Using JSON-RPC Services

About JSON-RPC for Wakanda

What is JSON-RPC?

Wakanda controls the execution of server-side JavaScript using the JSON-RPC (Remote Procedure Call) protocol.

Here is how it works: a client connects using the JSON-RPC service available on the server. This service consists of an HTTP query that is sent as a JSON object. It executes a method (procedure) that returns a response from the server (if any). The server then sends its response to the client in the form of a JSON object. For more information about this protocol, please refer to the JSON-RPC page on Wikipedia.

Within a Wakanda application, executing server-side code using the JSON-RPC protocol can be associated, for example, to the script of a button. This can be useful for:

- importing or exporting data to and from your application by using text files or Web services,
- recording information into a log file,
- performing calculations on the data as a whole,
- and much more.

Note: Wakanda provides other ways to execute server-side JavaScript code:

- *sending a REST request intercepted by an addHttpRequestHandler( ) (see HTTP Request Handlers) or*
- *executing a Datastore class method.*

Scope of RPC calls

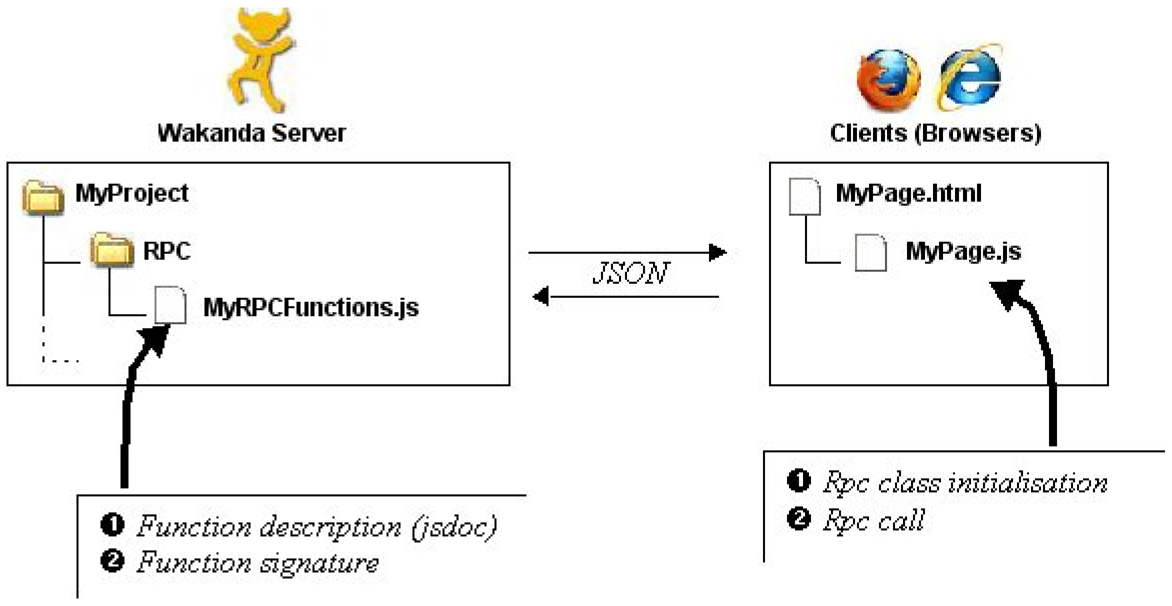
Server functions executed using RPC can access all the JavaScript modules implemented in Wakanda's SSJS API: **Datastore**, Files, Storage, etc.

These functions can access datastore classes in the application directly through the Datastore API (attributes, properties, etc.).

However, server methods cannot directly access JavaScript objects that are specified on the clients, in particular, those using Widgets. Passing and returning parameters is the only direct communication between methods executed by using RPC and the clients.

Architecture

In order to implement JSON-RPC services in Wakanda, specific files and code must present on both the server-side (Wakanda Server) and the client-side (pages displayed in the browser). This architecture is summarized below:



These elements are detailed in the following pages.

Setting up Wakanda Server for JSON-RPC

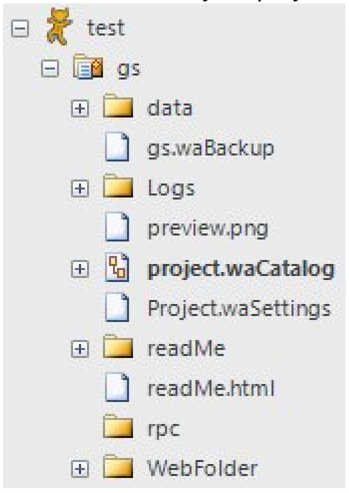
To call methods using RPC on the Wakanda server, you must do the following server-side:

- Specify files within a project that contain JavaScript functions that can be called using JSON RPC and
- Use JSDOC comments to describe JavaScript methods inside the function files.

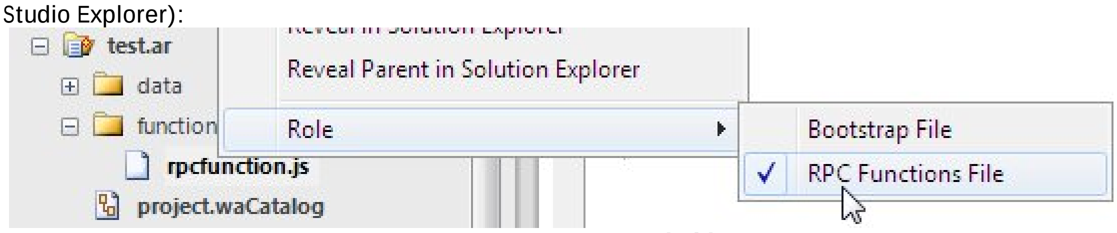
Declaring function files

On the server, there are two ways to explicitly designate the file(s) containing methods that will be available as JSON-RPC services for clients:

- Store the .js files containing these methods in a folder named **rpc**, which is created at the root of your project. Functions found in your project's **rpc** folder are automatically available for clients through RPC services.



- Assign the **RPC Functions File** role to a JavaScript file in the project (right-click on the file in the Wakanda Studio Explorer):



When you assign this role to the file, its name is displayed in **bold**. You can designate several JavaScript files as RPC function files in different locations in the project. The server will load the contents of each file.

Configuring RPC functions

For security and performance reasons, JavaScript functions specified on the server cannot be executed using RPC by default. You must explicitly define the server JavaScript functions as "published" in RPC as follows:

- Each function must be included in a JavaScript file stored in the current "rpc" folder or designated as an "RPC Function File" (see previous paragraph).
- Each function must be described (parameters and return) in the form of "JSDOC" comments using the following keywords:

```
/**
* @param {type of first parameter} param1
* @param {type of second parameter} param2
...
* @return {type of variable returned}
**/
```

  This description allows Wakanda to interpret and transmit the data correctly between the server and the clients when communicating using JSON.
- The function signature must be written just after its JSDOC description.
  Here is an example of a function definition:

```
/**
  * Comments for myRPCFunction
  * @param {number} param1
  * @param {string} param2
  * @return {string }
  **/
  function myRPCFunction(param1, param2) {
  ...
  }
```

All defined functions are published by the Wakanda server and can be called from the client. The Wakanda catalog of RPC functions is compliant with the following format: http://jsonrpc.org/spec.html

Note that the server automatically publishes a second version of the function with the "Async" suffix added to its name. This name should be called to execute the function in asynchronous mode. For example, the function defined above will automatically be published with both names **myRPCFunction** (standard synchronous calls) and **myRPCFunctionAsync** (asynchronous calls). For more information, please refer to the **Calling methods in synchronous mode** and **Calling methods in asynchronous mode** paragraphs.

Calling Methods Using an RPC

There are two steps for executing a server-side method from a client using an RPC:

- Initializing the connection and instantiating the **rpc** class.
- Calling the method (in synchronous or asynchronous mode).

Initialization and configuration

To facilitate calls to methods using an RPC, Wakanda provides an interface where you can show the RPC methods on the client side and therefore access them directly in the code. There are two ways you can take advantage of this interface:

- Use the document properties in the GUI Designer (standard mode)
- Call the initialization code in your script (advanced mode).

Calling methods in synchronous mode

Calling an RPC method in synchronous mode is a simple and quick way to execute code on the server but it has a few drawbacks.

When you call a server-side JavaScript method in **synchronous mode**, script code execution on the client is suspended at the location where the RPC request is sent while waiting for the response from the server. This makes sure that the value returned by the function is available in the rest of the client-side script code. In this case, server-side processing must be quick so as to avoid unduly freezing the HTML page. This point is discussed in the paragraph **Synchronous or asynchronous execution?**

When an error occurs, the server sends a JavaScript exception. You must manage this exception in a "try catch" structure.

You call a method using an RPC in synchronous mode by entering the function name directly in the script of the HTML page:

```
//Call in synchronous mode
var result = myMethods.functionName(params);
```

You use the *params* argument to send RPC function parameters. This argument is described in the **params parameter** section.

Calling methods in asynchronous mode

When you call RPC methods in asynchronous mode, server-side code execution is independent from code execution on the client machine. Once the execution request has been sent by the client, the script continues to execute without waiting for any response from the server. This point is discussed in the paragraph **Synchronous or asynchronous execution?**

When a server response is returned, the callback function specified during the RPC method call is then called. You must process the result in this callback function. The callback function can also be called when an error occurs if you have specified an 'onerror' function (standard call).

There are two ways to write a call to an RPC method in asynchronous mode:

- the standard call, where you pass a complete structure to the RPC method.
- the simplified call, where you only pass a callback function.

params parameter

Regardless of how the function is called, you can send one or more values to the RPC function using the *params* parameter.

- If only one parameter is expected, you can pass it directly in *params* as a variable or a literal value.
- If you are sending several parameters, you must organize them as an array in the order they were specified in the JSDOC of the function on the server side (see the **Configuring RPC functions** section). You can send the name of the array or send its values directly between brackets [ ].
- All the parameters declared are optional during a call; you can pass as many of them as you need to (they are processed in the order they are declared).
- If the mechanism for validating parameters is activated, the -32602 exception is generated when an error occurs (see **Error Handling**). For more information about this mechanism, refer to the description of the **validation** option in **Initialization using the GUI Designer (standard mode)**.

Error Handling

If an error occurs when you call a method using an RPC, the Wakanda server returns to the client a JavaScript exception to the RPC client. This exception can be processed if the RPC call has been carried out:

- in synchronous mode in a "try catch"
- in standard asynchronous mode (in this case, the exception is sent to the function specified by the 'onError' keyword).

For more information, refer to the **Calling Methods Using an RPC** section.

List of errors

An exception returned by Wakanda contains three properties:

- **code**: the error number
- **name**: a character string characterizing the error
- **message**: a text describing the error

For reasons of better readability and code density, you can choose to intercept errors based on their code or name (see the examples below).

Here is a list of errors processed by the RPC client of Wakanda:

code	name	message	Comment
-32601	MethodNotFoundError	Method not found	The RPC method called does not exist/is not available on the server
-32602	InvalidParamsError	Invalid params	The number or type of the parameters is incorrect
-32603	InternalError	Internal error	JSON-RPC internal error

Example

Example of an RPC call in synchronous mode to the myRpcFunction function with error processing based on the error name (which makes the code easier to read):

```
try{
    myRpcFunction(42);
} catch (e) {
    switch(e.name) {
        case 'InternalError' :
            console.log(e.message);
            break;
        case 'InvalidParamsError' :
            console.log(e.message);
            break;
        case 'MethodNotFoundError' :
            console.log(e.message);
            break;
        default:
            console.log(e.message);
            break;
    }
}
```

Note: The last case (default) lets you process any errors sent directly by sources other than the RPC client.

Example

Same example as above for an RPC call in synchronous mode to the myRpcFunction function but this time with error processing based on the error code:

```
try{
    myRpcFunction(42);
} catch (e) {
    switch(e.code) {
        case '-32603' :
            console.log(e.message);
            break;
        case '-32602' :
            console.log(e.message);
            break;
        case '-32601' :
            console.log(e.message);
            break;
        default:
            console.log(e.message);
            break;
    }
}
```

Note: The last case (default) lets you process any errors sent directly by sources other than the RPC client.