neuray® Reference: JavaScript Plugin

Document version 1.2
December 20, 2010

Table of Contents

1 Overview 1
2 Tutorials 1
2.1 Echo 1
2.1.1 What You Will Learn 1
2.1.2 Introduction 2
2.1.3 Step 1: Creation of Meta-Data 2
2.1.4 Step 2: Creation of JavaScript 3
2.1.5 Step 3: Testing of JavaScript 3
2.2 Factorial 4
2.2.1 What You Will Learn 4
2.2.2 Introduction 4
2.2.3 Step 1: Creation of Meta-Data 4
2.2.4 Step 2: Creation of JavaScript Using JavaScript's Recursion 4
2.2.5 Step 3: Testing of JavaScript 5
2.2.6 Step 4: Creation of JavaScript Using neuray Services Recursion 5
2.3 Batching Service Commands 7
2.3.1 What You Will Learn 7
2.3.2 Introduction 8
2.3.3 Step 1: Creation of Meta-Data 8
2.3.4 Step 2: Creation of JavaScript 8
2.3.5 Step 3: Testing of JavaScript 9
3 Configuration 9
4 API 10
4.1 Meta-Data 10
4.2 JavaScript API 11
5 Runtime 12
6 Release Notes 12

1 Overview

The JavaScript plugin provides a means to script neuray Services [neuray Services API] through JavaScript. This allows one to script neuray Services without the use of C++.

In particular the JavaScript plugin provides:

- The ability to register a service command that, when executed, runs associated JavaScript contained in a file on disk
- The ability to call any other service commands from within a JavaScript service command independent of the language in which the service command is written

Currently the plugin supports JavaScript version 1.8.0 as implemented in Spidermonkey version "JavaScript-C 1.8.0 pre-release 1 2009-02-16".

This document contains the following:

- **JavaScript plugin tutorials** - Describes how to use the JavaScript plugin to script neuray Services
- **Configuration documentation** - Describes how to configure the JavaScript plugin
- **API documentation** - Describes the JavaScript API presented by the JavaScript plugin
- **Runtime documentation** - Describes how a registered JavaScript service command changes as you modify or remove its associated JavaScript file.
- **Release notes** - Describes this release of the JavaScript plugin

2 Tutorials

2.1 Echo

2.1.1 What You Will Learn

In this tutorial you will learn how to:

- Create "meta-data", data used to specify the interface presented by a JavaScript plugin
- Create JavaScript that executes as a service command
- Test a JavaScript based service command

2.1.2 Introduction

For your first JavaScript plugin you will create a simple JavaScript `echo` service command, similar to the UNIX command line tool `echo`. When presented with a string, your echo service command will simply return the presented string.

To do this you must know that the JavaScript plugin provides the ability to register a service command that, when executed, runs JavaScript contained in a file on disk. The C++ portion of the registered service command is provided by the JavaScript plugin; so, you never need to worry about C++. You, the user, need only provide the JavaScript file.

The JavaScript plugin requires the specification of a directory in which all such JavaScript files reside, and any file with the extension `.js` in that directory will then be used to implement a service command. So, all of your JavaScript files must reside in this directory. By default this directory is `javascript_services/` relative to the server's installation directory. So, all of your JavaScript must reside within this directory.

2.1.3 Step 1: Creation of Meta-Data

Now, to start, create and open a file named `echo.js` in the `javascript_services/` directory. The JavaScript contained in this file will be executed in response to a request for your echo service command and thus must echo its input as output.

Before you add any JavaScript to the file `echo.js`, you must add meta-data, required by the JavaScript plugin, to the file `echo.js`. This meta-data is required so that the JavaScript plugin can specify your service command's interface to neuray Services and to other programmers.

This meta-data takes the form of a formatted comment at the start of the file `echo.js`. This comment must have the following format:

```
//# name = <name>
//# namespace = <namespace>
//# description = <description>
//# argument_descriptions = <argument_descriptions>
//# return_type = <return_type>
```

where `<name>` is the name of the service command, `<namespace>` its namespace, `<description>` its human readable description, `<argument_descriptions>` describes the names and types of its input arguments, and `<return_type>` its return type. Also, you can have more than one line with an `argument_descriptions` comment and the argument descriptions accumulate.

For example, your `echo` command, which has no namespace, takes a `String` as input, and returns a `String` as output, has the following meta-data

```
//# name = echo
//# description = "Echo command that echos a string input"
//# argument_descriptions =  [ { "name": "input", "type" : "String" } ]
//# return_type = String
```

Note: The value of `argument_descriptions` indicates that the `echo` command has a single argument named "input" that is of type `String`.

2.1.4 Step 2: Creation of JavaScript

Now, you can start writing the JavaScript that implements the echo command. Add the following line to the end of the file `echo.js`:

```
input;
```

This simple line "returns" the value held in the variable `input` to the calling code. As the value held in the variable `input` is the input `String`, this line echos the input as output.

In general, the last expression evaluated in the JavaScript file is the return value of the file. The return value is *not* returned, as you might expect, by a normal `return` statement at the file's end.

In summary, the entire contents of your file `echo.js` should be:

```
//# name = echo
//# description = "Echo command that echos a string input"
//# argument_descriptions = [ { "name": "input", "type" : "String" } ]
//# return_type = String
input;
```

2.1.5 Step 3: Testing of JavaScript

To test your first JavaScript service command, you need to create a client that can communicate with neuray Services through the appropriate protocol. The simplest way is to use the JavaScript client library that ships with neuray Services. However, you can use any apropos client.

As the details of creating such a client are contained in the JavaScript client library documentation [JavaScript Client API], we assume here that you have read said documentation, or documentation describing some other means of creating a client, and can create such a client on your own. Thus, we do not cover creating such a client here.

You are encouraged to play with this JavaScript service command. Try changing the name of the command by editing the `name` comment in the `echo.js` file, setting it to `ohce`. Also change the JavaScript in `echo.js` to the following:

```
//# name = ohce
//# description = "Ohce command that sohce a string input"
//# argument_descriptions = [ { "name": "input", "type" : "String" } ]
//# return_type = String
var output = "";
for(var index = input.length;index > 0;--index)
    output = output + input.charAt(index - 1);
output;
```

Use your client to access this new service. What happens?

2.2 Factorial

2.2.1 What You Will Learn

In this tutorial you will learn how to:

- Call a service command from within a JavaScript based service command

2.2.2 Introduction

For your next JavaScript plugin you will create a simple JavaScript `factorial` service that computes the factorial of a number passed to it[1]. When presented with a positive integer, your `factorial` service will return the factorial of the passed positive integer. In creating this `factorial` service you will learn how to call other services from within JavaScript.

2.2.3 Step 1: Creation of Meta-Data

To start, create and open a file named `factorial.js` in the `javascript_services/` directory. As before, the JavaScript contained in this file will be executed in response to a request for your factorial service and thus must compute the factorial of its input as its output. Also, as before, the first thing you must specify in this JavaScript file is the meta-data for the factorial service.

Your factorial service will be named `factorial`. It will take as input an integer named `number` that is of type `Uint64`, and it will return an integer of type `Uint64`. Hence, `factorial.js` begins with the meta-data:

```
//# name = factorial
//# description = "Factorial command that returns the factorial of its input as output."
//# argument_descriptions = [ { "name": "number", "type" : "Uint64" } ]
//# return_type = Uint64
```

2.2.4 Step 2: Creation of JavaScript Using JavaScript's Recursion

Next, you will write the JavaScript that implements the factorial service.

One way of implementing the factorial service is to use JavaScript's ability to recursively call itself. Doing so you would add the following JavaScript after the meta-data:

```
function factorial(val)
{
    if(1 == val)
      return 1;
    return val * factorial(val - 1);
}

factorial(number);
```

[1] For those that need a refresher, the factorial of a positive integer n, written as $n!$, is by definition $n*(n-1)*(n-2)*\cdots 2*1$. So, the factorial of 1 is 1; the factorial of 2 is 2; and the factorial of 3 is 6.

Thus, the entire `factorial.js` file now should have the form:

```
//# name = factorial
//# description = "Factorial command that returns the factorial of its input as output."
//# argument_descriptions = [ { "name": "number", "type" : "Uint64" } ]
//# return_type = Uint64
function factorial(val)
{
    if(1 == val)
      return 1;
    return val * factorial(val - 1);
}

factorial(number);
```

2.2.5 Step 3: Testing of JavaScript

Now, to test your JavaScript service command, you need to create a client that can communicate with neuray Services through the appropriate protocol. As before, we will assume that you can create such a client on your own.

2.2.6 Step 4: Creation of JavaScript Using neuray Services Recursion

Now your next step will be to implement this factorial service in a different manner, not using JavaScript's ability to recursively call itself, but using the ability of the JavaScript plugin to call other service commands from within JavaScript. In particular, you will have the factorial service call itself. But, before you do this, you need to examine how to call other service commands from within JavaScript.

The JavaScript runtime contains a global instance named `NRS_Commands` that has a set of member functions each of which corresponds to a service command. So, for example, as there is a command named "import_scene", the JavaScript global instance `NRS_Commands` has a member function `NRS_Commands.import_scene()`. Similarly, as a result of your factorial command, the global instance `NRS_Commands` has a member function `NRS_Commands.factorial()`.

The signature of these `NRS_Commands` member functions is derived from the signature of the corresponding service command. If you examine the documentation for a given service command, you find that it takes a set of named arguments with specified types. These arguments determine the signature of the corresponding `NRS_Commands` method.

For example, your factorial command has a single argument named `number` of type `Uint64`. Hence, the corresponding method `NRS_Commands.factorial()` takes as an argument a single JavaScript `Object` with a member named "number". Specifically, this JavaScript `Object` has a single member named "number" that is an integer. So, to call the factorial command with an argument `10` from within JavaScript, you would write the following code:

```
var arg = new Object();
arg.number = 10;
NRS_Commands.factorial( arg );
```

Equivalently, using JavaScript object literal syntax [Flanagan 02], you can call the factorial command from within JavaScript with an argument `10` as follows:

```
NRS_Commands.factorial( {number : 10} );
```

Note: The arguments passed to the member functions of `NRS_Commands` are passed by value in contrast to the normal way JavaScript `Object`'s are passed to JavaScript functions.

More generally, if there is a command named `generic_command` that takes arguments named `argument1`, `argument2`..., then calling such a command from within JavaScript is done as follows:

```
var arg = new Object();
arg.argument1 = ...
arg.argument2 = ...
...
NRS_Commands.generic_command( arg );
```

where the assignments

```
arg.argument1 = ...
arg.argument2 = ...
...
```

assign `argument1`, `argument2`... to values with types specified by the documentation of the service command `generic_command`.

The question then arises: How are instances of such types created in JavaScript? In most cases this is straightforward. You create a JavaScript `Object` and add members named after the specified type's members then assign each member a value of the desired type. For example, a `Color` can be created in JavaScript by creating a JavaScript `Object` and adding number valued members `r`, `g`, `b`, and `a`:

```
var color = new Object();
color.r = .01;
color.g = .20;
color.b = .40;
color.a = 1.0;
```

Other types can be dealt with in a similar manner. The only exception to this rule is the type `Canvas` which can not be directly created in JavaScript.

A similar rule applies to the values returned from a given command. For example, if there is a command named `generic_command` documented as returning a `Color`, then calling the method `NRS_Commands.generic_command()` would return a JavaScript `Object` with four number valued members `r`, `g`, `b`, and `a`. Similarly, if it were documented as returning a `Float64<2>`, then calling the method would return a JavaScript `Object` with two number valued members `x` and `y`. The only exception to this rule is `Canvas`. A method documented as returning a `Canvas` returns an opaque JavaScript `Object`. This opaque `Object` can, however, be passed to other methods expecting a `Canvas`.

An error condition raised by a service command called in this manner is converted in to JavaScript exception. In particular, an `Error` is thrown containing the error message produced by the corresponding service command. In addition, any uncaught JavaScript exception in a script will propagate up causing the service command the script defines to raise an error condition.

Your original implementation of `factorial.js` has the following form:

```
//# name = factorial
//# description = "Factorial command that returns the factorial of its input as output."
//# argument_descriptions = [ { "name": "number", "type" : "Uint64" } ]
//# return_type = Uint64
function factorial(val)
{
    if(1 == val)
      return 1;
    return val * factorial(val - 1);
}

factorial(number);
```

So, using the above comments on how to call a service command from within JavaScript, you can use the global `NRS_Commands` instance to implement the factorial function as follows:

```
//# name = factorial
//# description = "Factorial command that returns the factorial of its input as output."
//# argument_descriptions = [ { "name": "number", "type" : "Uint64" } ]
//# return_type = Uint64

var result;
if(1 == number)
  result = 1;
else
  result = number * NRS_Commands.factorial( {number : (number - 1)} );

result;
```

With a little thought you can see that this is functionally the same code, but it makes use of the fact that in JavaScript you can call other service commands.

To test this new implementation you can use the same client you created previously. You will obtain the same results as before.

2.3 Batching Service Commands

2.3.1 What You Will Learn

In this tutorial you will learn how to:

- Batch calls of service commands from with a JavaScript based service command

2.3.2 Introduction

For your next JavaScript plugin you will create a simple JavaScript service `javascript_batch` that executes several services together.

2.3.3 Step 1: Creation of Meta-Data

The JavaScript service `javascript_batch` imports a scene in blocking mode, sets the camera's aspect ratio, and sets the camera's resolution while returning no value. It takes as input the name of the scene (a String), the name of the `.mi` file (a String), the camera name (a String), the image width (a Uint64), and the image height (a Uint64).

Hence, if you store this command's implementation in a file `javascript_batch.js` and place it in the normal location, then the meta-data in `javascript_batch.js` will be of the form:

```
//# name=javascript_batch
//# description=An example using Javascript to batch commands
//# argument_descriptions=[{"name":"sceneName","type":"String"},{"name":"fileName","type":"String"}]
//# argument_descriptions=[{"name":"cameraName","type":"String"}]
//# argument_descriptions=[{"name":"imageWidth","type":"Uint64"}]
//# argument_descriptions=[{"name":"imageHeight","type":"Uint64"}]
//# return_type=Void
```

Note, when there are many argument descriptions comment lines, as above, the argument descriptions accumulate.

2.3.4 Step 2: Creation of JavaScript

The JavaScript in `javascript_batch.js` calls existing services to import the scene, set the camera aspect ratio, and set the camera's resolution. So, the JavaScript only needs to make calls to the services `NRS_Commands.import_scene()`, `NRS_Commands.set_camera_aspect()`, and `NRS_Commands.set_camera_resolution()`. When you look at the documentation for these services, you find they can be called as follows:

```
NRS_Commands.import_scene({block:"true", scene_name:sceneName, filename:fileName});
NRS_Commands.set_camera_aspect({camera_name:cameraName,
                                aspect:(imageWidth/imageHeight)});
NRS_Commands.set_camera_resolution({camera_name:cameraName,
                                    resolution:{ x:imageWidth, y:imageHeight}});
```

So, in summary, the file `javascript_batch.js` has the following content:

```
//# name=javascript_batch
//# description=An example using Javascript to batch commands
//# argument_descriptions=[{"name":"sceneName","type":"String"},{"name":"fileName","type":"String"}]
//# argument_descriptions=[{"name":"cameraName","type":"String"}]
//# argument_descriptions=[{"name":"imageWidth","type":"Uint64"}]
//# argument_descriptions=[{"name":"imageHeight","type":"Uint64"}]
//# return_type=Void
```



```
NRS_Commands.import_scene({block:"true", scene_name:sceneName, filename:fileName});
NRS_Commands.set_camera_aspect({camera_name:cameraName,
                                aspect:(imageWidth/imageHeight)});
NRS_Commands.set_camera_resolution({camera_name:cameraName,
                                    resolution:{ x:imageWidth, y:imageHeight}});
```

2.3.5 Step 3: Testing of JavaScript

Now you only need a client to call this service. As usual, we assume that you can write such a client on your own.

3 Configuration

Generically, when you start neuray Services you specify a configuration file that is used to configure neuray Services. This configuration file is specified through the value of the command line flag `--config_file`. The default installation of neuray Services sets the value of this flag to be the file `realityserver.conf` in the root directory of the neuray Services.

The configuration file, be it the default `realityserver.conf` or any another specified configuration file, is used to configure the JavaScript plugin. The configuration of the JavaScript plugin consists, in total, of setting the value of a single key, `directory`, in the configuration file.

The key `directory` in the configuration file should be set to have a value which is the relative path of the directory containing the JavaScript files that implement the service commands. This path should be relative to the neuray Services root directory.

For example, if the JavaScript files that implement the service commands are in the directory `javascript`, relative to the neuray Services root directory, then the key/value pair in the configuration file will have the form:

```
directory javascript
```

However, to indicate that this key/value pair is to be used by the JavaScript plugin, and not some other plugin, you have to enclose this key/value pair with a `<user>...</user>` tag as follows:

```
<user javascript>
directory javascript
</user>
```

Note, the specified directory must exist; the JavaScript plugin will not create it for you.

In addition, you can specify that neuray Services look in more than one directory for the JavaScript files that implement the service commands. This is done by repeating the key “directory” in the configuration file with a different value each time.

For example, if the JavaScript files that implement the service commands are in the directories `javascript1`, `javascript2`,... ,`javascriptN`, all relative to the neuray Services root directory, then the key/value pairs in the configuration file will have the form:

```
<user javascript>
directory javascript1
directory javascript2
...
directory javascriptN
</user>
```

Again, the specified directories must exist.

4 API

4.1 Meta-Data

The meta-data for a JavaScript service command is defined by a set of comments that begin the JavaScript file in which a JavaScript service command is defined. Generically, such comments are of the form:

```
//# name = <name>
//# namespace = <namespace>
//# description = <description>
//# argument_descriptions = <argument_descriptions>
//# return_type = <return_type>
```

where you can have more than one line with an `argument_descriptions` comment and the argument descriptions accumulate. Also, note that the `namespace`, `description`, and `argument_descriptions` comments are optional. While `name` and `return_type` are required. If absent, `namespace` defaults to an empty string, `description` defaults to an empty string, and `argument_descriptions` defaults to no arguments. Also, note that if there are two service commands with the same name and namespace, the last service command registered is the active command and the other is discarded.

The EBNF grammar, where {...} indicates the enclosed expression is repeated *0* or more times, for the various pieces above is as follows:

```
<name> : <identifier>
<namespace> : <identifier>
<description> : <string_literal>
<argument_descriptions> : <json_array_of_argument_descriptions>
<return_type> : <typename>
```

where `<identifier>` is an alphabetic character followed by a possibly empty sequence of alphabetic characters, decimal digits, and underscores that is neither a typename nor a JavaScript reserved word; `<string_literal>` is a possibly empty sequence of characters not including a double quote or escape sequences, enclosed in double quotes.

`<json_array_of_argument_descriptions>` is a JSON formatted array of objects which describe the command arguments. Each object supports the following keys:


```
name : JSON string containing the argument name.
type : JSON string containing the argument typename.
description : JSON string containing the argument description.
default : the arguments default value, the type of this depends on the arguments type
          and may also be null.
```

`name` and `type` are required but all others are optional. `<type>` names a type in the neuray Services. (See the *Types* documentation [neuray API] to find the supported type names.)

4.2 JavaScript API

The JavaScript API presented by the JavaScript plugin is lean and focused. First, it presents the pure JavaScript API presented by version 1.8.0 of JavaScript as implemented in Spidermonkey version "JavaScript-C 1.8.0 pre-release 1 2009-02-16". Second, it presents a global instance named `NRS_Commands` that has a member function for every installed service command.

Generically, if there is a service command named `generic_command` that takes arguments named `argument1`, `argument2`... of types `typename1`, `typename2`... respectively, then `NRS_Commands` will have a member function named `generic_command` that will take a single argument, a JavaScript `Object`. This JavaScript `Object` must have members named `argument1`, `argument2`... that are of types `typename1`, `typename2`... respectively. Calling such a command from within JavaScript is done as follows:

```
var arg = new Object();
arg.argument1 = ...
arg.argument2 = ...
...
NRS_Commands.generic_command( arg );
```

where the assignments

```
arg.argument1 = ...
arg.argument2 = ...
...
```

assign `argument1` to a value of type `typename1`, `argument2` to a value of type `typename2`...

To create a JavaScript value of a specified type create a JavaScript `Object` and add members named after the specified type's members then assign each member a value of the desired type. The only exception to this rule is the type `Canvas` which can not be directly created in JavaScript.

A similar rule applies to return values. For example, a command documented as returning a `Color`, returns a JavaScript `Object` with four number valued members `r`, `g`, `b`, and `a`. The only exception to this rule is `Canvas`. A method documented as returning a `Canvas` returns an opaque JavaScript `Object` that can be passed to other methods expecting a `Canvas`.

An error condition raised by a service command called in this manner is converted to a JavaScript exception. In particular, an `Error` is thrown containing the error message produced by the corresponding service command. In addition, any uncaught JavaScript exception in a script will

propagate up causing the service command the script defines to raise an error condition. The specific error message will be returned to the user as the `data` parameter of the error.

5 Runtime

The neuray Services server is robust and flexible with respect to changes in the JavaScript and meta-data defining a service command.

You can, at any time, change the JavaScript code defining a service command. Any subsequent execution of the service command will use the new JavaScript code without requiring a server restart.

In addition, you can, at any time, change any and all meta-data in a file defining a service command. Upon doing so, the old service command, corresponding to the old meta-data, will be un-registered and the new service command, corresponding to the new meta-data, will be registered. Any subsequent call to the new service command will work as expected while any call to the old service command will fail. Again, this requires no server restart.

6 Release Notes

This is version 1.0 of the JavaScript plugin and there are no known issues.

Bibliography

[Flanagan 02] D. Flanagan, *JavaScript, The Definitive Guide*, 4th edn. O'Reilly, 2002.

[neuray API] mental images GmbH, Berlin, Germany, *neuray API.*

[neuray Services API] mental images GmbH, Berlin, Germany, *neuray Services API.*

[JavaScript Client API] mental images GmbH, Berlin, Germany, *JavaScript Client API.*