Web Services

AJAX

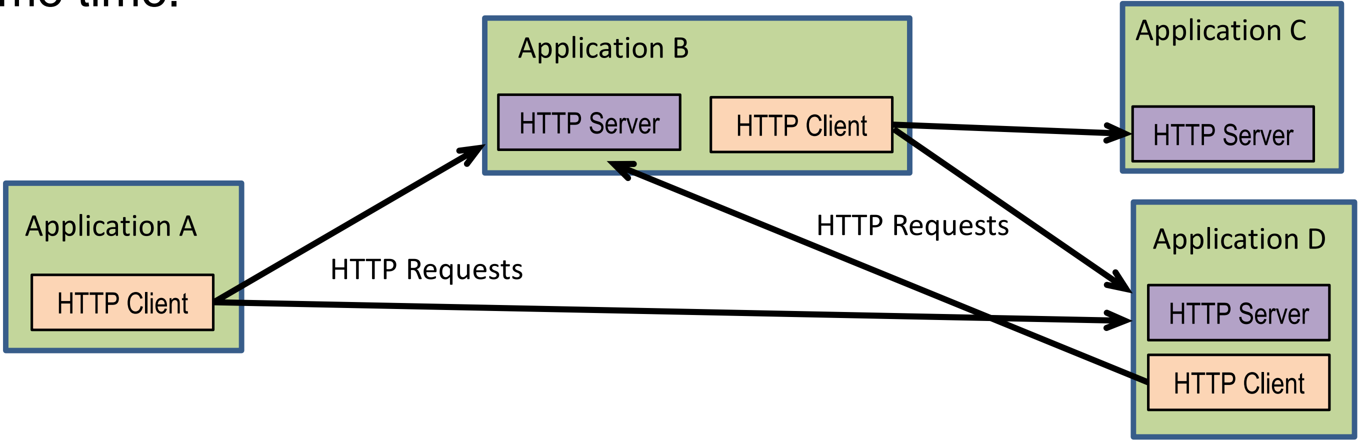
SCOMRED, November 2018

HTTP/HTTPS as general purpose application protocol

Designing and implementing a new application layer protocol for some 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.

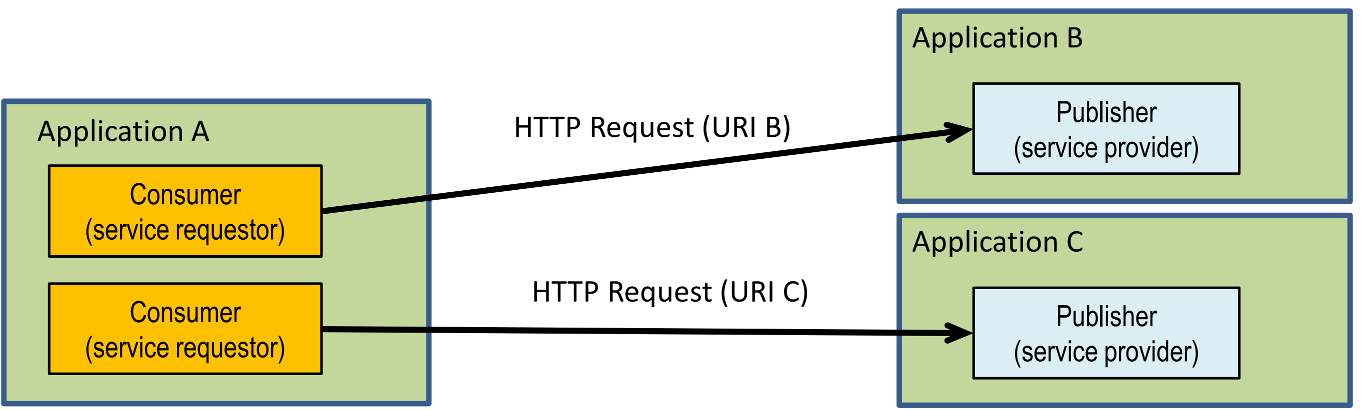
We must bear in mind this may not be the best technical option, however, it may be the best option under investment point of view. Some adaptations to the original protocol usage may be required because it was designed with a different purpose, nevertheless, the protocol specification itself must be 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 (**service requestor or consumer**) assumes the HTTP client's role and the other (**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, that 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 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.

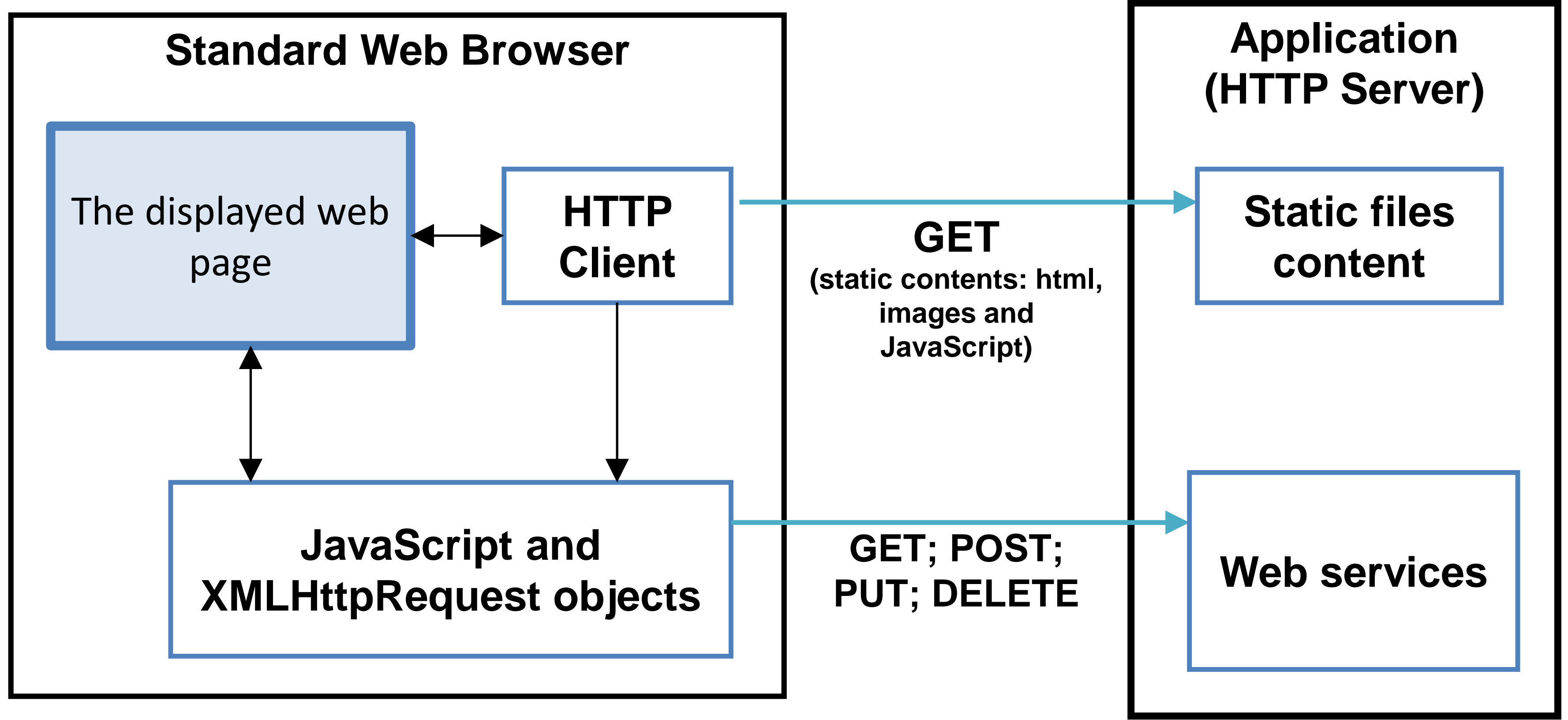
Web browsers as consumers - JavaScript

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.

AJAX example (consumer side)

Another team is developing an HTTP application server, this application is a online voting system, there are only two options for voting: A or B.

The HTTP server is capable of providing static contents from files and also capable of acting as a web services provider for some specific requests:

PUT /votes/A – casts a vote on option A (neither the request nor the reply have content).

PUT /votes/B – casts a vote on option B (neither the request nor the reply have content).

GET /votes – returns an HTML table with the current voting standings.

Your team's mission (*in case you accept it!*) is creating web contents (files) to be provided by the server so that by using a Web Browse users can cast votes and also have a permanently updated view of current voting standings.

The current vote standings presented to the user should be kept up to date, for that, we can use a timer for a periodic refresh of that element it the web page.

AJAX example – HTML (/index.html)

```
<!DOCTYPE html>
<html>
<head><title>Voting demo</title>
<script src="voting.js"></script>
</head>
<body bgcolor=#C0C0C0 onload="refreshVotes()"><h1>Voting demo</h1>
<center><hr />
<p><input type=button value="Vote for A" onClick="castVote(A)"></p>
<p><input type=button value="Vote for B" onClick="castVote(B)"></p>
<hr />
<div id="votes">
Please wait, loading voting results ...
</div>
</center><hr />
</body></html>
```

The **refreshVotes()** function is called once the content is loaded, it will have the mission of updating the current vote standings (presented on the <div> element with id votes) by sending a **GET /votes** request to the server. The function must be periodically run to reflect any change on vote standings.

The **castVote()** function is triggered by the user click on buttons to cast votes, it will send a **PUT /votes/A** or **PUT /votes/B** to request to the server.

AJAX example – JavaScript (/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", "/votes", true);
        request.timeout = 5000;
        request.send();
        }
```

This function is triggered by the HTML page loading, after creating the **XMLHttpRequest** object, it settles call-back functions for (onload, ontimeout and onerror), then defines the request (**open()** method), a timeout and sends it (**send()** method).

Most important: notice that the next execution of this function is scheduled only once there's a result for the current request, this ensures at every time there is only one pending request and auto adapts the requests rate to the server and the network performance.

AJAX example – JavaScript (/voting.js) (Cont.)

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

This function is triggered by the user click on a button, the voting option is passed as argument and added to the request.

In this case the PUT request has no body (content), otherwise it could be specified as argument of the send() method.

No call-back functions are defined for the request. This means whether requests are successful or not it's completely ignored. Of course, this could be improved.

The page could be loaded locally without the server running, but these functions can only be tested once the other team has the server ready and running to provide the required web services.

Still, once the other team announces their server is ready, we should test if it's behaving accordingly to the project's requirements before testing it with our page and functions. This will exclude issues related to the server side implementation.

Web services testing – Postman

In standard web browsers, when a URL is manually typed, the browser always assumes the method to use is GET. So a web browser is not a suitable tool to impersonating consumer applications and test web services.

Applications generically called **postman** do the trick, they are able to generate HTTP requests with any method, also setting HTTP headers and the message's content as required. In addition, they also provide extensive information about the response received from the provider.

Postman is an essential tool when developing web services, by testing them, the developer assures himself they are working properly before they are actually used by real consumer applications.

Postman is often used to manually perform functional tests, but it may also be programmed through scripts to automatically perform sets of **unity tests**, thus ensuring web services are kept in conformity during development.

Several more or less sophisticated versions of postman are freely available, some even run on standard web browsers as plugins or extensions.