HTTP Request Handlers

As detailed in the HTTP Request Handlers **Introduction** chapter, Wakanda allows you to call server-side JavaScript functions named "Request handlers" by sending an HTTP request that fits a specific pattern (JavaScript regex). Request handler functions are defined in the .js file that is passed as a parameter to the **addHttpRequestHandler( )** method. This function must have the following type of signature:

```
function myHandler(request, response) {
    return "Hello world";
}
```

The function that is called handles two parameters and can return a value:

- *request*: Contains an *HTTPRequest* object, which is a representation of the current request. This parameter provides all the information about the handled request so that you can analyze it fully.
- *response*: Contains an *HTTPResponse* object, which is a representation of the response that will be sent back. You can use this parameter in the function to write information in the header or status line of the response.
- *return*: The result returned by the function, if any, in the *return* parameter is set to the 'body' property of the *HTTPResponse* object.

Usually, you do not need to modify the *HTTPResponse* object directly (partially or in its entirety). The *return* parameter lets you return the appropriate values in the 'body' field and the response will be formatted automatically. You can also change the contents of a single field as seen in the following example:

```
function test(request, response) {
    return 'Communication with server working!';
    response.contentType = 'text/plain';
}
```

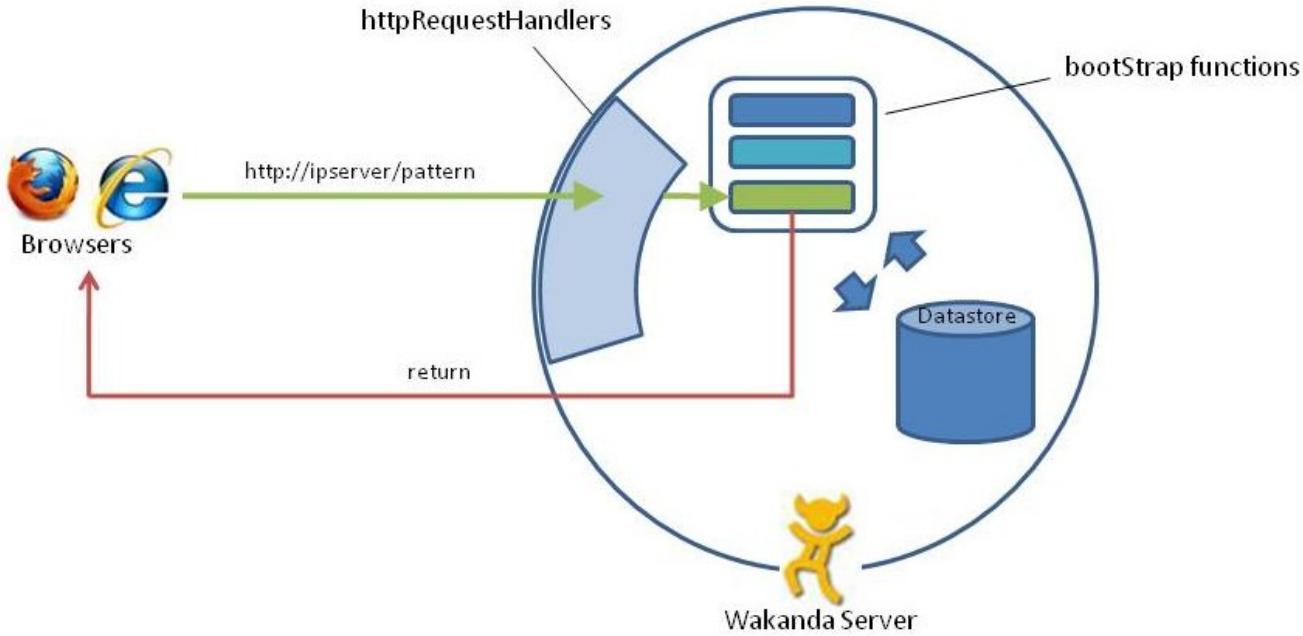
In advanced cases, you may want to modify the *response* object directly using properties and methods in the *HTTPResponse* class.

Introduction

About HTTP Request Handlers

With Wakanda, you can call a server-side JavaScript function by sending an HTTP request that fits a specific pattern (JavaScript regex).

Here is how it works: the client sends an order to execute the function using a REST type request. On the server side, when the request pattern is detected, it triggers a call to the HTTP request handler installed by the corresponding **addHttpRequestHandler( )** method. This method executes the function and returns its result to the client. The following figure shows how the handlers work:



Unlike server-side JavaScript functions that are called using RPC services (see **Using JSON-RPC Services**), code executed using an HTTP request handler does not require the WAF library to be initialized or loaded on the client. The only requirement is for the request to be properly formatted. This means that it is particularly suited for creating "services" on the Wakanda server, giving HTTP clients access to the Web site data.

Also note that you can change the code in your JavaScript functions without restarting the application. This feature can be especially useful during the development phase.

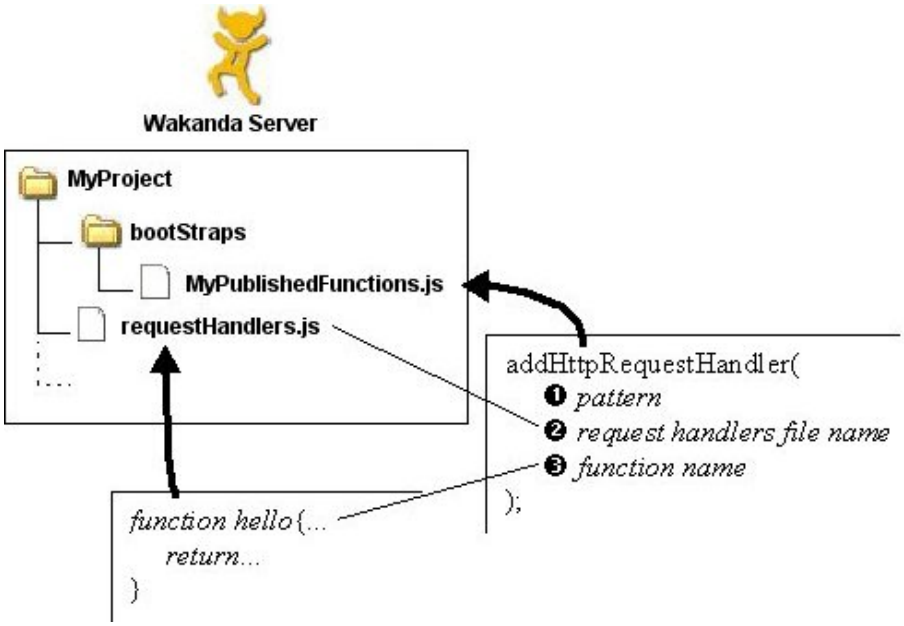
Note: Wakanda provides several other ways to execute server-side JavaScript code. For more information, refer to the Executing code on the server section.

Server Configuration

To be able to call server functions using an HTTP request handler on the server you must:

- Designate the files in the project containing request handler function calls (**addHttpRequestHandler( )**).
- Install and configure each HTTP request handler (pattern, file name, function name).

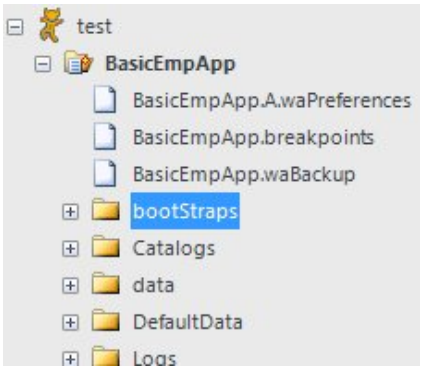
The following architecture is required on the server for setting up and enabling HTTP request handlers:



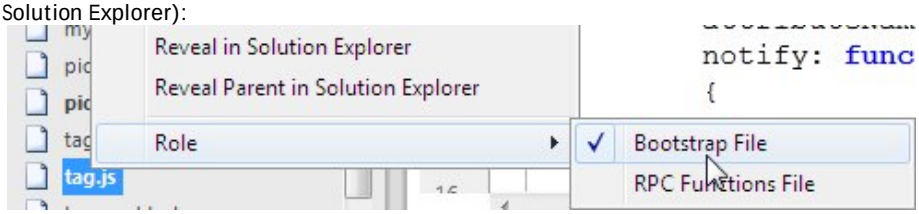
Designating bootStrap Files

There are two ways to explicitly designate the file(s) containing HTTP request handlers on the server:

- Create a folder named **bootStraps** at the project root and store the .js files containing the HttpRequestHandler methods in it.



- Assign the **BootStrap File** role to one or more JavaScript files in your project (right-click on the file in Wakanda Studio's Solution Explorer):



The filename becomes bold when you assign this role to it. You can designate several JavaScript files in the project (even if stored in different locations) as bootStrap files.

When the project is launched, all JavaScript files designated as bootStraps (integrated in the bootStraps folder and/or defined as bootStrap files) are automatically loaded and the **addHttpRequestHandler( )** methods are parsed. As a result, if you change the contents of the bootStraps files (for example if you add a request handler), you will have to restart the server in order to force the parsing of the files.

However, you can modify function files that are called by the request handler without having to restart the server.

Installing a Request Handler

You can create a request handler by adding a call to the **addHttpRequestHandler( )** method in one of the bootStrap files in the project (see previous section).

Each call must be in the following form:

```
addHttpRequestHandler(
      '/pattern',              // (string) regex used to filter the requests to be intercepted
      'requestHandlers.js',  // (string) name of the file where the request handler function is s
      'hello'         // (string) name of the request handler function
);
```

This code can be interpreted as follows: when the client sends a request of type **http://{ project IP address }:{ project port}/pattern**, Wakanda Server executes the function named “hello” in the “requestHandlers.js” file.

In the second parameter, you must pass the pathname of the file containing the request handler. If you only pass a file name (as in the example above), the file must be located at the project root.

Calling a Request Handler from a Client

On the client, you call request handler functions by simply sending an HTTP request that is formatted as follows:

```
http://{ project IP address }:{ project port }/pattern
```

This request allows any HTTP client to retrieve data from Wakanda Server without having to install or initialize libraries. In principle, the client does not even need to use JavaScript.

Executing code using a request handler is particularly suited for setting up custom "services" that use a specific API. For example, a "catalog" type Wakanda application could be queried as a service and return information concerning prices or product availability.

HTMLForm

An *HTMLForm* object is created by the **parts** property of an **HTTPRequest**. This object gives access to the list of uploaded parts from an HTTP client in the context of a multipart form.

count

Description

The **count** property returns the total number of parts uploaded by the HTTP client.

encoding

Description

The **encoding** property returns the encoding used for the request parts.

boundary

Description

The **boundary** property returns the boundary tag value used to delimit the parts in the form.

[n]

Description

The **[n]** property gives access to the n^{th} part of the *HTMLForm*. This part is an object of the **HTMLFormPart** class for which you have properties and a method.

HTMLFormPart

HTMLFormPart objects are the individual parts of a multipart form request. These objects are accessible through the following syntax:

```
request.parts[n]
```

... where *n* is the part number in the request.

name

Description

The **name** property returns the name of the input field used for the POST of the binary data.

fileName

Description

The **fileName** property returns the name of the uploaded file.

mediaType

Description

The **mediaType** property returns the value of the part's "content-type" field.

size

Description

The **size** property returns the size of the body in bytes.

asText

Description

The **asText** property returns the body of the part as a Text value. If the body contents do not match the String type, an *Undefined* value is returned.

asPicture

Description

The **asPicture** property returns the body of the part as an *Image* value if possible. If the body contents do not match the image type, an *Undefined* value is returned.

asBlob

Description

The **asBlob** property returns the body of the part as a *BLOB* value regardless of the actual data type in the body (text, image, or any other data type).

save( )

void **save**(String *filePath* [, Boolean *overWrite*])

Parameter	Type	Description
filePath	String	Path of the destination file
overWrite	Boolean	true = overwrite the destination file if it already exists

Description

The **save( )** method saves the body of the part in the file whose path is passed in *filePath*.

If *filePath* describes a full path including a filename, the given name is used for the file. Otherwise, if *filePath* only describes a folder path, the original filename (returned by the **name** property) is used.

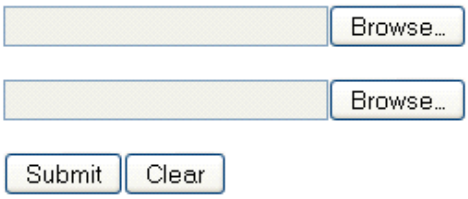
If the *overWrite* parameter value is set to true, the destination file is replaced if it already exists. If it is set to false or is omitted, the **save( )** action is ignored if the destination file already exists.

Example

The following requestHandler function displays the parts posted by a simplified HTML form:

```
function displayFormContent (request, response)
{
    var i;
    var result;

    result = 'request.parts.count: ' + request.parts.count;
    result = result + '\nrequest.parts.encoding: ' + request.parts.encoding;
    result = result + '\nrequest.parts.boundary: ' + request.parts.boundary;
    result = result + '\n-------------------------------------';

    for (i = 0; i < request.parts.count; ++i) {
        result = result + '\nrequest.parts[' + i +'].name: ' + request.parts[i].name;
        result = result + '\nrequest.parts[' + i + '].fileName: ' + request.parts[i].fileName;
        result = result + '\nrequest.parts[' + i + '].mediaType: ' + request.parts[i].mediaType;
        result = result + '\nrequest.parts[' + i + '].size: ' + request.parts[i].size;
        result = result + '\nrequest.parts[' + i + '].asText: ' + request.parts[i].asText;
        result = result + '\n-------------------------------------';

        request.parts[i].save('e:/Data/', true);
    }

    return result;
}
```

The HTML form appears as shown below:



Here is the HTML form code:

```
<form method="post" action="/displayFormContent" enctype="multipart/form-data">
<p><input type="file" name="fileBlob1" size="25"></p>
<p><input type="file" name="fileBlob2" size="25"></p>
<p><input type="submit" value="Submit"><input type="reset" value="Clear"></p>
</form>
```

HTTPRequest

An *HTTPRequest* object describes an incoming HTTP message (a request).

It is mainly made up of a request-line, a header (a *HTTPRequestHeader* object) and, optionally, a body that can even be empty.

Here is an example of a basic request:

GET / HTTP/1.1[CRLF]	request-line (HTTP verb + URL + version)
Host: 127.0.0.1[CRLF]	Headers (Host is mandatory in HTTP 1.1)
User-Agent: XXX[CRLF]	
[CRLF]	Body (empty in this case)

url

Description

The **url** property returns the URL of the current HTTP request. The URL is decoded, i.e., characters like %3D, for example, are decoded.

rawURL

Description

The **rawURL** property returns the raw URL of the HTTP request.

urlPath

Description

The **urlPath** property returns the path part of the request.

For example, if the **url** is:

```
/index.html?test=1
```

... the **urlPath** property will return:

```
index.html
```

urlQuery

Description

The **urlQuery** property returns the query part of the request.

For example, if the **url** is:

```
/index.html?test=1
```

... the **urlQuery** property will return:

```
test=1
```

host

Description

The **host** property returns the request's "Host" header.

method

Description

The **method** property returns the HTTP method name of the request ("GET", "POST", "HEAD"...).

version

Description

The **version** method returns the version of the HTTP protocol used for the request. Usually, it will be 1.1.

user

Description

The **user** property returns the user name when the request requires authentication.

password

Description

The **password** property returns the user password when the request requires authentication. This property is filled with BASIC authentications only.

requestLine

Description

The **requestLine** property returns the request-line as it was received by the server.

body

Description

The **body** property returns the body part of the message.

It can be of different types:

- TEXT: a *String* value is returned
- IMAGE: an *Image* value is returned
- BINARY: a *BLOB* value is returned

The type of the returned value is automatically defined depending on the Content-Type of the request (see the **contentType** property).

headers

Description

The **headers** property returns an *HTTPRequestHeader* object, containing the header of the request.

For more information on *HTTPRequestHeader* objects, refer to the **HTTPRequestHeader** class description.

contentType

Description

The **contentType** property returns the contents of the request's 'content-type' header field.

It is actually a shortcut for retrieving the following value:

```
var contents = request.headers['content-type']
// or
var contents = request.headers.CONTENTS_TYPE
```

parts

Description

The **parts** property gives access to the different parts of the **body** (for multipart forms).

For more information, refer to the **HTMLForm** class description.

addHttpRequestHandler( )

void **addHttpRequestHandler**(String *pattern*, File | String *file*, String *functionName*)

Parameter	Type	Description
pattern	String	Pattern of the request to intercept
file	File, String	File in which the handler function is defined
functionName	String	Name of the function to handle the request matching the pattern

Description

The **addHttpRequestHandler( )** method installs a request handler function on the server. Once installed, this function will

intercept and process any incoming HTTP request matching a predefined pattern.

- In the *pattern* parameter, pass a string describing the HTTP requests that you want to intercept. This pattern should be defined through a JavaScript Regex (Regular expression). For more information, see the following paragraph.
- In the *file* parameter, pass the file containing the function to call for this handler. You can pass either a File object reference or a string containing the name of the bootStrap file. The file can be located in a folder named **bootStraps** at the project root or be a file to which the **Bootstrap file** role has been assigned.
- In the *functionName* parameter, pass the name of the request handler function to call when it matches the *pattern*. This function will receive two object parameters (*request* and *response*) and can return a value.

For a complete description of the server-side HTTP request handlers feature, refer to the **HTTP Request Handlers** documentation.

Defining the Pattern Parameter

The *pattern* parameter sets the requests to be intercepted and processed using the HTTP request handler.

To define this parameter, you must use a JavaScript Regex (Regular expression). Here are a few principles for pattern definitions that are generally used in Web applications:

- **^/myPattern$**: intercepts requests containing only the "/myPattern" pattern, for example

  ```
  GET http://mydomain.com/myPattern
  ```

  . "/files/myPattern" or "/myPattern/myvalue" type requests are not taken into account.
- **^/myPattern/**: intercepts requests that begin with the "/myPattern" pattern. This pattern intercepts any requests beginning with "/myPattern/myfoldr" as well as requests like "/myPattern/myfile.html" or "/myPattern/myfile.js".
- **^/myPattern[?]***: intercepts requests that begin with the "/myPattern" pattern and that contain, optionally, a query string. You can use this pattern to handle requests containing "/myPattern?Name="Martin", and so on.
- **/myPattern**: intercepts any requests containing /myPattern regardless of its position in the string. This pattern will accept indifferently requests such as "/files/myPattern", "/myPattern.html", and "/myPattern/bar.js".

For more information about regular expressions in JavaScript, refer to the documentation available on the Internet:
http://www.regular-expressions.info/javascript.html
https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/RegExp

Example

If you write in the bootStrap.js file:

```
addHttpRequestHandler('(?i)^/doGetStuff$', 'myFile.js', 'myFunction');
```

... the *myFunction* request handler will be called each time the server receives a query containing the "/doGetStuff" URL.

HTTPRequestHeader

An *HTTPRequestHeader* object describes the header of an **HTTPRequest** object, which is an array of name/value pairs. Accessing a value can be done with the following syntax:

```
value = header['name'];
```

Keep in mind that an *HTTPRequestHeader* object is a read-only object. You cannot write:

```
header['name'] = "value";   // Not allowed
```

In Wakanda, you can access the most frequently used header fields in HTTP clients through attributes. Therefore, attributes such as **ACCEPT** and **ACCEPT_CHARSET** correspond to "Accept" and "Accept-Encoding" headers.

For detailed information on HTTP header fields, refer to the RFC 2616 and (for cookie management) RFC 2109.

ACCEPT

Description

ACCEPT_CHARSET

Description

ACCEPT_ENCODING

Description

ACCEPT_LANGUAGE

Description

AUTHORIZATION

Description

CACHE_CONTROL

Description

CONTENT_LENGTH

Description

CONTENT_TYPE

Description

COOKIE

Description

EXPECT

Description

FROM

Description

HOST

Description

IF_MATCH

Description

IF_MODIFIED_SINCE

Description

IF_NONE_MATCH

Description

IF_RANGE

Description

IF_UNMODIFIED_SINCE

Description

KEEP_ALIVE

Description

MAX_FORWARDS

Description

PRAGMA

Description

PROXY_AUTHORIZATION

Description

RANGE

Description

REFERER

Description

TE

Description

USER_AGENT

Description

HTTPResponse

An *HTTPResponse* object describes an outgoing HTTP message (a response).

It is mainly made of a status-line (statusCode + explanation message), a header (a *HTTPResponseHeader* object) and, optionally, a body.

An *HTTPResponse* object has several attributes as well as specific methods.

statusCode

Description

The **statusCode** property allows you to set the response's status code to return.

By default, the status *200* (OK) is returned.

For more information about status codes, refer to the RFC 2616.

body

Description

The **body** property sets the body part of the returned message.

This property corresponds to the value returned by the **return** statement in the handler function.

The type of the value is defined by the Content-Type of the response (see the **contentType** property).

Example

This HTTP request handler function:

```
function test(request, response) {
    response.body = '{"good":"job"}';
    response.contentType = 'application/json';
    response.headers['Warning'] = "I'm watching you"; // custom header
    response.statusCode = '200';
}
```

... sends back the following HTTP response:

```
HTTP/1.1 200 Ok
Content-type: application/json
Content-Length: 14
Warning: I'm watching you

{"good":"job"}
```

contentType

Description

The **contentType** property allows you to set the contents of the response's "content-type" header field, which is actually a shortcut to the following attribute:

```
response.headers['content-type']
     // or
response.headers.CONTENTS_TYPE
```

headers

Description

The **headers** property contains the *HTTPResponseHeader* object of the response.

For more information on *HTTPResponseHeader* objects, refer to the **HTTPResponseHeader** class description

allowCache( )

void **allowCache**(Boolean *useCache*)

Parameter	Type	Description
useCache	Boolean	true = use Wakanda's cache, otherwise false (default)

Description

The **allowCache( )** method indicates if the contents of the *HTTPResponse* should be cached on the server. This method overrides the default setting for the current *HTTPResponse* object.

Pass **true** in *useCache* to load the contents to the server cache and **false** to leave the cache untouched. By default, the cache is not used. This setting is defined in your Wakanda Project's Settings file.

allowCompression( )

void **allowCompression**(Number *minThreshold*, Number *maxThreshold*)

Parameter	Type	Description
minThreshold	Number	Minimum size (in bytes) below which the response should not be compressed or -1 to use default value
maxThreshold	Number	Maximum size (in bytes) up to which the response should not be compressed or -1 to use default value

Description

The **allowCompression( )** method sets custom compression thresholds for the *HTTPResponse*.

Defining compression thresholds is useful to avoid wasting server time by compressing data that is either too small or too large. The cache is used, for example, by the jsLoader to send .css or .js files in an optimized mode.

This method allows you to override the server's default settings, which are defined at the project level. Its scope is the *HTTPResponse* object. Default values are (in bytes):

- *minThreshold*: 1024
- *maxThreshold*: 10*1024*1024 (10 MB)

You can use default values by passing -1 in *minThreshold* or *maxThreshold*.

sendChunkedData( )

void **sendChunkedData**(String | Image | BLOB *data*)

Parameter	Type	Description
data	String, Image, BLOB	Data to send

Description

The **sendChunkedData( )** method sends an *HTTPResponse* in chunks without knowing in advance the size of the *data*.

Pass in *data* each chunked value to send. The content-type of the response will be set automatically by the method depending on the type of the *data* (TEXT, IMAGE, or BLOB).

Example

This basic example illusrates how you can send chunked data:

```
function myChunkedData(request, response)
{
    var i = 0;
    var result = "";

    for (i = 0; i < 10000; ++i) // preparing 10000 chunks
    {
        var timer = new Date()
        result = result + "chunk# " + i.toString() + " - " + timer.toGMTString() + "\n";
        if ((i % 10) == 0)
        {
            response.sendChunkedData(result);
            result = "";
        }
    }

    response.sendChunkedData (result);
}
```

HTTPResponseHeader

An *HTTPResponseHeader* object describes the header of an **HTTPResponse** object, which is an array of name/value pairs.

Since this array is available in read/write mode, you can write:

```
response.header['name'] = 'value'; // Set value
value = response.header['name']; // Get value
```

In Wakanda, you can access the most frequently used header fields by HTTP clients through attributes. Attributes, such as ACCEPT_RANGES and ETAG, correspond to "Accept-Range" and "Etag" headers. For detailed information on HTTP header fields, refer to the RFC 2616 and (for cookie management) RFC 2109.

The following attributes are available:

ACCEPT_RANGES
AGE
ALLOW
CACHE_CONTROL
CONNECTION
DATE
ETAG
CONTENT_ENCODING
CONTENT_LANGUAGE
CONTENT_LENGTH
CONTENT_LOCATION
CONTENT_MD5
CONTENT_RANGE
CONTENT_TYPE
EXPIRES
LAST_MODIFIED
LOCATION
PRAGMA
PROXY_AUTHENTICATE
RETRY_AFTER
SET_COOKIE
VARY
WWW_AUTHENTICATE
X_STATUS
X_POWERED_BY
X_VERSION