# Web Services.

# AJAX example – voting.

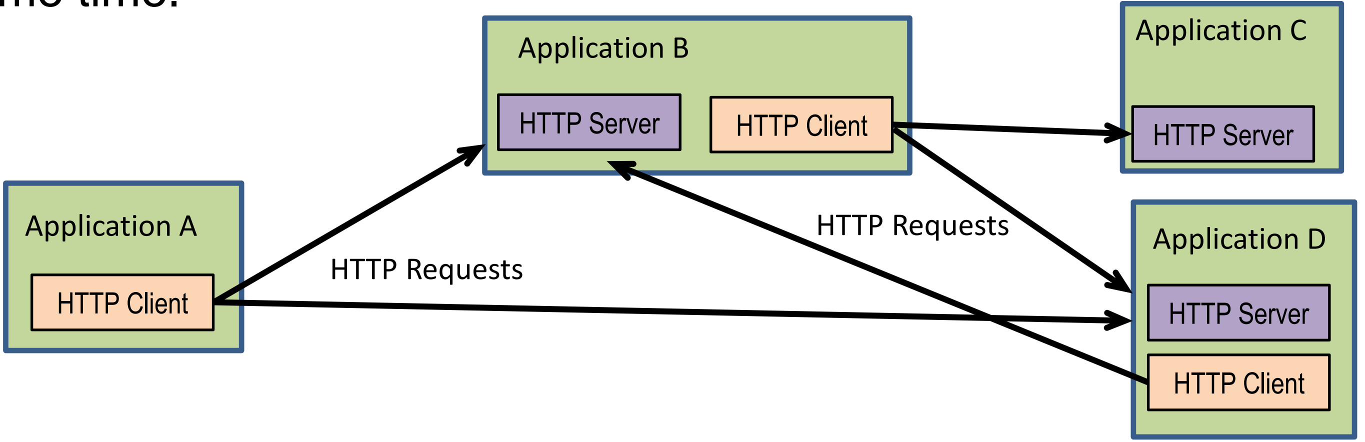
SCOMRED, January 2022

# HTTP/HTTPS as general-purpose application protocol

Designing and implementing a new application layer protocol for a distributed applications architecture is a rather significant effort and investment. This investment can be avoided if an already existing application layer protocol is reused, and eventually adapted.

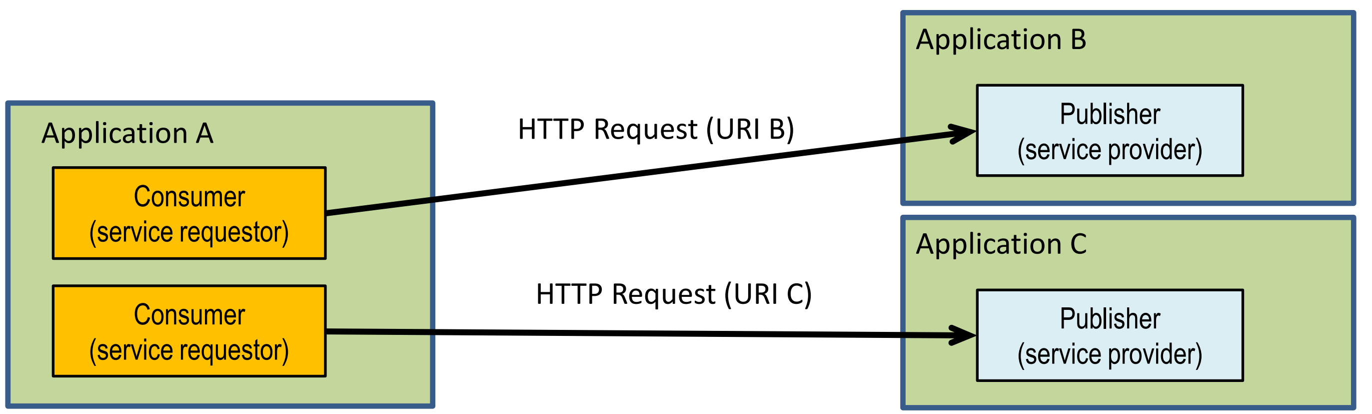
We must bear in mind this may not be the best technical option, however, it might be the best option under investment point of view. Some adaptations to the original protocol usage may be required because it was designed with a different purpose, nevertheless, the protocol specification itself must be kept.

Because it's standard, simple and flexible, HTTP/HTTPS is widely adopted for this purpose. To achieve this, each application contains an HTTP server, an HTTP client, or both. The client-server model is preserved, though one application can be client and server at the same time.



# Web Services

The central concept on web services is the use of HTTP for application-to-application communications without direct human involvement.
To make use of a web service, one application (the service **requestor** or **consumer**) assumes the HTTP client's role, and another (the service **provider** or **publisher**) the HTTP server's role. The web service is made available to service requestor applications by the service provider application.




From this central concept, some typical distributed systems issues rise. One issue is about **data representation**, it should be independent of individual local systems so that received data can be understood on any kind of node. Another issue is about **publishers' identification** by requestors, this will encompass the publisher's node address or DNS name, and also, the resource itself within that node address.

# Web Services – data representation

As it is, the web services concept is very wide. As far as the HTTP protocol is fulfilled and respected, all kind of information exchanges between applications can be implemented as web services.

Concerning data representation, the most widely used solution is extensible markup language (XML), another alternative is JavaScript Object Notation (JSON).

Both represent data in a human readable text format, and yet also suitable for automated parsing. In each case the appropriate content-type specification should be used, correspondingly, **application/xml** and **application/json**.

Also, some higher-level standards have been established on more details about how applications can communicate through web services, two examples are SOAP (Simple Object Access Protocol) and XML-RPC (Remote Procedure Calls in XML format through HTTP).

Due to the client-server model, implicit by HTTP, requestors must know where to find publishers, and then, what services are provided by that publisher.

# Web Services – resources identification

Resources are identified by an URL, an URL identifies a resource, and also, how to access it. So, an URL starts by an access protocol name, for web services **http://** or **https://**, then it identifies the node's address, usually through a DNS host's name. Optionally it may also specify a port number preceded by a colon. If the port number is not specified, then the protocol's default port number is assumed.

This is called the **origin** part of the URL. The remaining part of the URL identifies the resource within that origin, it starts by a slash and may reflect an internal hierarchical resources organization with names separated by a slashes.

Regarding the origin part, it shouldn't be hardcoded into applications because it depends on the running environment, they should be provided to applications as runtime configuration data. Each resource's local identification within the origin, on the other hand, may be hardcoded into requestor applications.

Usually, requestors known what resources are provided by a publisher, nevertheless, standards have been established on how publishers can inform requestors about that. Web Services Description Language (WSD) and Universal Description, Discovery, and Integration (UDDI) make that information available to requestors in XML format.

## 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 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.

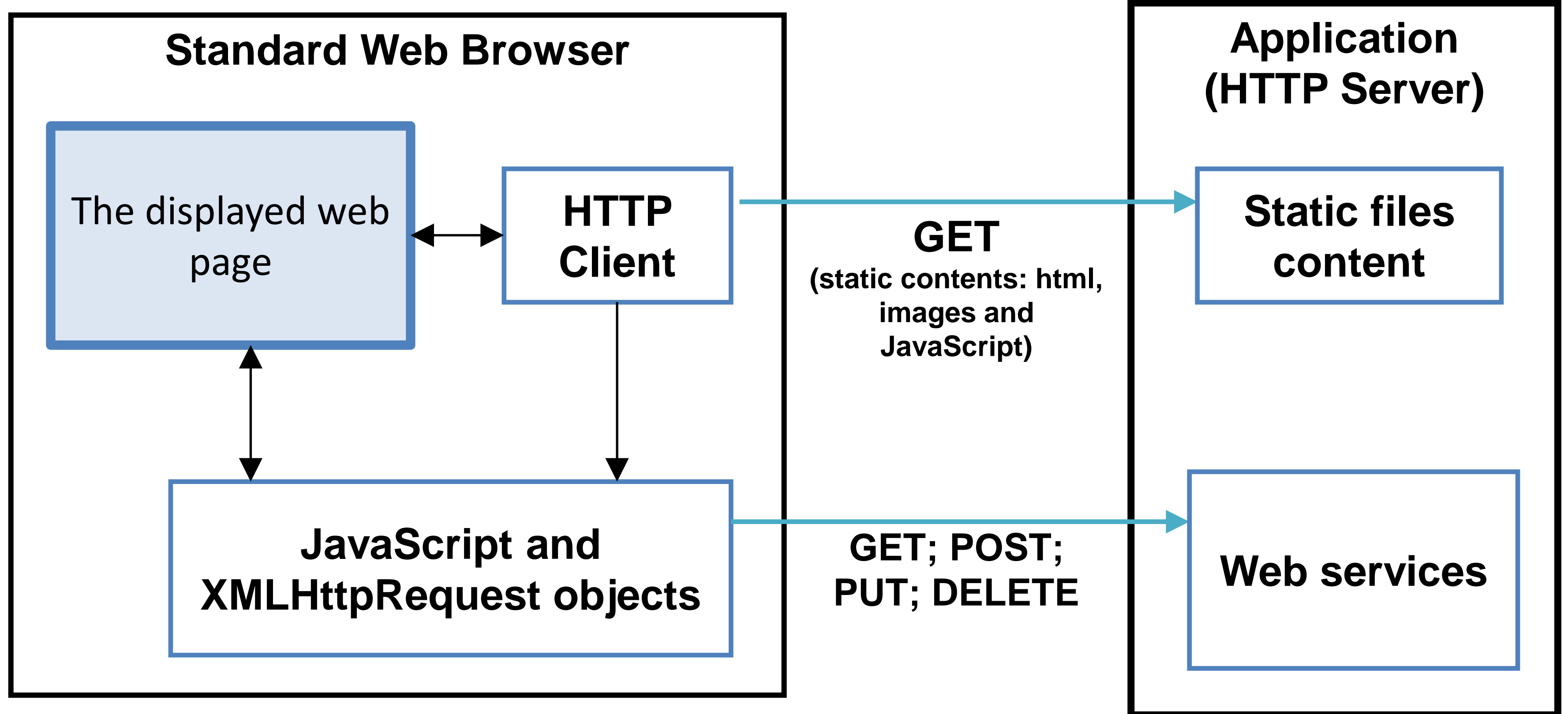
# Web browsers as consumers - JavaScript

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). |

# The AJAX 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.

# An AJAX example

This example is a very simple application for online voting. The purpose is making available an **online voting service**.

Voting is completely free and out of control, any one is allowed to cast as many votes as wanted in whatever candidates.

The number of candidates is established on the backend configuration and candidates are simply named as **Candidate 1**, **Candidate 2**, …

The backend is composed by a single BASH application, mapped to the **/cgi-bin/votes** URI, the application accepts two types of requests:

- **PUT /cgi-bin/votes?c=N** – casts a vote on candidate number N.
- **GET /cgi-bin/votes** – returns a set of HTML tags making a content that will show the current vote standings, together with buttons to cast a vote for each candidate.

# AJAX example – the main HTML page

```
<!DOCTYPE html>
<html>
<head><title>Voting demo</title>
<script src="voting.js"></script>
</head>
<body bgcolor=#C0C0C0 onLoad="refreshVotes()"><h1>Voting demo - SCOMRED 2020/2021</h1>
<center><hr />
<div id=votes>
Please wait ...
</div>
</center><hr />
</body></html>
```

# AJAX example – Frontend (voting.js)

```
function refreshVotes() {
        var request = new XMLHttpRequest();
        request.onload = function upDate() {
                document.getElementById("votes").innerHTML = this.responseText;
                setTimeout(refreshVotes, 1500);
                };
        request.ontimeout = function timeoutCase() {
                document.getElementById("votes").innerHTML = "Still trying ...";
                        setTimeout(refreshVotes, 1000);
                };
        request.onerror = function errorCase() {
                document.getElementById("votes").innerHTML = "Still trying ...";
                        setTimeout(refreshVotes, 1000);
                };
        request.open("GET", "/cgi-bin/votes", true);
        request.timeout = 5000;
        request.send();
        }

function castVote(option) {
        var request = new XMLHttpRequest();
        request.open("PUT", "/cgi-bin/votes?c=" + option , true);
        request.send();
        }
```

# AJAX example – the backend application

```
#!/bin/bash
N_CANDIDATES=8;
STANDING_FILE_BASENAME=/tmp/.voting.demo
if [ "$REQUEST_METHOD" == "GET" ]; then
        echo "Content-type:text/plain"
        echo ""
        for((i=1;i<=N_CANDIDATES;i++));do
            STANDING_FILE=${STANDING_FILE_BASENAME}.$i
            if [ ! -f $STANDING_FILE ]; then echo "0" > $STANDING_FILE; fi
            echo "<p>Candidate $i - $(cat $STANDING_FILE) votes <input type=button value=VOTE onClick='castVote(${i})'>"
        done
        exit
fi
if [ "$REQUEST_METHOD" == "PUT" -a "${QUERY_STRING#c=}" != "$QUERY_STRING" ]; then
        echo ""
        N_CAND="${QUERY_STRING#c=}"
        STANDING_FILE=${STANDING_FILE_BASENAME}.$N_CAND
        VOTES=$(cat $STANDING_FILE)
        VOTES=$((${VOTES}+1))
        echo "$VOTES" > $STANDING_FILE
        exit
fi
echo "Status: 400 Bad Request“
echo “”
```

# AJAX example – Final look

# Concurrent access issues

In the lab class, this very simple example is going to be implemented and tested, at first glance it may seem to be working pretty fine, nevertheless is has some flaws one of which is the total absence of concurrent access control.

As with any network service, in a real environment, a web service will often have several clients accessing it at the same time. We can't make those clients wait in a queue, they are simply not willing to wait. So, such clients are all served at the same time by using multiple processes or multiple threads.

Having several clients being served at the same time, each by a dedicated process or thread is perfectly ok, and has several advantages, however, it's a fact that must be taken into account when implementing the service. And in our last example it wasn't.

Having several clients being served at the same time means several clients may access the same resource (e.g., a file) at the same time. This is a concurrent access to the resource, and becomes a big problem if such accesses encompasses changing the resource (write operations), the result will be simply unpredictable and thus it's a scenario to avoid.

# Concurrent access control

It’s ok to allow concurrent access to a resource, as far as all accesses are read operations, and thus the resource is completely static and never changes. Once write operations are involved, then concurrent access must be avoided at all cost.

One approach is ensuring each client being served uses its specific resources not shared with others. That approach was used in the previous class’s **/var/www/cgi-bin/hashCalculate** web service:

```
#!/bin/bash
### I’m in file /var/www/cgi-bin/hashCalculate
###
M_CONTENT_FILE=/tmp/.scomred-hash-calculate.$$.tmp
###
(...)
```

The trick is on the red line, because the $$ sequence is expanded by the shell to the current process’s PID, we have the guarantee each client will be using a different filename to temporarily store the request’s content.

In today’s example, this method is not an option because clients have to update a shared resource: the current voting standings, implemented as a set of files, one for each candidate. For these cases, a mechanism known a **lock** or **mutex** must be used.

# Concurrent access control – lock/mutex

A **lock** or **mutex** has two states (**free** or **acquired**) and the acquire operation is mutually exclusive (from where the **mutex** term comes). Being mutually exclusive means if several different entities try to acquire it, there's the guarantee only one will be successful, the lock is then acquired until the lucky guy frees it.

For a **lock** to be effective, it must be used (acquired) by everyone before trying to access the shared resource, once the access ends the lock has to be freed, otherwise it becomes indefinitely inaccessible to everyone.

Most programming languages and operating systems provide locking mechanisms, like for instance semaphores. In the Java language, each class and each object has an intrinsic lock that may be acquired by using the **synchronized** declaration.

One popular way to implement a lock over a filesystem is through the folder creation operation, if several applications try to create a same folder at the same time, only one will be successful, the others get an error because the folder already exists. In our lab class this technique will be used.