

HTTP Request Handlers

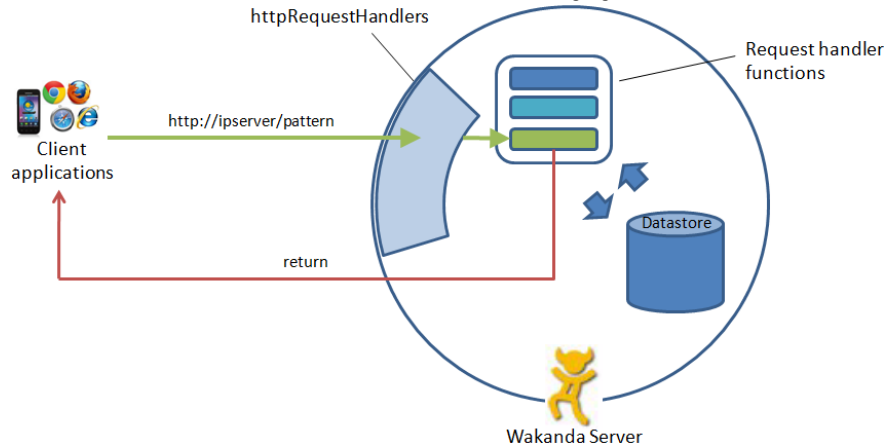
As detailed in the [Introduction to HTTP Request Handlers](#) chapter, Wakanda allows you to call server-side JavaScript functions named "Request handlers" by sending an HTTP request that fits a specific pattern (JavaScript regex).

Introduction to HTTP Request Handlers

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 install a HTTP request handler on the server, you just need to execute the `addHttpRequestHandler()` method with the appropriate parameters. To install automatically HTTP request handlers at the project launch, you can use the `bootStraps` folder.

Installing a Request Handler

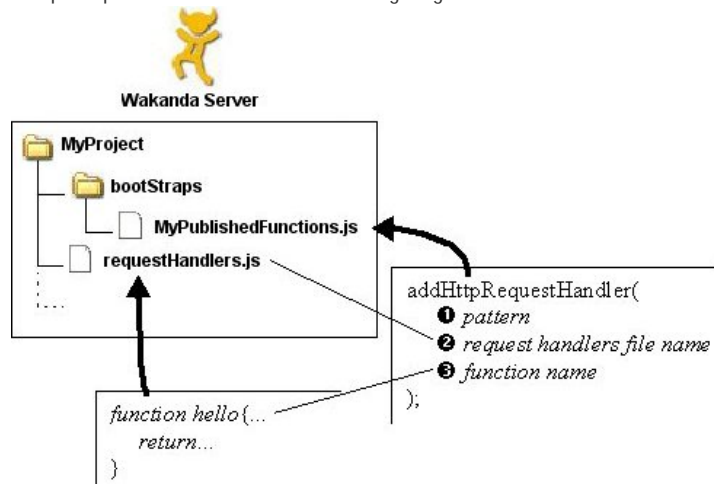
You install a request handler by calling the `addHttpRequestHandler()` method. It defines the pattern to intercept as well as the function to call in response and its location.

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 specified  
    '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.

This principle is illustrated in the following diagram:



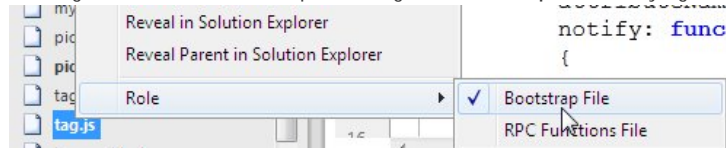
For more information, refer to the `addHttpRequestHandler()` method description.

Using Bootstrap File(s)

If you want to install HTTP request handlers automatically when the project is launched, you can put the `addHttpRequestHandler()` calls in a **Bootstrap file**.

When the project is launched, all JavaScript files designated as Bootstrap files are automatically loaded and the `addHttpRequestHandler()` methods are parsed. As a result, if you change the contents of a Bootstrap file (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.

To designate a file as a Bootstrap file, assign the **Bootstrap File** role by right-clicking 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.

Defining a Request Handler Function

Request handler functions are defined in the .js file that is passed as a parameter to the `addHttpRequestHandler()` method. These functions 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) {  
    response.contentType = 'text/plain';  
    return 'Communication with server working!';  
}
```

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

Autocompletion in the Handler Function

If you want to benefit from a comprehensive autocompletion feature on both the *request* and *response* objects within the handler function in the Wakanda Studio Code Editor, just add the following jsdoc tags before the call:

```
/**  
 * @param {HttpRequest} request  
 * @param {HttpResponse} response  
 */  
function test(request, response) {  
    response.contentType = 'text/plain'; //here you will have autocompletion  
    return 'Communication with server working!';  
}
```

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.

To generate and handle this kind of request from a button on the client side using JavaScript, you can use for example the `XMLHttpRequest()` method.

Executing code using a request handler is particularly suited for setting up custom "web 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.

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.CONTENT_TYPE
```

parts

Description

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

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

remoteAddress

Description

The `remoteAddress` property returns a string representation of the remote IP address of the client that sent the HTTP request.

The property will return either an IPv6 address or an IPv4 address. IPv6 addresses consist of eight groups of four hexadecimal digits separated by colons, for example 2001:0db8:85a3:0042:0000:8a2e:0370:7334, while IPv4 addresses consist of 32 bits, which may be divided into four octets, written in decimal numbers ranging from 0 to 255, for example 192.168.2.1.

The `remoteAddress` property will return a string representing either an IPv6 or an IPv4 address depending on both your hardware configuration and your Wakanda Server settings:

- If your network infrastructure supports hybrid dual-stack IPv6/IPv4 implementation, the result returned depends on the "Listening IP Address" project settings (see [Project Settings File](#)):
 - If your server listens on **Any** addresses, the property will always return an IPv6 address:
 - a standard IPv6 address if the client address is IPv6,
 - an **IPv4-mapped IPv6 address** if the client address is IPv4. In such addresses, the first 80 bits are zero, the next 16 bits are one, and the remaining 32 bits are the actual IPv4 address. For example, ::ffff:192.168.2.1 represents the IPv4 address 192.168.2.1. This feature allows the server to be able to handle all protocols. For more information on mapped addresses, please refer to the following article on Wikipedia: http://en.wikipedia.org/wiki/IPv6#IPv4-mapped_IPv6_addresses.
 - If your server only listens on an IPv4 address, the property will always return an IPv4 address. Wakanda server will not respond to clients with IPv6 addresses.
 - If your server only listens on an IPv6 address, the property will always return an IPv6 address. If the client address is IPv4, an IPv4-mapped IPv6 address is returned (see below for more information on IPv4-mapped IPv6 addresses).
- If your network infrastructure is IPv4-only or IPv6-only, the server will only be able to respond to clients with addresses of the matching protocol (mapping between IPv4 and IPv6 addresses is done at the network system level), regardless of the "Listening IP Address" Wakanda Server settings.

Note that IPv6-only configurations are supported but not recommended. They should be reserved for very specific configurations only.

The following table summarizes the `remoteAddress` values depending on the project settings and IP configurations:

- If the "Listening IP Address" project settings is on **Any**:

	Server Dual Stack	Server IPv4 only	Server IPv6 only
Client IPv4	IPv4-mapped IPv6	IPv4	ignored
Client IPv6	IPv6	ignored	IPv6

- If the "Listening IP Address" project settings is on a **single IPv4(*) address**:

	Server Dual Stack	Server IPv4 only	Server IPv6 only
Client IPv4	IPv4	IPv4	n/a
Client IPv6	ignored	ignored	n/a

(*) Listening on a single IPv6 address is supported but not recommended.

Example

In this simple example, we want to return and display (client-side) the IP address of the client sending the request.

1- On the server side, we create the necessary files for the HTTP request handler.

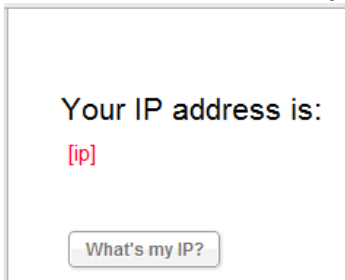
- a "boot.js" with the **Bootstrap File** role to load the handler:

```
//boot.js bootstrap file
addHttpRequestHandler('/testIP', 'handler.js', 'testIP');
```

- the "handler.js" file will only return the request IP address:

```
//handler.js file
function testIP(request, response)
{
    var ip = request.remoteAddress; // gets the remote client address
    return ip;
}
```

2- On the client side, we build a very simple interface containing a rich text area and a button:

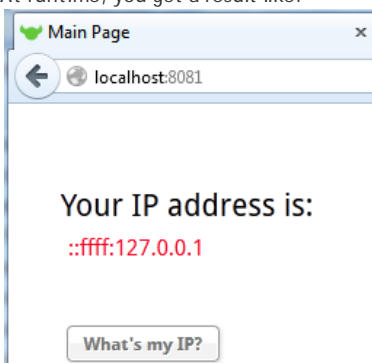


- We created a variable `datasource` named `ip` and defined it as the Source property of the area.
- The code for the **What's my IP?** button is:

```
button1.click = function button1_click (event){
var xhr=new XMLHttpRequest(); // create the xhr object
    xhr.onreadystatechange = function (){
        if (xhr.readyState == 4 && xhr.status == 200) { // server answered
            ip = xhr.responseText; // assign a global variable
            sources.ip.sync(); //synchronize local datasource
        }
    }

    xhr.open('GET','/testIP',true); //call the testIP handler
    xhr.send(); //Send the XHR
};
```

At runtime, you get a result like:



remotePort

Description

The `remotePort` property returns the port used by the remote client that sent the HTTP request.

Example

In this simple example, we want to return and display (client-side) the port used by the client.

1- On the server side, we create the necessary files for the HTTP request handler.

- a "boot.js" with the **Bootstrap File** role to load the handler:

```
//boot.js bootstrap file
addHttpRequestHandler('/testPort', 'handler.js', 'testPort');
```

- the "handler.js" file will return the request port:

```
//handler.js file
function testPort(request, response)
{
    var port = request.remotePort; // gets the remote client port (number)
    return port+""; // add an empty string to return a string
}
```

2- On the client side, we build a very simple interface containing a rich text area and a button:

Your port is:

[port]

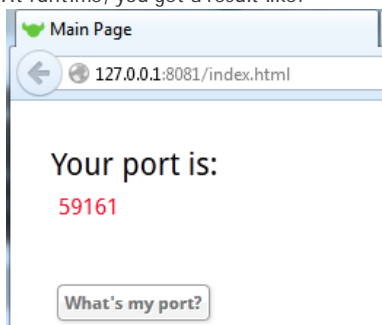
What's my port?

- We created a variable `datasource` named `port` and defined it as the Source property of the area.
- The code for the **What's my port?** button is:

```
button1.click = function button1_click (event){
var xhr=new XMLHttpRequest(); // create the xhr object
    xhr.onreadystatechange = function (){
        if (xhr.readyState == 4 && xhr.status == 200) { // server answered
            port = xhr.responseText; // assign a global variable
            sources.port.sync(); //synchronize local datasource
        }
    }

    xhr.open('GET', '/testPort', true); //call the testPort handler
    xhr.send(); //Send the XHR
};
```

At runtime, you get a result like:



localAddress

Description

The `localAddress` property returns a string representation of the local IP address used by Wakanda Server to handle the HTTP request.

The `localAddress` property will return a string representing either an IPv6 or an IPv4 address depending on both your hardware configuration and your Wakanda Server settings. For more information, please refer to the method description.

localPort

Description

The `localPort` property returns the port used by the Wakanda Server to handle the HTTP request.

isSSL

Description

The isSSL returns true if the current HTTP request came in through an SSL connection (HTTPS); otherwise, it returns false.

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.

In this example, we want the request handler function to return binary data in a Blob and force the client to download it as a text file:

```
function doReturnBinary(request, response) {
    var vBuffer = new Buffer("sample string for Blob object"); // we need to use a Buffer to write data
    var vBlob = vBuffer.toBlob(); // converting to Blob
    response.contentType = 'application/octet-stream';
    response.headers['content-disposition'] = 'attachment; filename=test.txt'; // header to force download
    response.body = vBlob; // return the Blob data
}
```

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"}
```

Example

The following function:

```
function doReturnImage(request, response) {
    var varImage = application.loadImage(application.getFolder().path+'WebFolder/'+myPict.png);
    return varImage;
}
```

... is equivalent to:

```
function doReturnImage(request, response) {
    var varImage = application.loadImage(application.getFolder().path+'WebFolder/'+myPict.png);
    response.body = varImage;
}
```

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.CONTENTES_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

Example

We want to the request handler function to return binary data in Blob and force the client to download it as a text file:

```
function doReturnBinary(request, response) {
  var vBuffer = new Buffer("sample string for Blob object"); // we need to use a Buffer to write data
  var vBlob = vBuffer.toBlob(); // converting to Blob
  response.headers.CONTENT_TYPE = 'application/octet-stream';
  response.headers['content-disposition'] = 'attachment; filename=test.txt'; // header to force download
  response.body = vBlob; // return the Blob data
}
```

allowCache()

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

Parameter	Type	Description
<code>useCache</code>	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
<code>minThreshold</code>	Number	Minimum size (in bytes) below which the response should not be compressed or -1 to use default value
<code>maxThreshold</code>	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 (see [Text Compression](#)). Its scope is the `HTTPResponse` object. Default values are (in bytes):

- `minThreshold`: 1024
- `maxThreshold`: $10 \times 1024 \times 1024$ (10 MB)

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

sendChunkedData()

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

Parameter	Type	Description
<code>data</code>	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 illustrates 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);
}
```

Example

As *File* objects inherit from *Blob* class, you can directly send a file as an HTTP response using the `sendChunkedData()` method:

```
function myFunction(request, response)
{
    var foo = File("c:/pages/alert.html");
    response.sendChunkedData(foo);
}

addHttpRequestHandler('/admin', 'startup.js', 'myFunction');
```

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

addHttpRequestHandler()

```
void addHttpRequestHandler( String pattern, String filePath, String functionName )
```

Parameter	Type	Description
pattern	String	Pattern of the request to intercept
filePath	String	Path to the 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 Regex (Regular expression). For more information, see the following paragraph.
- In the *filePath* parameter, pass a string containing the path to the file that has the function to call for this handler. You can pass either an absolute path or a path relative to the project folder (POSIX syntax).
- 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*, object of the `HttpRequest` type
 - *response*, object of the `HttpResponse` typeThe function also can return a value by using the `return` statement.

For a complete description of the server-side HTTP request handlers feature, refer to the [Introduction to HTTP Request Handlers](#) section.

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 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/myfolder"` 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"`.

Wakanda server supports ICU's Regular Expressions package. For more information, please refer to the following address:

<http://www.icu-project.org/userguide/regexp.html>

For more information about Regex in general, please refer to the following address:

http://en.wikipedia.org/wiki/Regular_expression

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 (or `"/DoGetStuff"`, `"/dogetstuff"`, etc.).

Example

You will find a full example of a http request handler used for sending emails in the following section: [Writing and Sending Emails from the Client Side \(Detailed Example\)](#)

removeHttpRequestHandler()

```
void removeHttpRequestHandler( String pattern, String filePath, String functionName )
```

Parameter	Type	Description
pattern	String	Request pattern to remove
filePath	String	Path to the file in which the handler function is defined
functionName	String	Name of the function to handle the request matching the pattern

Description

The `removeHttpRequestHandler()` method uninstalls an existing HTTP request handler function running on the server. The request handler should have been installed using the `addHttpRequestHandler()` method.

For a complete description of the server-side HTTP request handlers feature, refer to the [HTTP Server Request Handlers](#) documentation.