Script Extensions

About Script Extensions

A script extension can be used as a library with a collection of objects, variables and functions. BPS itself contains several script extensions, for example `bps` and `bps.gui` which are documented for public use, `bps.audit`, `bps.installer` etc which are used internally by bps itself, and `samples` amd `samples.reports` as simple examples.

Technically extensions can be written as c++ plugin, as plain script or as a combination thereof. The `bps` and `bps.gui` are examples of extensions combined by a c++ plugin (`plugins\script\bpsscript.dll` and `plugins\script\bpsscrgui.dll`) and of scripts (`plugins\script\bps` and `plugins\script\bps\gui`). This article only handles plain script plugins. If you plan to use C++ for your extension please read about it in the Qt documentation.

Custom Script Extensions

When creating your own extensions, take care of the following naming conventions:

- Name your plugin so that it does not conflict with standard bps plugins. For example if your company is *ACME Corp*, use plugins such as `acme`, `acme.interface` etc.
- Plugin names must match the constraints of JavaScript variables. Do not use any blanks or special characters not allowed in a variable name, and start all parts of the plugin path names with a character allowed as first character of a JavaScript variable name.

Create your first plugin by creating a folder under the BPS plugins/script directory, and then create the extension script with the standard name `__init__.js`:

`C:\Program Files\IBK BPS 2.19.0\plugins\script\test1\__init__.js`:

```
__setupPackage__(__extension__);   // initialize extension

var ext = eval(__extension__);     // create shortcut for this extension

ext.version = '0.0.1';             // create a variable

ext.hello = function(aName)        // create a function
{
    print('Hello '+aName);
}
```

Now lets explain this:

__extension__	A predefined variable with the extension name. In this example the variable holds „custom“, in a sub-extension it may be „custom.interface“ for example. Using this variable instead of the name directly is helpful if you ever need to rename, move or clone your extension, because the following script content is then independent of the actual extension name.
__setupPackage__ (__extension__);	A convenience function for setting up a „namespace“ in the script environment. A typical application is to call __setupPackage__() with __extension__ as argument; e.g. __setupPackage__(„custom.interface.import“) would ensure that the object chain represented by the expression custom.interface.import exists in the script environment.
var ext = eval (__extension__);	The variable is a shortcut to the object (or namespace) „custom“ that was created by the previous __setupPackage__ call. Instead of this shortcut you could as well access the namespace explicitely, but then you loose the code independance of the script name of cause.

The rest are examples how to create a variable and a function within the extension.

Next create a script to test the extension:

D:\myscripts\testext1.js:

```
importExtension('test1');          // load my extension
print(test1.version);              // use variable in my extension
test1.hello('Peter');              // use function in my extension
```

Run the script:

```
D:\myscripts>bps testext1
0.0.1
Hello Peter

D:\myscripts>
```

Dependent Extensions

As explained before it is possible to order extensions hierarchically, and the higher up (parent) extensions are assumed to be required for the sub-extensions. Qt Script will automatically load the parent extensions in advance when you load a sub-extension. For example

```
loadLibrary('a.b.c');
```

will let Qt Script first load extension a, then b and finally c. So by this one line you make all elements of the three extensions available to your script.

It is however important to have a minimal __init__.js in all extension directories even if no functions or variables are exposed by the extension. A minimal __init__.js should contain the line

```
__setupPackage__(__extension__);
```

Organizing Your Extensions

In the previous article we told that storing any custom scripts in the BPS installation directories is not recommended, and same is true of cause also for extensions.

Basically what we need to do is create an additional `plugins` directory and tell BPS to also look there for extensions.

Our directories shall be organized as this:

D:\myscripts	Our main scripts are here
D:\myplugins\script\myexts	Our root extension with general purpose functions
D:\myplugins\script\myexts\test2	Our special purpose extensions go all below D:\myscripts\script\myexts

Create the directory `D:\myplugins\script`

Create the directory `D:\myplugins\script\myexts` and create file `__init__.js` in it:

```
__setupPackage__(__extension__);

var ext = eval(__extension__);     // create shortcut for this extension

ext.version = __extension__ + ' 0.0.1';   // create a variable
```

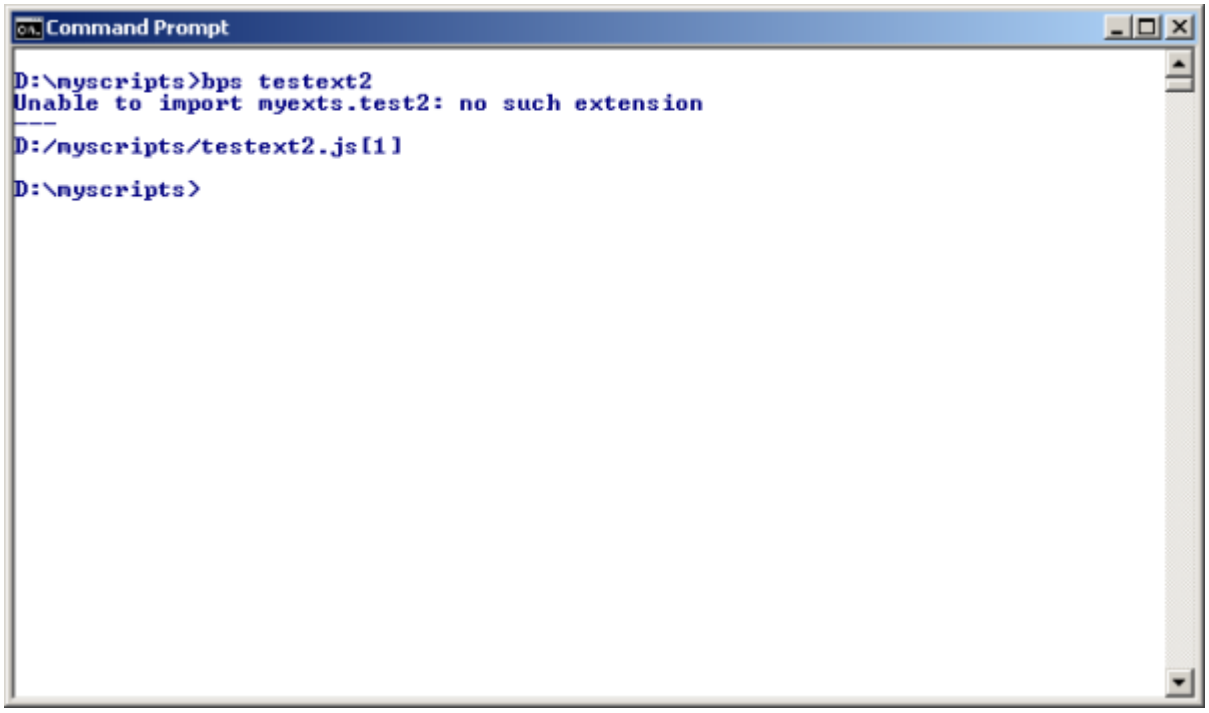
Create directory `D:\myplugins\script\myexts\test2` and create file `__init__.js` in it:

```
__setupPackage__(__extension__);   // initialize extension

var ext = eval(__extension__);     // create shortcut for this extension

ext.version = __extension__ + ' 0.0.2';   // create a variable

ext.hello = function(aName)        // create a function
{
    print('Hello '+aName);
}
```

Create the test application `D:\myscripts\testext2.js`:

```
importExtension('myexts.test2');  // load
print(myexts.version);            // use variable in my extension
print(myexts.test2.version);      // use variable in my sub-extension
myexts.test2.hello('John');       // use function in my sub-extension
```

If you run `testext2` now it is not surprising that you get an error message, because BPS yet knows nothing about the custom plugins directory:



The trick is to declare a custom `Plugins` path in section `[Paths]` of the `bps.conf` file:

```
[Paths]
Plugins=D:/myplugins
```

Up to 2.19 a custom `Plugins` path needed to be set in a manually created `qt.conf` file. However starting with 2.20 `qt.conf` is created automatically during software installation and holds the default plugins path which can not be changed, otherwise the platform pluging introduced with Qt 5 will not get found.

Therefore from 2.20 you *must* set a custom `Plugins` path in `bps.conf`.

Forward slashes are used here in the path. When using backslashes in a `.conf` file path they would have to be doubled, for example

```
[Paths]
Plugins=D:\\myplugins
```

Instead of directory on a local disk you could also use a network share:

```
[Paths]
Plugins=//myserver/myshare/myplugins
```

However be aware that BPS execution will be blocked whenever the network share is unavailable. For best robustness it is recommended to put the custom plugin directory on

a physical disk on the same machine where the BPS software is installed.

So now it should work: