

Using JSON-RPC Services

About JSON-RPC Services in Wakanda

Overview

Wakanda allows you to execute server-side JavaScript functions from any client application. In Wakanda, this feature is built upon two standard technologies:

- **CommonJS modules** on the server side to write code that you want to make available to clients,
- **JSON-RPC** (Remote Procedure Call) protocol to handle the communication between the server and the clients. For more information about this protocol, please refer to the [JSON-RPC](#) page on Wikipedia.

Within a Wakanda application, executing server-side code using RPC can be associated, for example, with 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 various other ways to execute server-side JavaScript code:

- *sending a **HTTP REST** request intercepted by an `addHttpRequestHandler()` (see [HTTP Request Handlers](#)) or*
- *[Using Datastore Class Events](#) or [Using Datastore Class Methods](#).*

Architecture

JSON-RPC services in Wakanda are based upon the [CommonJS](#) architecture:

- on the Wakanda server, each RPC function(s) .js file must be placed in the **Modules** folder; within a module file, each RPC function is declared using a standard CommonJS syntax:

```
exports.myRPCFunction = function(){...}
```

- to allow clients access to an RPC function, the function must be declared in the `.waPerm` file of the project;
- on the client side, RPC modules must be declared in each client page.

These elements are detailed in the following pages.

Compatibility Note: If you were using JSON-RPC services in Wakanda v1 projects, you need to edit your projects so that they can work with Wakanda v2 and higher. For more information, please refer to the [Upgrading JSON-RPC from Wakanda v1 to v2 and higher](#) section.

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.

Configuring CommonJS Modules for RPC

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

- Write JavaScript functions that can be called by JSON RPC using CommonJS architecture and gather these functions into .js module file(s) located in the **Modules** folder of the project.
- Designate each RPC module/function to be available client-side and assign appropriate permissions.

Creating RPC function modules

On the Wakanda Server, functions that will be available as JSON-RPC services must follow the CommonJS modules architecture (for more information about this architecture, please refer to the [CommonJS specification](#)):

- a CommonJS module is a JavaScript file (one file per module) containing one or more functions to export. Wakanda modules must be stored in a **Modules** folder located at the root of the application folder.

For example, a "myRPC.js" file must be placed at the following location:

```
{project folder}/Modules/myRPC.js
```

You can add subfolders in the 'Modules' folder. In this case, you must pass pathnames when declaring the module.

- each RPC function must be declared using the **exports** keyword. For example, if you want to provide the following function through RPC:

```
function add (a, b) {  
    return (a + b);  
}
```

You must include it in the *myRPC.js* module file as follows:

```
exports.add = function add (a, b) {  
    return (a + b);  
};
```

You can add one or more functions in a single module file. You will be able to assign permissions to either the whole module file, or separately to each function (see the following paragraph).

Note: If you add several functions with the same name in separate .js files, only one of them will be available on the client side.

You can use any type of parameter that is accepted by JSON: String, Boolean, Number, Null, Array and Object. You can use ISO8601 to format dates.

Namespace declarations

The namespace avoids naming conflicts between different DOM functions when executed client-side. If you do not define specific namespaces for your RPC methods, by default the RPC module file name will be used as the namespace (see [Note for advanced users](#)).

Otherwise, you can declare several namespaces for your RPC methods: you just need to add them to the RPC module file. A RPC methods file can contain a single top level method declaration and multiple namespace declarations.

The following namespace declarations are supported:

```
// Creating dataTools namespace functions in an object  
var dataTools = {  
    createData : function () {},  
    loadData : function () {},  
    saveData : function () {}  
}
```

```
};

// Adding functions to mathTools namespace
var mathTools = {};
mathTools.sinh= function () {};
mathTools.cos = function () {};
mathTools.atan = function () {};
```

Publishing RPC functions


For security and performance reasons, by default JavaScript functions included in the **Modules** folder on the server cannot be executed using RPC on the client side(*). You must explicitly allow client access and assign appropriate group permissions for each module or function in the **.waPerm** file of the project (for more information about this file, please refer to the [Assigning Group Permissions](#) section).

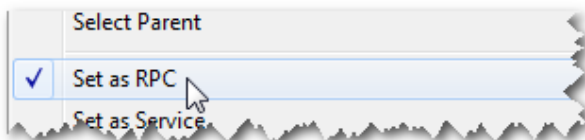
(*) On the server side, RPC module functions can be executed as any CommonJS module function using the **require()** method. For example, you can write:

```
require('myRPC').add() //SSJS code
//execute the add function from the myRPC CommonJS module
```

Defining permissions

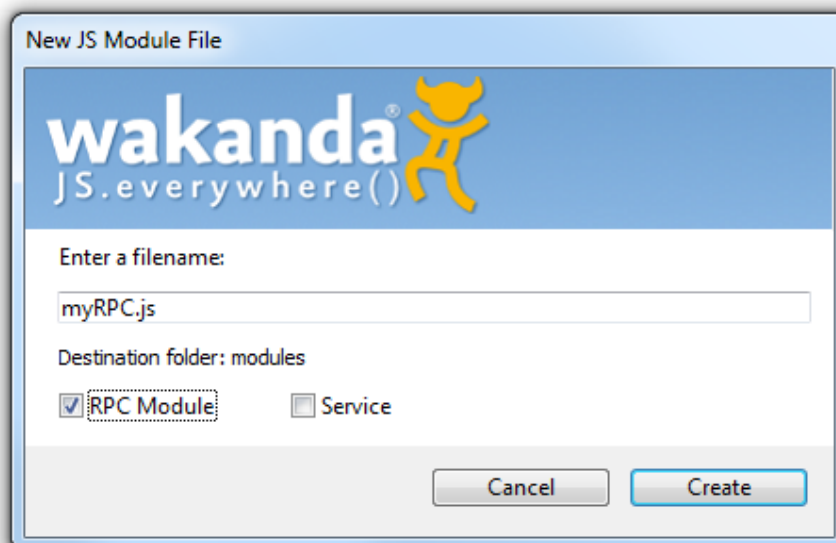
Basic permissions are automatically added to the *<permissions>* element of the **.waPerm** file of the project when you set a file as an RPC file. There are several ways that you can set an RPC file in Wakanda Studio:

- Select any **.js** file in the **Modules** folder at the root of your project and select **Set as RPC(*)** in the contextual menu (or in the global menu ):

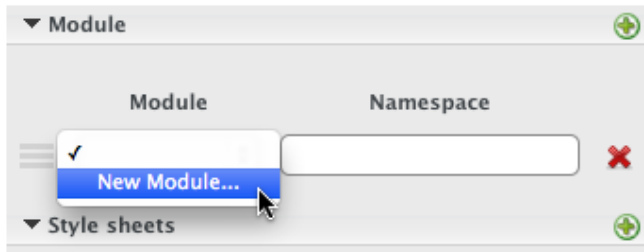


(*) Unchecking this item will cause the corresponding entry in the **.waPerm** file to be removed.

- Select **New>JS Module** in either the **File** menu, the **New** toolbar icon, or the **Project explorer** contextual menu and check **RPC Module** in the file creation dialog box:



- In the **GUI Designer**, create a new RPC file by selecting **New Module...** in the **Module** area dropdown menu:



Whichever way you choose to create or define the file, a new entry is added to the `.waPerm` file. In this entry, the module is declared and client access is allowed, for example:

```
<allow type="module" resource="myRPC" action="executeFromClient" />
```

If you want to restrict the access to a group or define only a function, you need to edit the `.waPerm` file (see below).

Permissions settings

Permission actions are defined within the `<permissions>` element of the `.waPerm` file of the project:

- Available permissions for a module:

```
<allow type="module" resource="{module path}" action="executeFromClient" [groupID="{groupID}"] />
<allow type="module" resource="{module path}" action="promote" [groupID="{groupID}"] />
```

- Available permissions for a module function:

```
<allow type="function" resource="{module path}/{function name}" action="executeFromClient" [groupID="{groupID}"] />
<allow type="function" resource="{module path}/{function name}" action="promote" [groupID="{groupID}"] />
```

- If you want to publish a module or a function module without client authentication, just omit the `groupID` tag.

Here is the description of each attribute:

Attribute	Values	Description
type	"module" or "function"	Defines permission for an entire RPC module (all functions defined in the RPC module can be called client-side) or for a function (only defined functions can be called)
resource	"{module path}" or "{module path}/{function name}"	The <i>{module path}</i> attribute can be the name of the module file (without the <code>.js</code> extension) if it is located at the root of the modules folder. If the file is included in subfolders, you need to pass a path relative to the modules folder.
action	"executeFromClient" or "promote"	executeFromClient allows the function or module functions to be executed from the client side. promote executes the function or module functions with specific access permissions (for more information about this permission action, please refer to the "Class Methods" paragraph in the Assigning Group Permissions section)
groupID	UUID of a group	(optional) Only users that belong to the defined group can execute the RPC function(s). If you want to publish a module or a function module without client authentication, just omit the <code>groupID</code> attribute

Examples:

```
// Publish the whole "myRPC" module only for one group
<allow type="module" resource="myRPC" action="executeFromClient" groupID="3C9...
// Publish the "add" function of the "myRPC" module and only for one group
<allow type="function" resource="myRPC/add" action="executeFromClient" groupID="3C9...
// Publish the whole "myFreeRPC" module without permissions
<allow type="module" resource="myFreeRPC" action="executeFromClient" />
```

```
// Publish the "test" function from "mySubRPC" module located in the "RPC"  
<allow type="module" resource="RPC/mySubRPC/test" action="executeFromClient"/>
```

Note: In future versions of Wakanda, you will be able to edit permission actions using menu commands in Wakanda Studio.

Published functions

All functions defined in the `.waPerm` file are published by the Wakanda server and can be potentially called from clients belonging to the appropriate group (or all clients if no group ID is defined). Note that the server automatically publishes a second version of each 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 Using an RPC](#) section.

Note: RPC functions code is loaded by the Wakanda server at startup. Each time you modify the contents of a RPC module, you need to restart the server so that edits are available on the client side.


Calling Methods from the Client Side

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

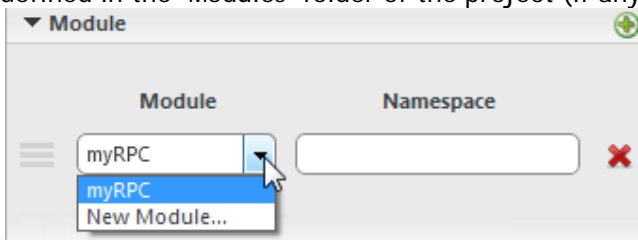
- Declaring the modules/functions you want to make available in the client interface.
- Calling the method (in synchronous or asynchronous mode).

Declaration and configuration

You can configure RPC methods on the client side quickly and easily using the GUI Designer of Wakanda Studio. You just need to declare RPC module files containing the RPC functions to be available from the Page, as well as their namespace.

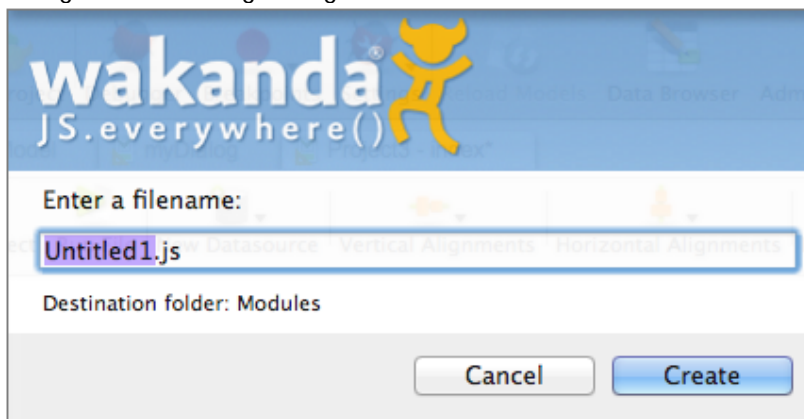
In the GUI Designer, select the **document** object and then display its properties. In the "Module" area (empty by default), click on the  icon to add a new empty entry in the area.

- **Module:** the "Module" dropdown menu allows you to select one of the RPC module files already defined in the "Modules" folder of the project (if any) or to create a new module:



Note: For more information on how to set a RPC module file, please refer to the [Publishing RPC functions](#) paragraph.

If you select **New Module...**, you can create and declare automatically a new RPC module file through the following dialog:

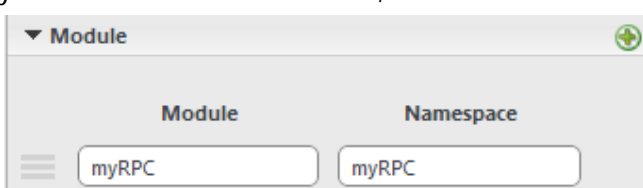


Enter a valid name and click **Create**. An empty module file is then created in the "Modules" folder at the root of your project (the folder is created if necessary).

- **Namespace:** enter the *namespace* for your RPC methods. By default, once it has been selected, it contains the **Module** file name. Use this default value if you did not define custom namespaces for your RPC methods. Otherwise, enter the namespace(s) defined in the module file for your methods (see [Namespace declarations](#)).

Note: Loading RPC methods into the global space is not recommended.

For example, if you selected "myRPC" as the Module name, Namespace will contain "myRPC" by default. If you leave this value untouched, and have a function named *add()*, you can call it by writing *myRPC.add()*.



Note that the RPC module name is written without the ".js" extension.

Once this option has been set, Wakanda adds the code necessary for RPC class instantiation and the initialization of the client-side interface. You can then call RPC methods directly in your code, provided

they are prefixed with the namespace that has been set.

Note for advanced users

Activating RPC in the document adds a <meta> tag to the header of the HTML document. This tag takes the following form:

```
<meta name="WAF.config.rpc.file" content="rpc-proxy/{RPCFileName}?namespace={RPCNamespace}"/>
```

For example:

```
<meta name="WAF.config.rpc.file" content="rpc-proxy/myRPC?namespace=myRPC" />
```

The value of the "content" attribute indicates each module file name and namespace declared in the GUI Designer. Since the presence of this tag automatically triggers RPC class instantiation in the page, make sure that you do not move or modify it.

If you want to call a Wakanda RPC method from any external Page, you just need to add a "script" tag to the HTML page. For example:

```
<bad>
```

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.

Async suffix

You call a method using an RPC in asynchronous mode by adding the **Async** suffix to the method name in the script of the HTML page. For example, a "myRpc()" function should be called "myRpcAsync()" in the client code. This tells the server that the call is asynchronous and that function result should be passed to the client using a callback. Without the **Async** suffix, the remote procedure function will be executed synchronously on the server, and no callback will be sent to the client.

Note: The "Async" suffixed version of each RPC function defined on the server is automatically exposed by Wakanda server on the client side. You do not have to create specific function versions.

Standard calls

In the case of a standard call, you must pass, as a parameter to the function, a structure containing one or two callback functions as well as arguments.

- "onSuccess" function: is called when all goes well and receives the execution result.
- "onError" function (optional): is called when an error occurs and receives the description of the JavaScript exception. If this function is called, the "onSuccess" function does not receive anything (and vice versa). For more information about JavaScript exceptions generated by RPC services, refer to the [Error Handling](#) section.

The call must be in the form:

```
moduleName.functionNameAsync( {  
    'onSuccess': function (result) {  
        ...  
    },  
}
```



```

        'onError': function (error) {
            ...
        },
        'params': [arg1, arg2...]
    });

```

The *functionName* function must be found among the RPC functions of the project RPC modules designated on the server (see the [Configuring CommonJS Modules for RPC](#) section) and declared in the page. The *onSuccess* and *onError* functions must be set in the script. You use the *params* argument to send RPC function parameters. This argument is described in the [params parameter](#) section.

Simplified calls

Simplified calls of an RPC function in asynchronous mode differs from [Standard calls](#) by the fact that instead of passing a complete structure, you only pass a callback function followed by arguments to the RPC function as a parameter.

The quick call is in the form:

```

moduleName.functionNameAsync(
    function (result) {
        ...
    },
    params
);

```

In this case, only the result of the request is returned to the callback function (this corresponds to the 'onSuccess' function of a standard call). Since there is no function dedicated to processing exceptions, if an error occurs, the server returns nothing.

You can implement RPC calls quickly using this type of call but you cannot process errors. It is particularly suited for changing from synchronous mode to asynchronous mode quickly by simply adding the **Async** suffix to the RPC function and inserting the callback function.

Example in asynchronous mode (standard call)

This basic example describes calling a server-side "sum" method using a standard RPC in asynchronous mode. This method adds two values together, which have been sent as parameters.

```

// Server-side code
// Put the 'sum' function in a javascript file named 'math.js'
// located in the 'Modules' folder of the project
// The module file must have been declared in the '.waPerm' file of the project
exports.sum = function (val1, val2) {
    return parseInt(val1, 10) + parseInt(val2, 10); // return base 10 numbers
}

// Client-side code
// The namespace was declared in the GUI Designer
// with the module file name: 'math'
math.sumAsync({
    'onSuccess': function(result) {
        console.log(result),
    }
    'onError': function(error){
        console.log('An error occurred: '+error)
    },
    'params': [4, 5]
});

```

Example in asynchronous mode (simplified call)

The following example describes calling the "writeLog" method using an RPC in asynchronous mode. This method saves a report as a file on the server. Fields having the 'time', 'user' and 'size' IDs that are specified on the HTML page are sent as parameters. You can call this method, for example, when a button is clicked.

```
// Setting up callback function
    callback = function (result) {
        console.log(result);
    }

    send.click = function (event) {
// Building array of values
        var data = $('#time').val(), $('#user').val(), $('#size').val();
// RPC call
        moduleName.writeLogAsync( callback , data);
    };
```

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 server response. 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 = moduleName.functionName(params);
```

You use the *params* argument to send RPC function parameters. This argument is described in the [params parameter](#) section.

Example of a call in synchronous mode

We recommend inserting a synchronous RPC call into a "try catch" structure so that you can process any errors that may occur:

```
//Example of a call in synchronous mode with error processing:
try {
    var result = moduleName.testFunction("echo", "Test");
    console.log("Result: " + result);
} catch (e) {
    console.log("Error: " + e.message);
}
```

Note: The `console.log()` statement generates an entry in the JavaScript console of the browser.

For more information about JavaScript exceptions, refer to the [Error Handling](#) section.

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. 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).

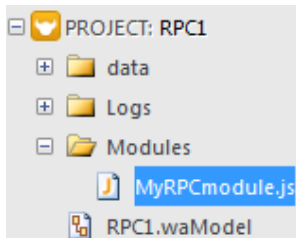
RPC Example

This section proposes a detailed example of an RPC implementation. In this example, we will:

- create a basic RPC module containing an "isEmpty" server-side function that returns **true** if it is called with an empty string as parameter
- add a text area and a button on a HTML page that sends a RPC call to this function

Creating the RPC Module

We create a "MyRPCmodule.js" file and put it in the **Modules** folder of the project:



The "MyRPCmodule.js" file only contains a very simple function:

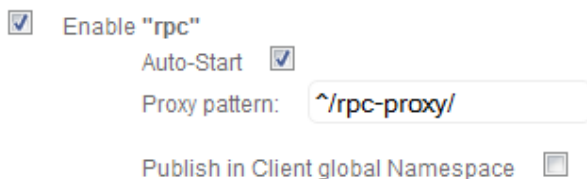
```
//MyRPCmodule.js function
exports.isEmpty = function(param){
    return (param=="")?true:false;
};
```

Configuring the .waPerm file

We add the following entry in the **.waPerm** file of the project to publish the RPC module:

```
<?xml version="1.0" encoding="UTF-8"?>
<permissions>
  <allow type="module" resource="MyRPCmodule" action="executeFromClient" />
</permissions>
```

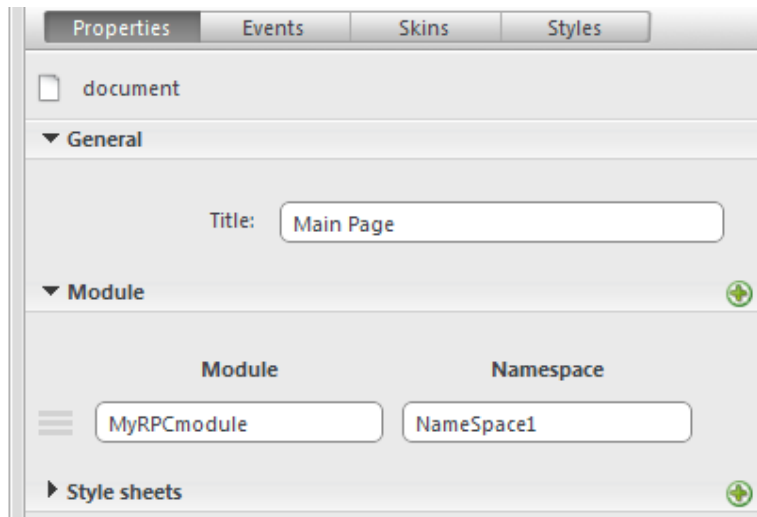
*Note: Make sure that the RPC services are enabled in the **.waSettings** file of the project (enabled by default):*



Configuring the client page

On the client side, we add a main page that contains a text entry area and a button. In the GUI Designer, we add a reference to our RPC module and define the 'Namespace1' namespace:

Value:



Here is the script of the button:

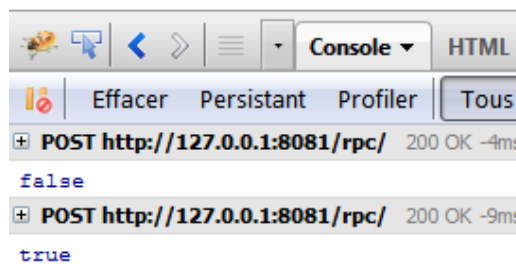
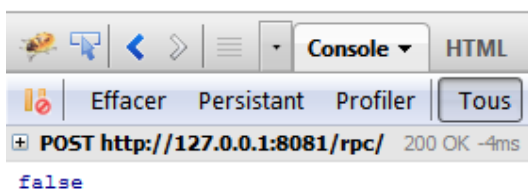
```
button1.click = function button1_click (event)
{
    sources.myValue.sync(); // synchronize the myValue source and variable
    NameSpace1.isEmptyAsync({ // Asynchronous call
        'onSuccess': function (result) {
            console.log(result);
        },
        'onError': function (error) {
            console.log("Error");
        },
    },
    'params': [myValue]
    });
};
```

Testing the RPC

Run the project, enter a value in the "Value" area and click the **Send a RPC** button. You can see in the Console that the result is **false**. Then leave the "Value" area empty and click the **Send a RPC** button: the result is **true**.

Value:

Value:



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 *myFunction* function with error processing based on the error name (which makes the code easier to read):

```
try{
  moduleName.myFunction(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 *myFunction* function but this time with error processing based on the error code:

```
try{
  moduleName.myFunction(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.

Upgrading JSON-RPC from Wakanda v1 to v2 and higher

Starting with Wakanda v2, the RPC implementation is based on CommonJS architecture, which is not compatible with the previous architecture. In particular, declaring RPC function files as "roles" and namespacing functions through the GUI designer **do not work anymore**.

This section provides you with basic steps to convert a Wakanda v1 project that uses RPC to a Wakanda v2-compliant project. For more detailed information on the JSON-RPC implementation in Wakanda v2 and higher, please refer to the sections of the [Using JSON-RPC Services](#) chapter.

1. Convert all your files with the "RPC Function file" role to CommonJS module files.

In this step, you have to:

- Copy all the files in the **modules** folder of your project.
- Edit the code so that it is CommonJS compliant.

For example, if you have an existing "test.js" file with a RPC role containing the following function:

```
function add (a, b) {
    return (a + b);
}
```

You have to update it as follows:

```
exports.add = function add (a, b) {
    return (a + b);
};
```

2. (Optional) If you defined custom namespaces in the GUI designer, transfer these namespaces to the RPC module file.

For more information, refer to the [Namespace declarations](#) section.

3. Declare each module file as "executable from client" in the .waPerm file of the project, within the <permissions> element.

For example, for the "test.js" file, you could add:

```
<allow type="module" resource="test" action="executeFromClient"/>
```

For more information, refer to the [Publishing RPC functions](#) section.

4. Restart the server.

This step is mandatory each time you edit an RPC module file or the .waPerm file.

5. In the Wakanda GUI Designer, open each interface file that uses RPC calls and delete old-style meta tags.

You need to display the source code. Old-style meta tags look like this:

```
<meta name="WAF.config.rpc.namespace" content="rpc" />
```

6. Back in the Design view, add appropriate module file name(s) in the "Modules" area (document property).

Write the module file name in the "Module" entry area. Unless you have defined custom namespaces (see step 2), use the file name as namespace as well in the "Namespace" entry area.

For more information, refer to [Declaration and configuration](#).

7. Save the updated interface file(s).

Now client-side RPC calls such as test.add() or test.addAsync() will work in Wakanda v2 and higher versions.