

Dynamic contents and web pages (backend).
Web services.
CGI (Common Gateway Interface).
AJAX.

SCOMRED, January 2022

Dynamic contents

When a web client sends an HTTP request to a web server, a target URI is identified on the request. How the server will handle that request depends on the server configuration, namely the behaviour for that specific requested URI must be somehow defined.

Two distinct general cases can be pointed:

- The URI is mapped to a static content, most often stored in a file. Then it's a **static resource** (URI) because the result for such a request is always the same (unless the file content is changed).
- The URI is **mapped to a program or function execution**, such program or function will produce an output, and that's the content being sent back to the client. This is a **dynamic resource** (URI), the result may be different on every request.

Often, a dynamic resource (program or function) will require input data to be processed, it must be sent by the client to the server, incorporated in the HTTP request. For requests with the POST or PUT methods, such input data is transported on the request's body. For requests with the GET method, data to be sent must be included in the URI itself (query-string).

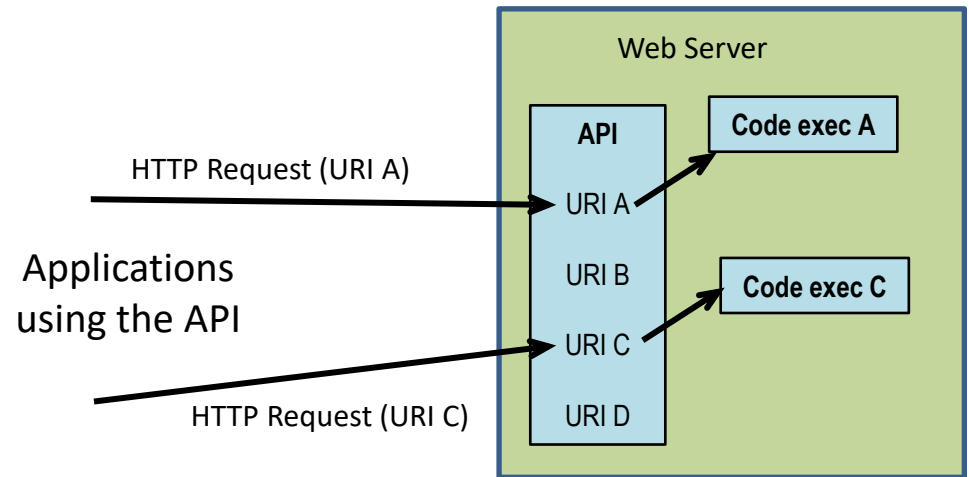
Dynamic resources as an API - Web Services

Dynamic resources provided by a web server are entry points to call functions that may receive input data, execute well defined actions, and may as well produce and return a result. Under all points of view it's a way to call a function.

Because these function call entry points are external to the calling application, they are better classified as services, and because they use HTTP they are called **web services**.

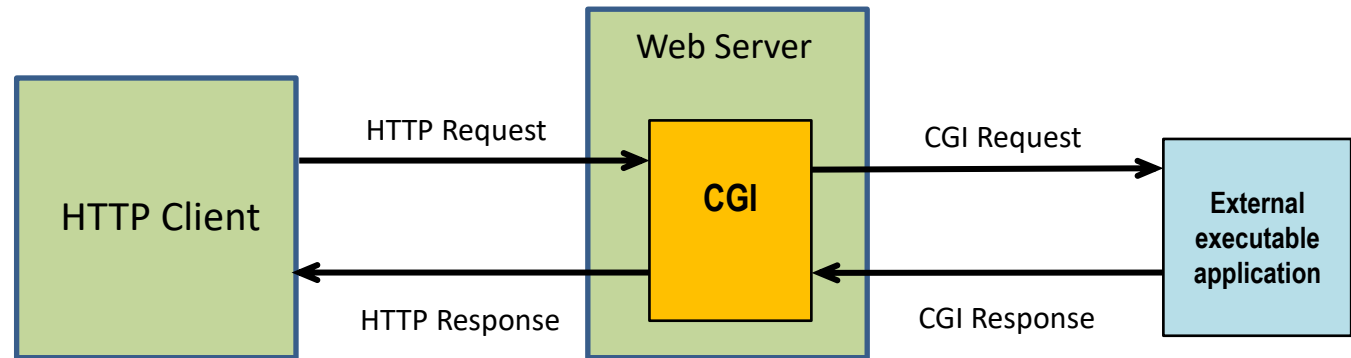
When a web server provides a coherent set of web services, then we can name that as an API (Application Programming Interface).

An API is an interface with a set of available functions that can be called (used) by an application. Of course in our web scenario applications using the API are going to call functions (web services) through HTTP requests.



CGI (Common Gateway Interface)

CGI is a standard (RFC 3875) that specifies how an HTTP server may execute external programs when HTTP clients send requests to some specific resources (URIs). Of course we are talking about dynamic resources. The image below shows the general layout with a Web Server with CGI support.



When a client sends a request to an URI mapped to CGI execution, the server will:

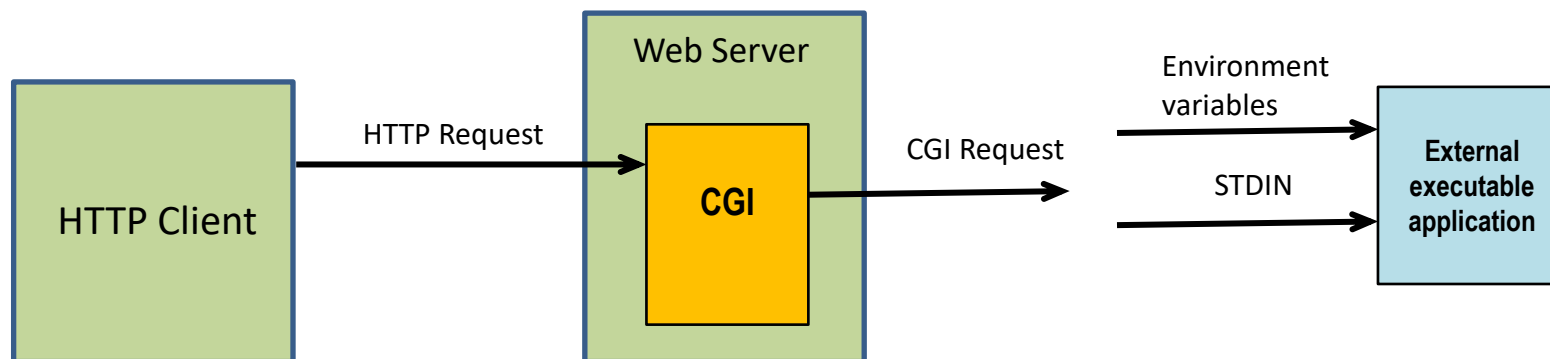
1. Create a **CGI Request** from the **HTTP Request**.
2. Execute the external application and pass to it the CGI Request together with additional data about the received request.
3. Wait for a CGI Response from the external application.
4. Create the **HTTP Response** from the **CGI Response** and sent it .

CGI Request

When a CGI mapped URI is accessed, the server will create a CGI Request from the received HTTP Request, and then executes the external program.

The request is passed to the external program in two manners⁽¹⁾:

- Environment variables are settled before executing the external program, thus available to it, one of those variables is `CONTENT_LENGTH`.
- If the HTTP request has a body (content), then it will be written to the external program's **STDIN**, so the program is expected to read `CONTENT_LENGTH` bytes from its **STDIN** (the HTTP message's body).



⁽¹⁾ The way the CGI Request passing is implemented is not part of the CGI specification and may vary from system to system. This scenario is the most frequent and the one used in the Apache server on Linux systems.

CGI Request – Environment variables

The CGI external program execution environment encompasses a set of variables with data about the HTTP request. Some most relevant are:

REQUEST_METHOD – Contains the up case name of the HTTP request method, e.g. GET, POST, PUT, DELETE.

CONTENT_LENGTH – If the request has a body, the body's length in bytes, otherwise not defined (empty).

CONTENT_TYPE - If the request has a body, the body's Internet Media Type (e.g. text/html), otherwise not defined (empty).

QUERY_STRING – The query-string embedded in the request's URI, following the interrogation mark. If the URI doesn't include a query-string this variable is not defined (empty).

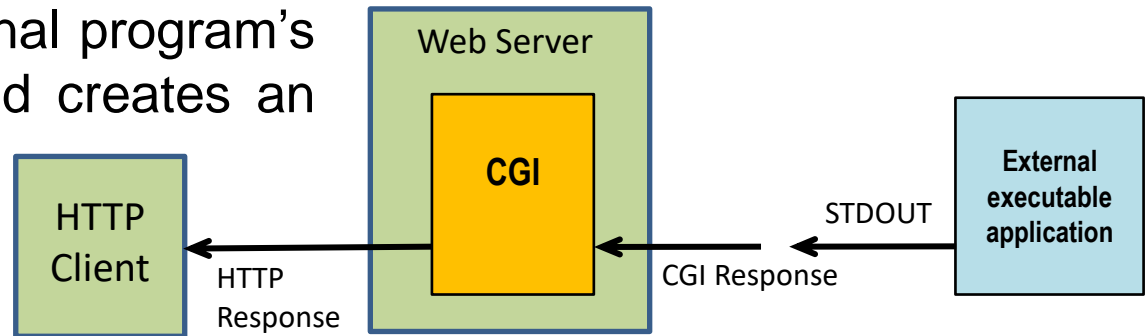
Several other useful variables are defined in RFC 3875.

Additional variables may be defined to store HTTP header fields, such variable names start by the **HTTP_** prefix and are followed by the header field name in up case with any hyphen converted to underscore, e.g. from RFC 3875, header field **Proxy-Authorization** becomes **HTTP_PROXY_AUTHORIZATION** variable name.

CGI Response

The CGI capable Web Server expects the external program to exit and produce a non-empty output on STDOUT. An execution timeout may be enforced and the external program terminated by force if that is exceeded.

The server reads the external program's output (CGI Response) and creates an HTTP Response from it.



The CGI Response (written to STDOUT) consists on two parts: a sequence of HTTP style header field lines, ended by an empty line, and optionally followed by a body/content.

The sequence of header field lines in the CGI Response encompasses standard HTTP header fields and some additional CGI specific header fields.

If the CGI response has a body, then the **Content-Type** header field must be added by the external program, but not the **Content-Length** header field, that will be automatically inserted by the server with the correct value.

CGI header fields

Standard HTTP header fields added by the external program to the CGI response are in principle copied as they are to the HTTP response, however, the server may remove or replace them and insert others.

One important CGI header field is **Status**, if settle by the external program, then it will be used to build the status line on the HTTP response, otherwise the status line on the HTTP response defaults to **200 OK**.

The value of the Status CGI field is the three digits status code followed by a space and the reason-phrase, just as defined by the HTTP specification.

The same rules for HTTP header field lines apply to CGI header field lines: the field-name is not case sensitive, the field-value may be case sensitive, there may be spaces between the colon and the field-value, but not between the field-name and the colon. Unlike what happens with HTTP header field lines, continuation lines (line folding) are not supported for CGI header field lines.

CGI external program execution context

We already know the CGI external program execution environment encompasses a set of variables with data about the received request.

Some other factors may be relevant for the external program execution, one is the **current working directory**. The CGI standard establishes that on systems where such concept exists, it should be set to the directory in which the external program is stored.

One key factor that may impact on the external program's execution is the permissions it will have. When a program is started by a given user, it belongs to that user and has the same permissions.

In fact, Unix systems applications may run with the username of the owner of the executable file instead of the username that starts the application. Those are called **setuid** applications and to be able to do so the executable file is required to have the **setuid** special permission.

In Unix systems, many background services (often referred to as **daemons**), don't run with root permissions, this is specially true for networks services, including typical Web Servers running external CGI programs.

CGI external programs runtime permissions

For the security sake, often services don't run with high permissions (e.g. root user permissions in Linux). The point is, if there's a glitch on the service, and an attacker is able to use it, whatever the attacker is able to do will not be done as root, and thus it's likely to be much less harmful for the system.

Take for instance the popular Apache Web server on Linux, it will start as **root** to read confidential configuration data like private keys, however it will then drop privileges and change to a low permissions username (commonly www-data or apache).

Because CGI external programs are executed by the Web Server daemon, they are likewise executed with that same low permissions username. That's something the CGI external programs developer has to keep in mind.

Running CGI external programs with low permissions assures to some extent that such programs will not be able to disturb the operating system. However, because they all run with the same username, they may interact (in both the good and bad sense) with the Web Server and other CGI external programs, including data stored by them.

Dynamic resources security and input validation

To make things clear, **making available dynamic resources is opening a door to attackers**. Anyone will be then able to trigger the execution of code in the server, if the executed code performs only what it was intended for, that's perfectly ok.

On the other hand, if an attacker is able to somehow change the behaviour of the code being executed, then it may be a severe security issue.

It's up to the developer to ensure the code will do only what's supposed to, and there are no ways to divert its behaviour.

The simplest method to change the code's behaviour is through input data, it might not be always the case, but code being executed will must often receive input data. Such input data content is not controlled by the Web Server it comes from the Web Client which may be under control of a malicious attacker.

Validating received input data is key to ensure the safe execution of code for dynamic contents. This means analysing if the content is what's supposed to be before using it.

HTTP requests' data and input validation

One main cornerstone of secure applications development is input validation. In a frontend/backend distributed web environment, input validations must be enforced on the backend, **the frontend is inherently untrusted**.

Input data for dynamic resources (backend) is provided by the Web Client (frontend) and included in HTTP requests in two ways:

- **The query-string**, included in the URI of the request, following the interrogation mark. In a CGI program it's available in variable `QUERY_STRING`.
- **The request's body**, is supported for POST and PUT requests only. In a CGI program it's available by reading from the standard in. Notice that with POST and PUT requests, a query-string and a body may be used together at the same time.

The whole HTTP request must be validated before effective processing by the backend. If not appropriate, it should be ignored and a suitable HTTP error response sent back to the frontend. Beyond input data, validating an HTTP request includes, for instance, checking if the correct HTTP method was used.

Input data validation

Input data validation is key to attain secure and stable applications, especially when such data is coming from an external and out of control source.

Such out of control sources include an innocent user that simply mistyped something on the user interface and also a malicious attacker trying to change the behaviour of the application. Under a backend's point of view, every HTTP request is out of control, thus it must be thoroughly validated.

To validate input data, it must be known what that data is supposed to be in the first place, and of course that depends on the application itself.

In most cases, it's possible to establish some rules on:

- Data length (minimum and maximum)
- Data contents (e.g. numeric digits only for an unsigned integer). Pattern matching with regular expressions is very useful to validate some restricted contents, for instance a floating point number or an IPv4 address.

Example CGI application in BASH – HASH calculator

To better illustrate some concepts addressed so far let's analyse a very simple CGI application written in BASH.

The purpose of this application is receiving a POST request with a body content, calculate a hash code (aka digest)⁽¹⁾ for the body content and send it back to the client as response. It's a rather simple web service.

The hash algorithm to be used is specified by the **algorithm** key in the query-string, supported values are: MD4, MD5, RIPEMD160, SHA1, SHA224, SHA256, SHA384, and SHA512.

The **openssl** command line utility is used to actually calculate the hash code. If the request is invalid, a **400 Bad Request** response is sent back with a text line describing the issue for debugging purposes.

(1) A hash code is a fixed size value returned by a hash algorithm (aka digest algorithm) after processing a variable size amount of input data. Hash algorithms are deterministic, meaning for the same input data the result is always the same. However, the hash code represents the entire input content, changing a single bit in the input content will change dramatically and unpredictably the resulting hash code. There's a wide range of applications for hash algorithms, including data integrity control and digital signatures.

Example CGI application in BASH – HASH calculator web service

```
#!/bin/bash
### I'm in file /var/www/cgi-bin/hashCalculate
###
M_CONTENT_FILE=/tmp/.scomred-hash-calculate.$$tmp
###
error_response() {
    echo "Status: 400 Bad Request"
    echo "Content-type: text/plain"
    echo ""
    echo "ERROR: ${1}"
    rm -f $M_CONTENT_FILE
    exit
}
###
if [ -n "$CONTENT_LENGTH" ]; then cat > $M_CONTENT_FILE
else error_response "No content found on the request"; fi
if [ "$CONTENT_LENGTH" == "0" ]; then error_response "No content found on the request"; fi
###
if [ "$REQUEST_METHOD" != "POST" ]; then error_response "Invalid method. The only supported method is POST."; fi
if [ -z "$QUERY_STRING" ]; then error_response "No query-string"; fi
ALG="${QUERY_STRING#algorithm=}"
if [ "$ALG" == "$QUERY_STRING" ]; then error_response "Bad query-string: $QUERY_STRING"; fi
case "$ALG" in
    MD4|MD5|RIPEMD160|SHA1|SHA224|SHA256|SHA384|SHA512);;
    *) error_response "Invalid hash algorithm: $ALG";;
esac
HASHCODE="$(openssl dgst -$ALG $M_CONTENT_FILE)"
rm -f $M_CONTENT_FILE
echo "Content-type: text/plain"
echo ""
echo "${HASHCODE#*= }"
###
```

Testing web services (functional tests)

Web services, like the preceding example, are intended to be used by applications, such applications take a role named as web services **consumers** or **requestors**, the web server side is often named as the web services **publisher** or **provider**.

Before making web services available to consumers, they should be carefully tested and proved to be behaving as they should. Without this step, when latter things don't work as expected between consumers and publishers it's impossible to know which side is to blame.

Because a web services consumer is an HTTP client, and so is a web browser, we might expect testing web services with a standard web browser would do perfectly.

However that's not the case, by typing a URL on the web browser window, the only HTTP request method that can be used is GET, and at most a query-string may be specified. No other options and methods are available.

To properly test web services a specific tool is required, one of such tools is the popular **Postman**.

Postman (<https://www.getpostman.com/>)

Postman is one of the most popular tools for web services and API testing, it's available for most operating systems and there are also versions capable of running on Web Browsers as extensions.

With Postman, HTTP requests using any HTTP method can be created and sent to the server. All sorts of adjustments and settings regarding the request are possible, for instance regarding header line fields and content.

When the response is received from the provider, all details are available, including the status code, header line fields, and the content.

Next, there's a sequence screenshots of tests with Postman over the preceding web service example, under these tests the web service URL was:

<https://vs181.dei.isep.ipp.pt/cgi-bin/hashCalculate>

Test 1 – GET request

The screenshot shows a web client interface with the following elements:



- Environment: Normal, Basic Auth, Digest Auth, OAuth 1.0, No environment (dropdown), keyboard icon, close icon, 0.
- URL: `https://vs181.dei.isep.ipp.pt/cgi-bin/hashCalculate`
- Method: GET (dropdown)
- URL params: (icon) URL params
- Headers: (icon) Headers (0)
- Buttons: Send, Preview, Add to collection, Reset
- Body: Cookies (5), Headers (5), STATUS 400 Bad Request, TIME 180 ms
- View options: Pretty, Raw, Preview, (icon), (icon), JSON, XML
- Response body:



```
1 ERROR: No content found on the request
2
```

The first validation by the CGI program is about existing a content, so independently from other issues on the request, if the requests has no body the outcome is always this.

This would also be the result of accessing the URL with a standard web browser.

Test 2 – PUT (with body)

Normal Basic Auth Digest Auth OAuth 1.0 No environment   0



https://vs181.dei.isep.ipp.pt/cgi-bin/hashCalculate PUT  URL params  Headers (0)

form-data x-www-form-urlencoded raw Text

```
1 SCOMRED sample data
```

Send Preview Add to collection Reset

Body Cookies (5) Headers (5) STATUS 400 Bad Request TIME 377 ms

Pretty Raw Preview   JSON XML

```
1 ERROR: Invalid method. The only supported method is POST.
2
```

Test 3 – POST (with body), but no query-string

Normal Basic Auth Digest Auth OAuth 1.0 No environment 0

https://vs181.dei.isep.ipp.pt/cgi-bin/hashCalculate POST URL params Headers (0)

form-data x-www-form-urlencoded raw Text

```
1 SCOMRED sample data
```




Send Preview Add to collection Reset



Body Cookies (5) Headers (5) **STATUS** 400 Bad Request **TIME** 215 ms


Pretty Raw Preview JSON XML

```
1 ERROR: No query-string
2
```

Test 4 – POST (with body), but invalid query-string

Normal Basic Auth Digest Auth OAuth 1.0  No environment   0



https://vs181.dei.isep.ipp.pt/cgi-bin/hashCalculate?algo=SHA512 POST  URL params  Headers (0)

form-data x-www-form-urlencoded raw Text 

```
1 SCOMRED sample data
```

Send Preview Add to collection Reset

Body Cookies (5) Headers (5) **STATUS** 400 Bad Request **TIME** 214 ms

Pretty Raw Preview   JSON XML

```
1 ERROR: Bad query-string: algo=SHA512
2
```

Test 5 – POST (with body), unsupported algorithm

Normal Basic Auth Digest Auth OAuth 1.0 No environment 0

https://vs181.dei.isep.ipp.pt/cgi-bin/hashCalculate?algorithm=SSSS POST URL params Headers (0)

form-data x-www-form-urlencoded raw Text

```
1 SCOMRED sample data
```

Send Preview Add to collection Reset

Body Cookies (5) Headers (5) **STATUS** 400 Bad Request **TIME** 226 ms

Pretty Raw Preview JSON XML

```
1 ERROR: Invalid hash algorithm: SSSS
2
```

Test 6 – POST (with body), success

Normal Basic Auth Digest Auth OAuth 1.0 No environment 0

https://vs181.dei.isep.ipp.pt/cgi-bin/hashCalculate?algorithm=SHA512 POST URL params Headers (0)

form-data x-www-form-urlencoded raw Text

```
1 SCOMRED sample data
```

Send Preview Add to collection Reset

Body Cookies (5) Headers (8) **STATUS** 200 OK **TIME** 163 ms

Pretty Raw Preview JSON XML

```
1 b77de6990f28704dde0848999e76e94953e3e9d33adf4c5e261025fcf9a80bc60940648e5124c7fc485807b55f135f41e83b7c4f3b2f050c899323d8102f592f
2
```

Checking if the hash code is correct

Enter your text below:

SCOMRED sample data

Generate Clear All MD5 SHA1 SHA256 Password Generator

Treat each line as a separate string

SHA512 Hash of your string:

B77DE6990F28704DDE0848999E76E94953E3E9D33ADF4C5E261025FCF9A80BC60940648E5124C7FC485807B55F135F41E83B7C4F3B2F050C899323D8102F592F

Copyright © 2012 - 2018 [PasswordsGenerator.net](http://passwordsgenerator.net). All Rights Reserved.

Web Services and Web Browsers

From the web services concept, which excludes direct end-users' interaction, it could be anticipated web browsers are out of scope. Nevertheless, modern web browsers are themselves able to run applications, namely in JavaScript language. This makes them able to take part in web services architecture.

Current web browsers support the **XMLHttpRequest** object, in essence it's an HTTP client and allows a web page to, whenever it desires, make an HTTP request, retrieve data, and typically use that data to update parts of the page being displayed. This may be done without actually reloading the page, by using the HTML DOM (Document Object Model).

Requests with the **XMLHttpRequest** object are by default asynchronous, this means, before triggering the request, a response handling function is defined (call-back function). Then, the request itself will not block the web browser on waiting for the response, if, and when the response arrives, then the response handling function is executed.

This technique is called **AJAX** (Asynchronous JavaScript and XML), by using it, the traditional web pages' behavior, requiring a reload or submission for an update with fresh data from the server, is overcome.

Web browsers as consumers - JavaScript

The standard use of web browsers: retrieve contents and display them to end-users, has no place in the web services model.

Having said that, the fact is, modern web browsers are themselves platforms where applications can be run, for instance using JavaScript.

The **XMLHttpRequest** object is an HTTP client available in JavaScript, by using it, JavaScript applications/functions may become web services' consumers.

In this object, the `open()` method is used to create a request (not actually send it), any HTTP method can be used over a specified URL, by default the request is asynchronous. HTTP header lines can be settled one by one with the `setRequestHeader()` method before finally sending the request to the provider by calling the `send()` method.

Asynchronous request means when calling the `send()` method the application will not be blocked waiting for the response, this is most important for a web browser.

If data it to be sent (PUT or POST), it can be specified as argument of the `send()` method, data can also be sent with GET, but in that case it will be part of the URI provided to the `open()` method.

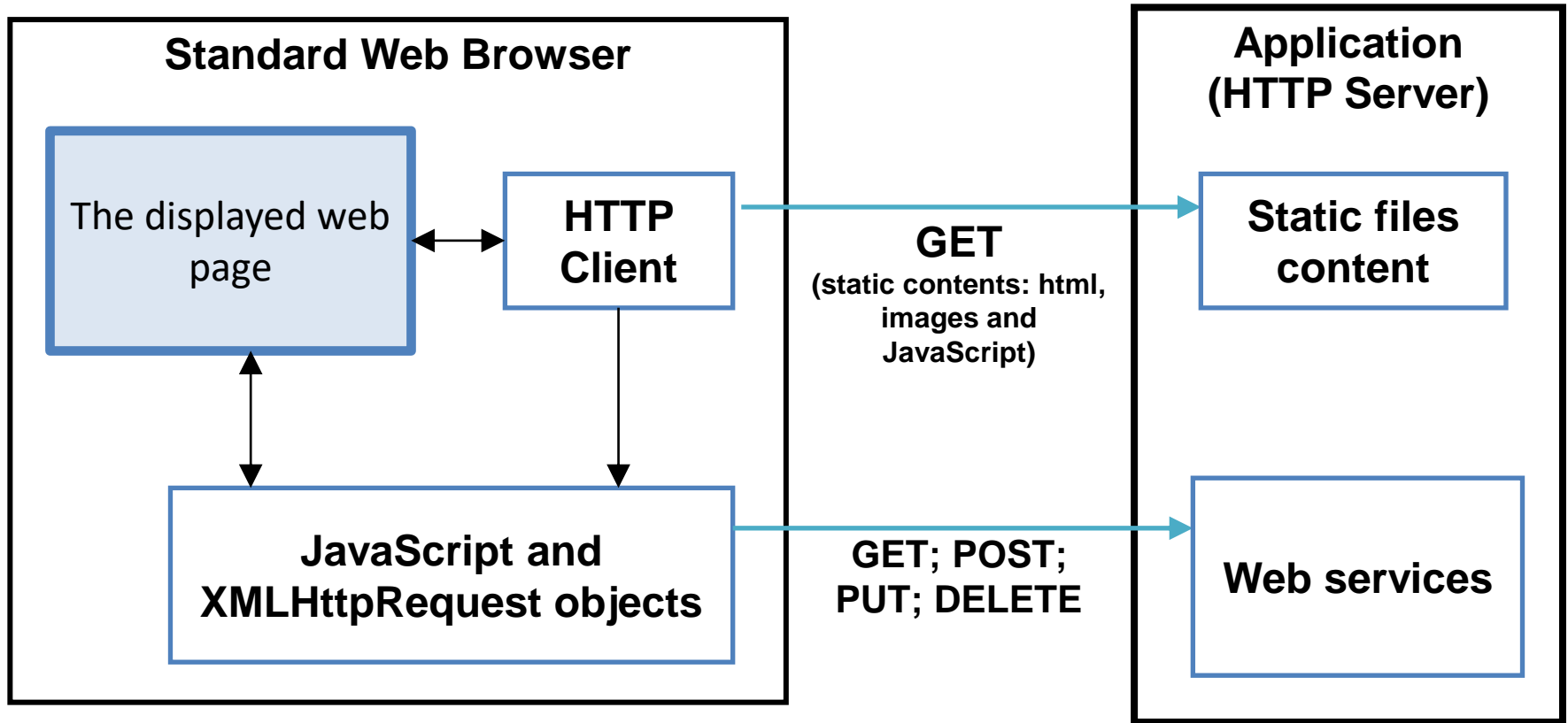
Web browsers as consumers - JavaScript

Before sending an asynchronous request, the object's property **onload** must be assigned with a call-back function to be called asynchronously when the response arrives. Once the response arrives, within the **onload** call-back function, the **status** property contains the HTTP status code, 200 for ok.

By default the **XMLHttpRequest** object has **timeout** zero, this stands for no timeout and it waits forever for a response. However, the **timeout** property can be assigned with a value in milliseconds to change that default behaviour. If **timeout** is settled, then the **ontimeout** property should be assigned with a call-back function to handle that scenario.

Event property	Standing for ...
onreadystatechange	The state has changed, the state property will contain one of the following values: 0 (request not initialized); 1 (server connection established); 2 (request received); 3 (processing request); 4: (request finished and response is ready)
onabort	The request was aborted by calling the abort() method.
onerror	The request has failed.
onload	The request was successful (load). The responseText property hold the response's content.
onloadend	The request processing has finished successfully or not.
ontimeout	The request failed due to timeout (as defined by the timeout property value greater than zero).

AJAX usage scenario



The Web Browser starts by using GET requests to load the page and all referred static resources stored in files on the server side, including images and JavaScript functions.

Then, loaded JavaScript functions are triggered by events or timers and they use **XMLHttpRequest** objects to make HTTP requests to web services provided by the server and change the page contents through DOM.