

# helios javascript framework tutorial

February 9, 2009

Copyright ©2008, 2009 Dmitry Prokashev.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. You should received a copy of GNU Free Documentation License along with this document. If not, see <http://www.gnu.org/licenses/>.

## Contents

1	Intro	4
2	Getting closer	4
3	Modules	4
4	Path to the module	6
5	Dynamic module loading	6
6	Unloading a module	7
7	Loading feedback	10
8	Debugging	10
9	Further reading	10

## 1 Intro

This tutorial aims to provide basic understanding of how helios framework works and how to create modular application on JavaScript using helios.

Helios is a framework for creating powerful browser-based JavaScript applications. That means that there is an **index.html** which is fed to the browser and provides to parse helios kernel, and of course that also means that any helios application may be loaded to the browser from the remote server, which makes the framework usefull for creating powerful client-side applications which will not require any installation.

When helios kernel is loaded, it starts parsing **main.js** file where the application code is located. Thus a programmer starts creating his application in pure JavaScript with no need to ache his head with inventing a set of HTML templates.

Helios introduces pure modularity into browser-based JavaScript, i.e. a programmer just need to call *include("foobar.js")* in the head of his module, and framework will take care of parsing and initializing the desired module and all other modules required by this module in order of dependence.

Helios also supports dynamic module loading and unloading. This may be usefull for creating applications which will extend their functionality only when it is required and free the memory back unloading the modules which will not be used anymore.

## 2 Getting closer

The programmer starts his code in **main.js**. Simple “hello-world” application will look like this:

```
// main.js

function init() {
    window.alert("hello world");
}
```

Let us explore the code written above. First of all, if it is not yet clear, to launch this application you need to put this code into **main.js** which should be located alongside with **index.html** provided by the helios distribution. Second, you need to open **index.html** in the browser. Obviously, this code above will show an alert with the text.

As mentioned above, helios manages modules, and **main.js** is a module too. Each module should have its initializer, a function which will be called as soon as module is loaded and all its dependence modules are initialized. The initializer function should be declared as *function init()*. In this example, **main.js** module does not have external dependences, so helios will load **main.js** and launch its *init()* function which will show an alert.

## 3 Modules

Now let us create a simple library consisting of two modules. The purpose of our example library will be to calculate a number of  $k$ -combinations (each

consisting of  $k$  elements) from a set:

$${}_n C_k = \frac{n!}{k!(n-k)!}$$

where  $n$  is a total number of elements in the set. For more information about combinations, read “Combination” article on Wikipedia.

First we have to create a subsidiary module which will provide a function for calculating the factorial of a number. It may be preferable to create a simple function instead, but in this example it will be separated to demonstrate how modules are included in helios.

**factorial.js** module will contain this code:

```
// factorial.js

function init() {

    factorial = function( number ) {
        if ( number > 2 ) {
            return ( number * factorial( number - 1 ) );
        } else {
            return number;
        }
    }
}
```

This module contains a definition of function called *factorial* and implements a classic recursive algorithm for computing the factorial of a number. For simplicity, this implementation does not contain necessary checks on whether a *number* is positive integer as required.

Second module, called **combination.js**, will provide a function for calculating  $k$ -combination:

```
// combination.js

include( "factorial.js" );

function init() {

    combination = function( n, k ) {
        return factorial( n ) / ( factorial( k ) * factorial( n - k ) );
    }
}
```

Here we can see that **combination.js** module *include()*'s **factorial.js**, and newly declared function *combination* uses function *factorial* from the **factorial.js** module. Helios kernel will take care of initializing **factorial.js** before **combination.js**. Now let's create a **main.js** which will use **combination.js** module:

```

// main.js

include( "combintation.js" );

function init() {

    window.alert( "combination( 10, 5 ) = " + combination( 10, 5 ) );

}

```

And after launching the program we will see a message saying “combination( 10, 5 ) = 252”. Notice that **main.js** does not include **factorial.js**, but this module is included by the kernel and is initialized before **combination.js** which uses it.

Since **factorial.js** is initialized before **main.js**, you can also use its *factorial* function inside **main.js** initializer and it will work fine. But in this case you should better include required **factorial.js** patently in **main.js** before module initializer:

```

// main.js

include( "combintation.js" );
include( "factorial.js" );

function init() {

    window.alert( "combination( 10, 5 ) = " + combination( 10, 5 ) );
    window.alert( "factorial( 5 ) = " + factorial( 5 ) );

}

```

The **factorial.js** module will not be parsed or initialized twice, but if some-time you will decide to remove a call to *combination* from your module, you will not be discouraged on why your *factorial* function is not working anymore.

## 4 Path to the module

Argument of *include()* function is a path to the module. You can provide either an absolute path (starting with “http://...”), or a relative (relatively to the path of the module from which your performing including). Thus, if you have two modules, path of the first one is **include/library1/module1.js** and it should use the second module which is available as **include/library2/module2.js**, then in the head of **module1.js** you should type:

```
include( "../library2/module2.js" );
```

## 5 Dynamic module loading

To include module dynamically (from inside the function), you have to use *load()* kernel funciton. When you call *load()* or *include()*, helios kernel includes

the desired module and all modules which the module requires. Function *load()* works in asynchronous way, meaning that the next statement of your code which follows the *load()* call will be processed before the module is included. For this reason *load()* function takes second optional argument, an action which should be performed after the desired module is included. This argument can be either a string containing some JavaScript expression (which will be evalled after the module is included), or a function object (which will be simply called without arguments). You can either provide a name of some existing function, or create an anynamous function right inside an argument list. Here are some examples of how to load modules in dynamic way:

```
// this will show an alert when the module is loaded
load( "myModule.js", "window.alert( 'myModule.js loaded!'" );

// this will call function named doSomething() after module is loaded
load( "myModule.js", doSomething );

// this will call a function which is provided as a second argument
load( "myModule.js", function(){
    window.alert( 'myModule.js loaded!' );
    doSomething();
} );
```

## 6 Unloading a module

Module excluding implies destroying all objects declared in the module initializer, removing the module object from the kernel, cleaning dependences of the module and finally excluding all modules included by that module. Excluding is safe, meaning that a module will not be excluded in case when there are some modules which still require it.

Helios kernel does not track the objects created by the module's initializer. Instead the module author should provide a function which will perform such cleanup (if author aims to create a module which will be probably loaded and unloaded dynamically). This function is called module uninitializer. But even if there is no uninitializer, the module object will be removed and all modules requested by that one will be excluded too.

The syntax of a module given above (whith *init()* function as a module initializer) is actually a short notation and should be used for a modules which do not have uninitializer. If a module should be unloadable, its initializer and uninitializer should be called as *initialize()* and *uninitialize()*, like this:

```
function initialize() {
    // this function is called when a module is included
    // objects are created here
}

function uninitialize() {
    // this function is called when a module is excluded
}
```

```
    // objects created in initialize() should be destroyed here
}
```

To provide an example, let us remake the library for calculating  $k$ -combinations in the unloadable manner. Three source files (**factorial.js**, **combination.js** and **main.js**) will now look like this:

```
// factorial.js
```

```
function initialize() {

    // creating the factorial function
    factorial = function( number ) {
        if ( number > 2 ) {
            return ( number * factorial( number - 1 ) );
        } else {
            return number;
        }
    }
}
```

```
function uninitialize() {

    // destroying the factorial function
    factorial = null;
    delete factorial;

}
```

```
// combination.js
```

```
include( "factorial.js" );
```

```
function initialize() {

    // creating the combination function
    combination = function( n, k ) {
        return factorial( n ) / ( factorial( k ) * factorial( n - k ) );
    }
}
```

```
function uninitialize() {

    // destroying the combination function
```

```

    combination = null;
    delete combination;
}

// main.js

function init() {

    // this will be called when combination.js dynamically loaded
    showCombination = function() {
        window.alert( "combination( 10, 5 ) = " + combination( 10, 5 ) );

        // unload a module since we don't need it anymore
        unload( "combination.js", unloadFinished );
    }

    // this will be called when combination.js is dynamically unloaded
    unloadFinished = function() {
        window.alert( "combination unloaded" );
        // uncommenting next string will make an error
        // window.alert( "combination( 10, 5 ) = " + combination( 10, 5 ) );
    }

    // dynamically including combination.js
    load( "combination.js", showCombination );
}

```

In the above example, **combination.js** is included dynamically from **main.js** module, *showCombination()* is called (as soon as the requested module is loaded) which makes an alert to appear. After that, a module will be unloaded and *unloadFinished()* function will be finally called, which will show an alert saying that module is unloaded. At this moment, the function *combination()* (and *factorial()* too) are already destroyed.

Notice that you can only *unload()* the modules which were included in a dynamic way using the *load()* function, and you can not patently *unload()* modules which were *include()*'d in some others' head. This is because *include()*'d modules are still required by those who included them. But they will be excluded as soon as they are not required anymore, as it happend with **factorial.js** in the example above.

Important note: if you provide a relative path to *load()* and *unload()* functions, this path must be related to your **main.js** location, not to the path of the module from where you perform this dynamic (un)loading. Otherwise the module will not be found.

## 7 Loading feedback

By default helios kernel shows a simple loading indicator bar which is filled as soon as more modules are initialized and is hidden when **main.js** starts. In above examples you may not noticed that bar, because it was hidden quickly, but in real applications it should be usefull.

In the head of **include/helios/kernel.js** file (which contains helios kernel source) there is a configuration section with a set of commented options which allow you to tune this bar representation or to hide it completely.

## 8 Debugging

Helios kernel also provides several simple features which will hopefully help you to debug your applications. First of all, there is a configuration option *debugMode* which enables these features. In debug mode loading bar is equipped with some additional information (which shows the path of the modules which are intialized/parsed at the moment). Also, error messages will appear if you will try to load or unload modules not in proper way.It will show warnings about modules, which were ignored to be included or excluded (and explain the reason), and warn you about circular dependences occured in your module structure.

After kernel is initialized (this happens when all indicators disappear and **main.js** is started), these messages will also disappear, but you can always get them fetching the *heliosKernel.messages* string variable.

## 9 Further reading

This section is not yet written. It should contain references to different helios libraries for creating widgets, interacting with the server, etc.