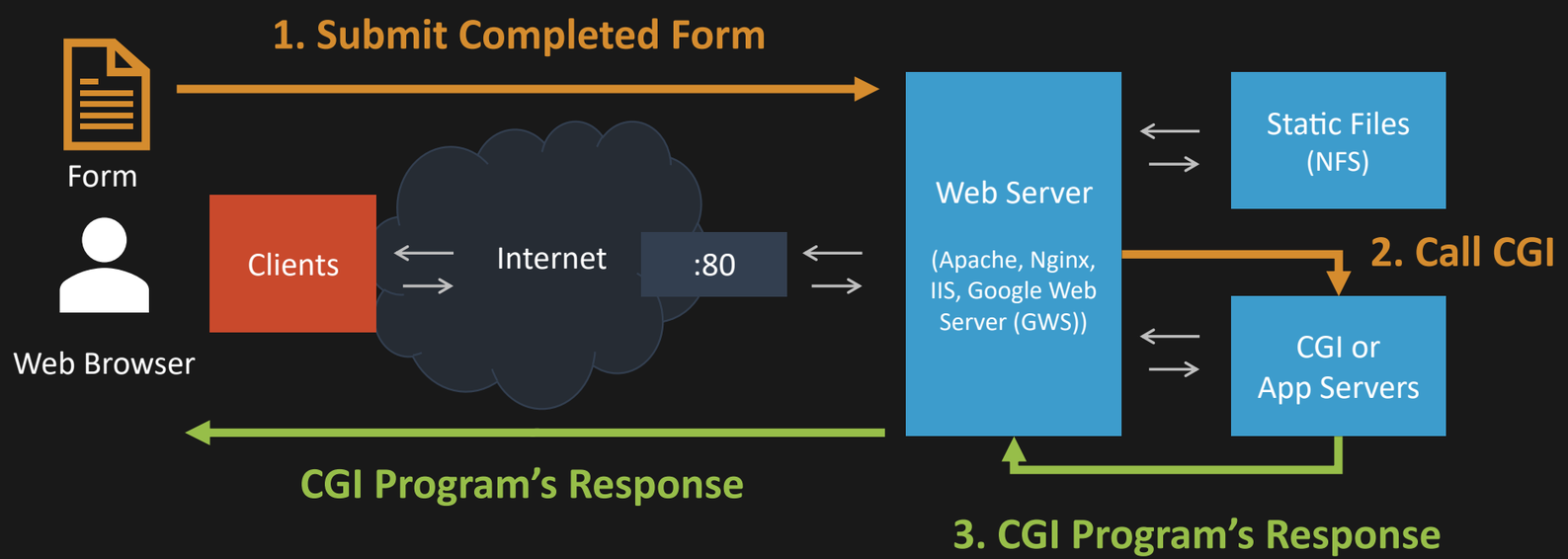
The Common Gateway Interface (CGI) is a simple interface for running external programs, software or gateways under an information server in a platform-independent manner.

Sources: The Common Gateway Interface (CGI) Version 1.1 and CGI - CSC309F Programming on the Web, David A. Penny (2001).

1. The client makes a request by specifying a URL and additional info.
2. The webserver:
    a. Receives the request. (in the URL)
    b. Identifies the request as a CGI request.
    c. Locates the program corresponding to the request.
    d. Starts up the handling program (heavy weight process creation!!)
    e. Feeds request parameters to the handler (through stdin or environment variables).
3. The Handler:
    a. Executes with the given request parameters
    b. The Output of the handler is sent via stdout back to the webserver for rerouting back to the requesting web browser.
    c. Output is typically a web page. Could be plaintext, JSON, or XML.
    d. Terminates.

COMMON GATEWAY INTERFACE (CGI)

The Common Gateway Interface (CGI) is a simple interface for running external programs, software or gateways under an information server in a platform-independent manner.



Sources: The Common Gateway Interface (CGI) Version 1.1 and CGI - CSC309F Programming on the Web, David A. Penny (2001).

MORE ON CGI

CGI defines the abstract parameters, known as metavariables, which describe the client's request. Together with a concrete programmer interface this specifies a platform-independent interface between the script and the HTTP server.

METAVARIABLES

AUTH_TYPE	REMOTE_IDENT	QUERY_STRING
CONTENT_LENGTH	REMOTE_USER	REMOTE_ADDR
CONTENT_TYPE	REQUEST_METHOD	REMOTE_HOST
GATEWAY_INTERFACE	SCRIPT_NAME	SERVER_PROTOCOL
PATH_INFO	SERVER_NAME	SERVER_SOFTWARE
PATH_TRANSLATED	SERVER_PORT	

EXAMPLE

```
GET /~vwchu/test.cgi?key=value&a=1234 HTTP/1.1
```

Item	Value
GATEWAY_INTERFACE	CGI/1.1
SERVER_PROTOCOL	HTTP/1.1
SERVER_SOFTWARE	Apache/2.4.48 (Unix) PHP/7.4.20 OpenSSL/1.0.2k mod_wsgi/4.7.1 Python/3.8
SERVER_NAME	www.eecs.yorku.ca
SERVER_PORT	443
REQUEST_METHOD	GET
SCRIPT_NAME	/~vwchu/test.cgi
QUERY_STRING	key=value&a=1234
REMOTE_ADDR	130.63.96.170
REMOTE_PORT	39256
REMOTE_USER	vwchu

Sources: CGI - CSC309F Programming on the Web, David A. Penny (2001)..

ASIDE

REFACTORING OUR CODE

GOALS

- Code maintainability
- Modularity
- Encapsulation & abstraction
- Separation of concerns
- Handling complexity
- We shouldn't have to rebuild the machinery that runs the webserver every time we want to create a web service!

```java
public class HTTPServer extends Thread {
  ...
  public void run() {
    log.printf("Connected to %s:%d\n", client.getInetAddress(),
                                       client.getPort());

    try (
      Socket client   = this.client;
      Scanner req     = new Scanner(client.getInputStream());
      PrintStream res = new PrintStream(client.getOutputStream(), true),
    ) {
      String request = req.nextLine();
      String method, uri, version;

      StringWriter response = new StringWriter();
      PrintWriter  writer   = new PrintWriter(response);

      try (Scanner parse = new Scanner(request)) {
        method   = parse.next().toUpperCase();
        uri      = parse.next();
        version  = parse.next().toUpperCase();
      }

      int status = 200;

      Map<String, String> reqHeaders = getHeaders(req);
      Map<String, Object> resHeaders = new HashMap<>();

      resHeaders.put("Server", "Java HTTP Server : 1.0");
      resHeaders.put("Date", new Date());
      resHeaders.put("Content-Type", "text/plain");

      try {
        if (!method.equals("GET") && !method.equals("HEAD") &&
            !method.equals("POST")) {
          status = 501;
        } else if (!version.equals("HTTP/1.1")) {
          status = 505;
        } else {
          String[] components = getURIComponents(uri);
          uri = components[0];
          Map<String, String> qs = getQueryStrings(components[1]);
          String body = null;

          if (method.equals("POST")) {
            body = getRequestBody(req);
          }
          ...
```

```java
          switch (uri) {
            case "/":
              response.println("Hello! Welcome to the server.");
              break;

            case "/calc":
              ...

            case "/add":    case "/subtract": case "/multiply":
            case "/divide": case "/exponent":
              ...

            case "/students":
              ...

            default:
              status = 404;
          }
        }
      } catch (Exception err) {
        log.println(err.getMessage());
        err.printStackTrace(log);
        status = 500;
      }

      if (status != 200 && response.toString().length() == 0) {
        writer.println(httpResponseCodes.get(status));
      }

      String responseText = response.toString();
      resHeaders.put("Content-Length", responseText.getBytes().length);
      sendHeaders(res, status, resHeaders);

      if (!method.equals("HEAD")) {
        res.println(responseText);
      }

      res.flush(); // flush character output stream buffer
    } catch (Exception e) {
      log.println(e);
    } finally {
      log.printf("Disconnected from %s\n", clientAddress);
    }
  }

  public static void main(String[] args) throws Exception {
    ...
  }
}
```

ASIDE

REFACTORING OUR CODE

GOALS

- Code maintainability
- Modularity
- Encapsulation & abstraction
- Separation of concerns
- Handling complexity
- We shouldn't have to rebuild the machinery that runs the webserver every time we want to create a web service!

```java
public class HTTPServer extends Thread {
  ...
  public void run() {
    log.printf("Connected to %s:%d\n", client.getInetAddress(),
                                       client.getPort());

    try (
      Socket client   = this.client;
      Scanner req     = new Scanner(client.getInputStream());
      PrintStream res = new PrintStream(client.getOutputStream(), true)
    ) {
      String request = req.nextLine();
      String method, uri, version;

      StringWriter response = new StringWriter();
      PrintWriter  writer   = new PrintWriter(response);

      try (Scanner parse = new Scanner(request)) {
        method   = parse.next().toUpperCase();
        uri      = parse.next();
        version  = parse.next().toUpperCase();
      }

      int status = 200;

      Map<String, String> reqHeaders = getHeaders(req);
      Map<String, String> resHeaders = new HashMap<>();

      resHeaders.put("Server", "Java HTTP Server : 1.0");
      resHeaders.put("Date", new Date());
      resHeaders.put("Content-Type", "text/plain");

      try {
        if (!method.equals("GET") && !method.equals("HEAD") &&
            !method.equals("POST")) {
          status = 501;
        } else if (!version.equals("HTTP/1.1")) {
          status = 505;
        } else {
          String[] components = getURIComponents(uri);
          uri = components[0];
          Map<String, String> qs = getQueryStrings(components[1]);
          String body = null;

          if (method.equals("POST")) {
            body = getRequestBody(req);
          }
          ...
```

```java
          switch (uri) {
            case "/":
              response.println("Hello! Welcome to the server.");
              break;

            case "/calc":
              ...

            case "/add":    case "/subtract": case "/multiply":
            case "/divide": case "/exponent":
              ...

            case "/students":
              ...

            default:
              status = 404;
          }
        }
      } catch (Exception err) {
        log.println(err.getMessage());
        err.printStackTrace(log);
        status = 500;
      }

      if (status != 200 && response.toString().length() == 0) {
        ...
      }

      String responseText = response.toString();
      resHeaders.put("Content-Length", responseText.getBytes().length);
      sendHeaders(res, status, resHeaders);

      if (!method.equals("HEAD")) {
        res.println(responseText);
      }

      res.flush(); // flush character output stream buffer
    } catch (Exception e) {
      log.println(e);
    } finally {
      log.printf("Disconnected from %s\n", clientAddress);
    }
  }

  public static void main(String[] args) throws Exception {
    ...
  }
}
```

The server code and the application code are all mixed together in the `run` method.

ASIDE

REFACTORING OUR CODE

GOALS

- Code maintainability
- Modularity
- Encapsulation & abstraction
- Separation of concerns
- Handling complexity
- We shouldn't have to rebuild the machinery that runs the webserver every time we want to create a web service!

```
1  public class HTTPServer extends Thread {
2    ...
3    public void run() {
4      ...
5      status = doRequest(method, uri, reqHeaders, qs, body, status, resHeaders, writer);
6      ...
7    }
8    private int doRequest(String method, String uri, Map<String, String> reqHeaders,
9          Map<String, String> qs, String body, int status, Map<String, Object> resHeaders,
10         PrintWriter response) throws Exception {
11     switch (uri) {
12       case "/":
13         response.println("Hello! Welcome to the server.");
14         break;
15       case "/calc":
16         ...
17       case "/add":    case "/subtract": case "/multiply":
18       case "/divide": case "/exponent":
19         ...
20       case "/students":
21         ...
22       default:
23         return 404;
24     }
25     return status;
26   }
27   ...
28 }
```

First, move the application code out of the `run` method into a separate `doRequest` method. Note that the `doRequest` method requires a lot of arguments, too many.

ASIDE

REFACTORING OUR CODE

GOALS

- Code maintainability
- Modularity
- Encapsulation & abstraction
- Separation of concerns
- Handling complexity
- We shouldn't have to rebuild the machinery that runs the webserver every time we want to create a web service!

```
public class HTTPServer extends Thread {
  ...
  private void doRequest(
        Request req,
        Response res) throws Exception {
    ...
  }
  ...
  class Request {
    final String method;
    final String uri;
    final Map<String, String> headers;
    final Map<String, String> qs;
    final String body;

    public Request(
      String method,
      String uri,
      Map<String, String> headers,
      Map<String, String> qs,
      String body
    ) {
      this.method  = method;
      this.uri     = uri;
      this.headers = headers;
      this.qs      = qs;
      this.body    = body;
    }
  }
```

```
  ...
  class Response {
    private int status;
    final Map<String, Object> headers;
    final StringWriter response = new StringWriter();
    final PrintWriter  out      = new PrintWriter(response);

    public Response(int status, Map<String, Object> headers) {
      this.status  = status;
      this.headers = headers;
    }
    public int getStatus() {
      return status;
    }
    public void setStatus(int status) {
      this.status = status;
    }
  }
}
```

So, we will group them into `Request` and `Response` classes.

ASIDE

REFACTORING OUR CODE

GOALS

- Code maintainability
- Modularity
- Encapsulation & abstraction
- Separation of concerns
- Handling complexity
- We shouldn't have to rebuild the machinery that runs the webserver every time we want to create a web service!

```
1  public abstract class HTTPServer extends Thread {
2    ...
3    protected abstract void doRequest(Request req, Response res) throws Exception;
4    ...
5  }
6
7  public class MainService extends HTTPServer {
8    public MainService(Socket client) {
9      super(client);
10   }
11   protected void doRequest(Request req, Response res) throws Exception {
12     switch (req.uri) {
13       case "/":
14         res.out.println("Hello! Welcome to the server.");
15         break;
16       case "/calc":
17         ...
18       case "/add":    case "/subtract": case "/multiply":
19       case "/divide": case "/exponent":
20         ...
21       case "/students":
22         ...
23       default:
24         res.setStatus(404);
25     }
26   }
27   public static void main(String[] args) throws Exception {
28     ...
29   }
30 }
```

Take the `doRequest` method entirely out of the `HTTPServer` class into a subclass, `MainService`.

ASIDE

REFACTORING OUR CODE

GOALS

- Code maintainability
- Modularity
- Encapsulation & abstraction
- Separation of concerns
- Handling complexity
- We shouldn't have to rebuild the machinery that runs the webserver every time we want to create a web service!

```
1  public abstract class HTTPServer extends Thread {
2    ...
3    protected abstract void doRequest(RequestContext req, ResponseContext res) throws Exceptio
4    public void run() {
5      ...
6      RequestContext  request  = new RequestContext(client.getInputStream());
7      ResponseContext response = new ResponseContext(client.getOutputStream());
8      request.processRequest(response);
9      doRequest(request, response);
10     ...
11   }
12 }
13
14 public class RequestContext {
15   ...
16 }
17
18 public class ResponseContext {
19   ...
20 }
```

Put the `Request` class and all of the methods in `HTTPServer` class related to handling a request out of the `HTTPServer` class into a new class called `RequestContext`. Do the same for the `Response` class and all of the methods in `HTTPServer` class related to forming a response, resulting in the `ResponseContext` class.

ASIDE

REFACTORING OUR CODE

GOALS

- Code maintainability
- Modularity
- Encapsulation & abstraction
- Separation of concerns
- Handling complexity
- We shouldn't have to rebuild the machinery that runs the webserver every time we want to create a web service!

```
1  public interface Service {
2    public void doRequest(RequestContext request, ResponseContext response) throws Exception;
3  }
4
5  public class CalcOpService implements Service {
6    public void doRequest(RequestContext request, ResponseContext response) { ... }
7  }
8
9  public class CalcService implements Service {
10   public void doRequest(RequestContext request, ResponseContext response) { ... }
11 }
12
13 public class HelloService implements Service {
14   public void doRequest(RequestContext request, ResponseContext response) {
15     response.body.println("Hello! Welcome to the server.");
16   }
17 }
18
19 public class StudentsService implements Service {
20   public void doRequest(RequestContext request, ResponseContext response) { ... }
21 }
```

Split the `MainService` class into each of the different services. One service per class. Each class has itself own `doRequest` method.

ASIDE

REFACTORING OUR CODE

GOALS

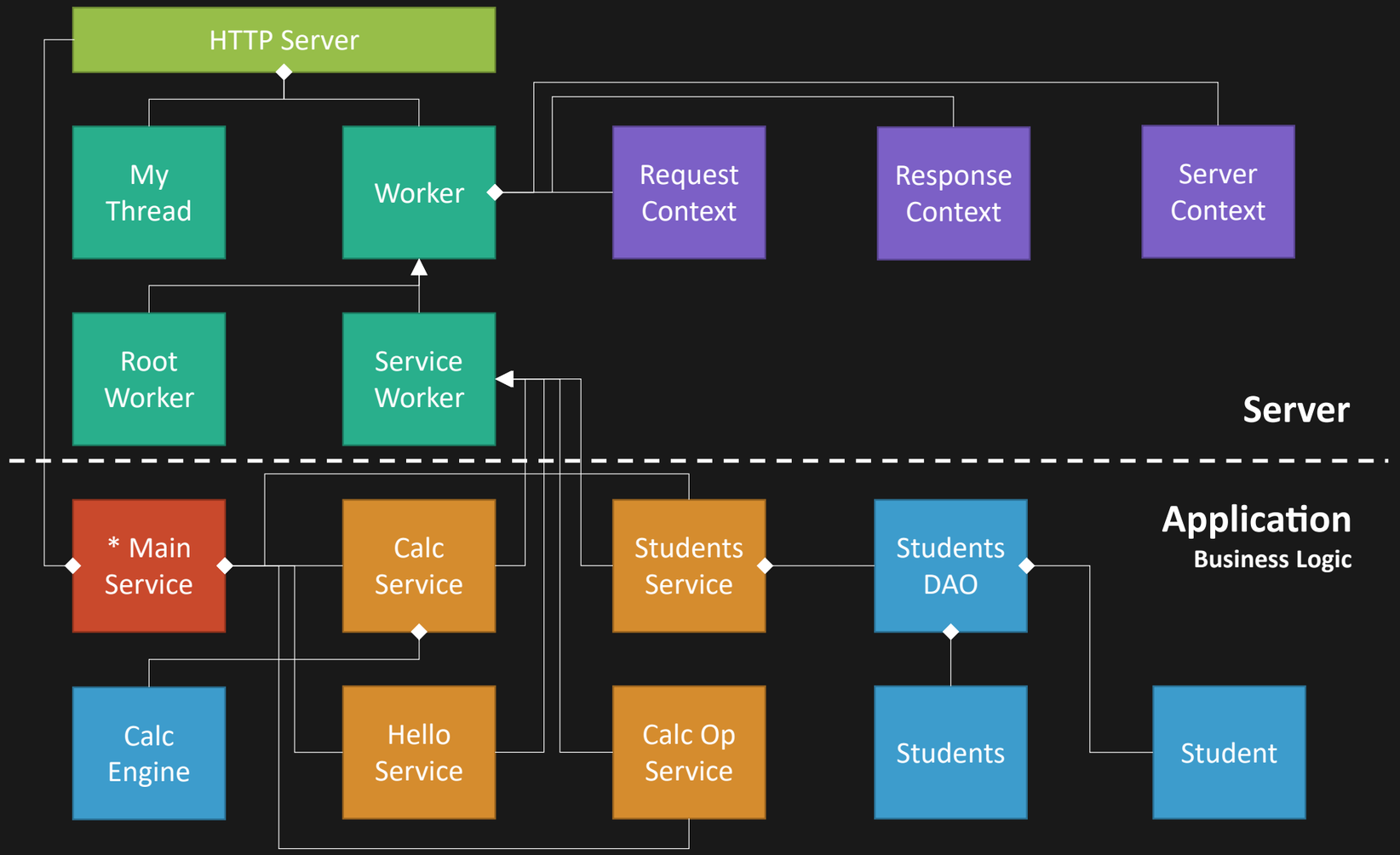
- Code maintainability
- Modularity
- Encapsulation & abstraction
- Separation of concerns
- Handling complexity
- We shouldn't have to rebuild the machinery that runs the webserver every time we want to create a web service!

```java
public class MainService extends HTTPServer {
  protected static final Map<String, Service> services = new HashMap<>();
  static {
    CalcOpService coService = new CalcOpService();
    services.put("/",          new HelloService());
    services.put("/calc",      new CalcService());
    services.put("/add",       coService);
    services.put("/subtract", coService);
    services.put("/multiply", coService);
    services.put("/divide",    coService);
    services.put("/exponent", coService);
    services.put("/students", new StudentsService());
  }
  public MainService(Socket client) {
    super(client);
  }
  protected void doRequest(RequestContext req, ResponseContext res) throws Exception {
    if (services.containsKey(req.uri)) {
      services.get(req.uri).doRequest(req, res);
    } else {
      res.setStatus(404);
    }
  }
  public static void main(String[] args) throws Exception {
    int port = 0;
    InetAddress host = InetAddress.getLocalHost(); // .getLoopbackAddress();
    try (ServerSocket server = new ServerSocket(port, 0, host)) {
      log.printf("Server listening on %s:%d\n", server.getInetAddress(), server.getLocalPort
      while (true) {
        (new MainService(server.accept())).start();
      }
    }
  }
}
```

That leaves our `MainService` class with the barest of `run` methods and a map that registers the available service handlers to their respective resource paths. Later, we will see how Tomcat does it via annotations.

If we continue refactoring, we might end up with something like this…

Despite all this added complexity, it's okay. Because to us, web app developers, the web server's code is a black-box that we access via public APIs.



This slide is intentionally left blank.

Return to Course Page or Part II.