Documentation

PURITY helper

[Developer: Dmitry Astafyev]

[version: 1.00]

1. About Purity

1.1. What is Purity?

Purity looks like framework. Open wiki and find there:

Wiki says: *"In computer programming, a software framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software. A software framework is a universal, reusable software platform to develop applications, products and solutions. Software frameworks include support programs, compilers, code libraries, tool sets, and application programming interfaces (APIs) that bring together all the different components to enable development of a project or solution."*

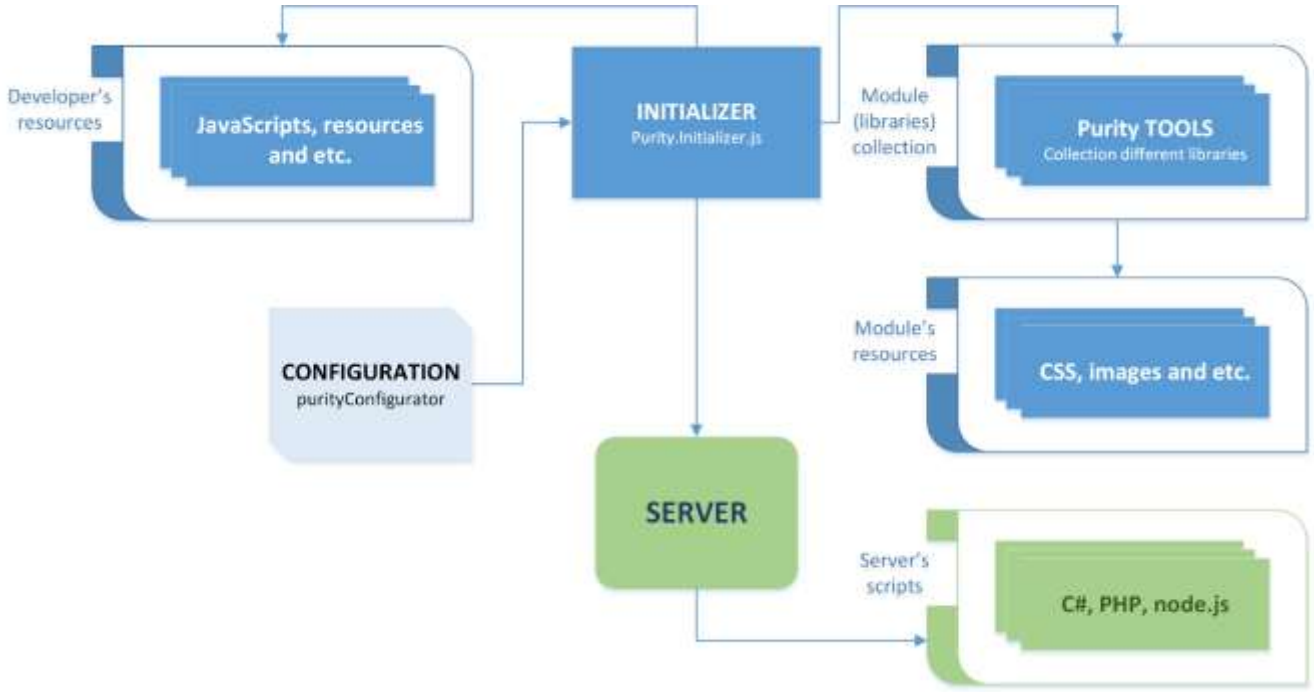
But here is one little problem. If developer use some framework, developer cannot go out of borders of this framework. For example, .NET framework. Developer cannot develop under .NET something not .NET; developer cannot use different from .NET's framework architecture.

And what about Purity? Developer can follow architecture of Purity and builds an application via a philosophy of Purity. But developer can don't follow Purity's architecture and use some functionality of Purity like library. It isn't all. Developer can follow architecture of Purity and doesn't do in same time. Purity cannot make any borders for developer, firstly because JavaScript is a full freedom. And what Purity is, depends only form developer.

If developer likes make structured code, he will use Purity as framework. I developer needs only some functionality, he will use Purity like library. So, Purity is: **a collection of libraries (modules), which can be used like single library (or group of libraries) or like framework.**

1.2. Purity structure

Let's see on Purity structure.



Picture 1.0 – Purity structure

Main part of Purity is [**Initializer**]. Be careful, it isn't kernel. Purity haven't kernel. [**Initializer**] is initializer. It's simply one of Purity's module, which controls loading of other modules and initialization of it. But this

module is basic because without this module Purity will not work. Developer can don't use any other module and he will get working Purity, but without Initializer Purity cannot "live". Also only [**Initializer**] has configuration. It allows developer configure Purity's functionality.

Purity consists of a different libraries (modules). And modules can depend from some resources, like css-content, images or some other content.

After [**Initializer**] build all Purity's modules, [**Initializer**] can manage developer's code: load it, initialize it, storage it and etc.

And last part of Purity is a server's scripts. Server's scripts allow use caching system of Purity to make loading of site faster.

> **NOTE** Nice specific of Purity – it can work without server part. But in this case, developer cannot use Purity's caching system.

1.3. For what is Purity?

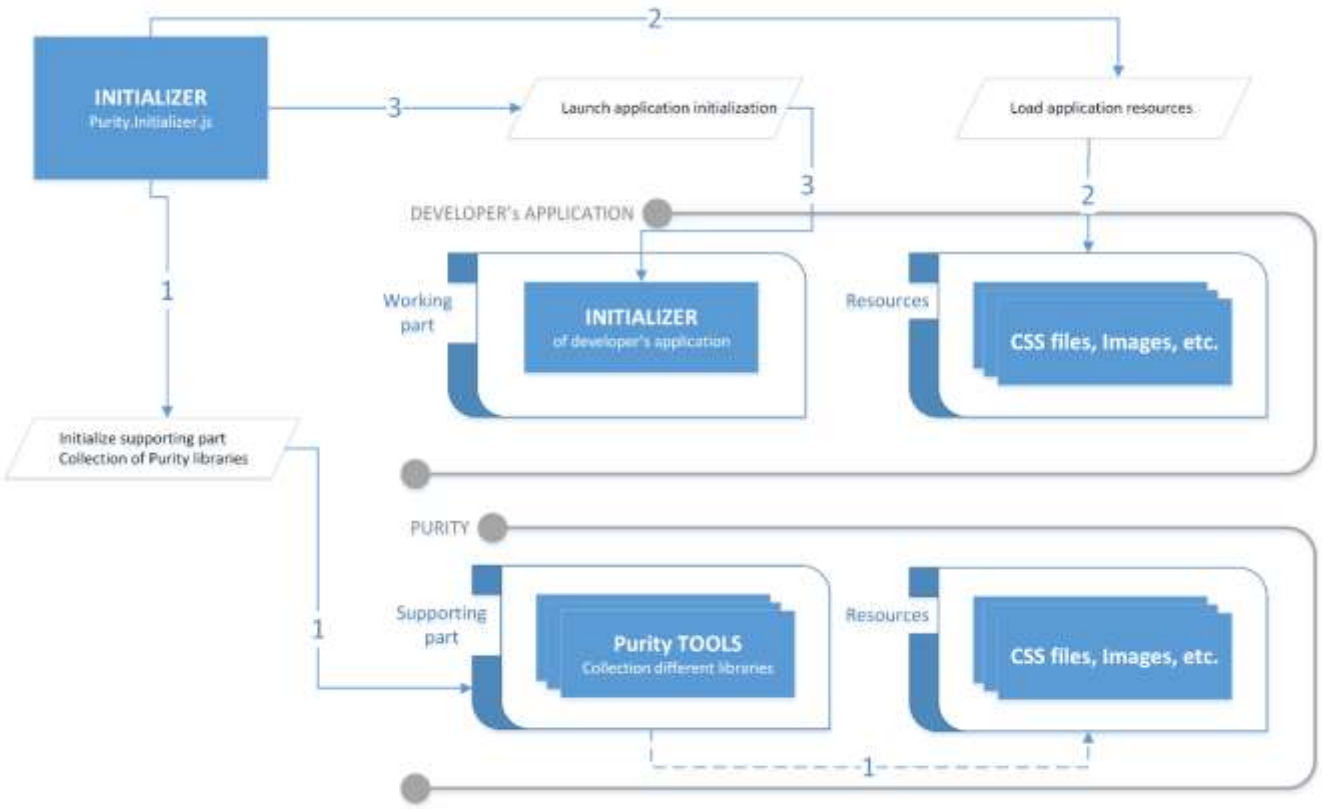
Purity was created for hard project based on AJAX technique. It doesn't mean what Purity cannot use whit other purposes. But Purity will not be effective solution for simple page-to-page sites. Reason of such situation is very simple.

As you've seen in previous article Purity depends from [**Initializer**]. This script should be loaded before using Purity. Always before using Purity. So, if developer want use Purity for simple sites, he should understand – pages will load slower, because each time will be loaded Purity's [**Initializer**].

2. Loading and initialization

2.1. Basic principals

Initialization of application does in three stages. Take a look on pic. 2.0.



Picture 2.0. – Initialization of application.

First stage. Purity is a collection of different libraries. But such libraries depend from each other. For example, to use [Purity.Environment.Events] (controls DOM's events) should be initialized [Purity.Tools] and [Purity.HTML] before [Purity.Environment.Events] will be initialized. So, [Purity.Initializer] controls integrality of all references. It's necessary to don't load unnecessary libraries and use only necessary.

So, on first stage [Purity.Initializer] make list of all necessary libraries and load it. If some library isn't available, [Purity.Initializer] will stop whole process of initializing. If some library (module) needs resource, such library start loading necessary resources in parallel process, without stopping of whole procedure of initialization.

> **NOTE** Purity's resources are loading in parallel process always. Developer's resources can be loaded after Purity's libraries (module) will be ready (like on scheme on pic. 2.0) or in parallel process too. To get more information see "2.3. Configuration of [Purity.Initializer]".

Besides, [Purity.Initializer] prepares libraries for work. [Purity.Initializer] builds prototypes of modules to make it available for using (read more "3. Purity's modules and developer's modules").

Second stage. [Purity.Initializer] are looking for list of necessary resources and load it.

Third stage. When all libraries are prepared and all resources are loaded, [Purity.Initializer] starts application's initialization.

2.2. Attaching [Purity.Initializer]

To make initialization available developer should attach initialization-script - [Purity.Initializer] via next code:

```
<script type="text/javascript" src="~/Kernel/JS/Purity.Initializer.js"></script>
```

Code listing 2.0.

As you can see, it's just link to script. Doesn't matter where this script will be defined: in <BODY> or in <HEAD>. But recommendation is: attach it to <BODY> after configuration script (see 2.3. Configuration of [Purity.Initializer]).

2.3. Configuration of [Purity.Initializer]

2.3.1. Configuration object

[Purity.Initializer] should be configured. It's very easy. Let's see next code.

```
<script type="text/javascript">
    purityConfigurator = {
        Purity: {
            domain  : window.location.host
            modules : {
                url         : "/modules/get",
                JSICPrefix  : "JSIC",
                useOnlyJSIC : true,
                pathToJS    : "/Kernel/JS/",
            },
            resources: {
                url           : "~/Resources/get",
                versions      : "~/Resources/versions",
                basicPath     : ""
            }
        },
        Tasks: {
            modules: {
                list    : ["Controls.Tools", "CSS.Manipulation", "Developer.Logs"],
                mode    : "release",
                useCache: true,
            },
            resources: {
                list                : [
                   { url: "", name: "", path: "", type: "jsic", cache: true, initas: "js"},
                   { url: "", type: "js"     },
                   { url: "", type: "css"    },
                ],
                loadAfterModules    : true,
                versions            : "~/Resources/versions",
            },
            compatibility: {
                localstorage    : true,
                transform       : true,
                animation       : true,
                xmlhttprequest  : true
            }
        },
        Events: {
            onFinish: function ()    { Program.init();    },
            onError : function (e)   { alert(e);          },
```

```
            },
            Visualization: {
                loadExternal: true,
                urlExternal: "http://domain.com/Purity.Initializer.Visualization.Clear.js",
                consoleLogs: true,
            },
            Debug: {
                allowCommandsViaHash         : true,
                showServersResponse          : true,
                showLocalStorageStatus       : true,
                doNotCloseConsoleOnFinish    : true,
                showConsole                  : true,
                clearLocalStorage            : false,
                feedback                     : {
                    url     : "/LogsSend/",
                    email   : "mail@mail.com"
                }
            }
        };
    </script>
```

Code listing 2.1.

To configure [Purity.Initializer] developer has to create global (that's why without directive "var") variable [purityConfigurator].

IMPORTANT	After initialization is finished, [Purity.Initializer] will remove [purityConfigurator] from global namespace and free (a bit) memory.

Configuration object [purityConfigurator] has several groups:

[**Purity**] In this group, developer can configure only settings of Purity.

[**Tasks**] Here developer can define list of Purity's modules and list of developer's resources, which should be loaded. Also in [Tasks] section developer define checking of compatibility.

[**Events**] In event's section developer define events, which will be called after [Purity.Initializer] finishes initialization procedure.

[**Visualization**] This section controls visualization's setting of loading process.

[**Debug**] In this section developer can configure some actions for debugging.

Let's see all properties of each section of configuration object [purityConfigurator]:

[Purity.**domain**] If developer place all Purity's sources into own server this property should has value like in code listing – [**window.location.host**]. It means, that [Purity.Initializer] should search all libraries on current server. To use any other server (for example, which support JSIC transport and version's control) developer should defined (as string) link to Purity's server (for example, "http://developitall.com").

[Purity.modules.**url**] This string will be added to [Purity.**domain**] to build URL for request of Purity's modules. For example, if [Purity.**domain**] = "www.coolsite.com", access URL will be "www.coolsite.com/modules/get?[request's parameters]".

[Purity.modules.**JSICPrefix**] [Purity.Initializer] uses two type of transport: XMLHttpRequest and JSIC format. XMLHttpRequest can be applied only if working domain and Purity's domain are same. And JSIC uses for cross-domains requests. Of course for two types of requests can be used one URL. It can be if server bases on PHP for example. But if we have server via ASP.NET (C#) – here is other situation, because C# is strong-typing language and we cannot return from one method result with different type – we have to use two methods. That's why sometime we need two URLs for different types of request. [Purity.modules.**JSICPrefix**] just will be added to access URL if JSIC is used and our previous URL will be: "www.coolsite.com/modules/getJSIC?[request's parameters]".

[Purity.modules.**useOnlyJSIC**] If this flag is [true], [Purity.Initializer] will use only JSIC transport and for same domains and for cross-domains requests. It can be helpful when we cannot use XMLHttpRequest for some reasons.

IMPORTANT	All properties of [Purity.modules] have owns default values. So developer can define only one of them. Setting of [Purity.modules] will be actual if developer applies such modes as "debug" or "debug-release". And this property gives developer much more freedom with configuration own Purity's server.

[Purity.modules.**pathToJS**] Developer should use this property if Purity's modules are storage in some other place than default [root/Kernel/JS/]. Here developer can define such place.

IMPORTANT	Never define here [Purity.modules.**pathToJS**] server name and protocol. [Purity.Initializer] automatically takes necessary data from [Purity.**domain**].

[Purity.resources.**url**] and [Purity.resources.**versions**]. Some Purity's modules has own resources, like css-files or image-file and etc. To get such resource [Purity.Initializer] use special url, which can be defined in [Purity.resources.**url**]. To make available caching of Purity's resources, [Purity.Initializer] should get access to server's versions manager. Link to such versions manager can be defined in [Purity.resources.**versions**].

IMPORTANT	If developer use any public Purity's server, developer should not define [Purity.resources.**url**] and [Purity.resources.**versions**]. Only in case, when developer use own Purity's server such properties should be defined.

[Purity.resources.**basicPath**]. Some Purity's modules has own resources like css-files or images. In this case [Purity.Initializer] loads such resources like developer's resources via standards for [Tasks.resources.**list**]. According this standard each module, which has some resource, defines [path] to module's resource (absolutely like in [Tasks.resources.**list**] for [type]=["jsic"]). But sometime developer can have different path in server structure to Purity's modules and resources. In this case, developer can define [Purity.resources.**basicPath**] to get access to Purity's resources. For example, if developer define [Purity.resources.**basicPath**] as "folder_1/folder_2/root/", Purity's module will use path to get access to resources: ["folder_1/folder_2/root/" + default_path_defined_in_module], for css-resources it will be "folder_1/folder_2/root/Kernel/CSS".

IMPORTANT	Don't use [Purity.resources.**basicPath**] and don't define [Purity.resources.**basicPath**], if you use any public Purity's server.

	This property can be useful only if developer deploy Purity's module on own server.

[Tasks.module.**list**] This property consists list of all necessary Purity's libraries. Pay you attention. Current project use such libraries like: [Purity.Environment.Events] or [Purity.HTML], but in example such libraries weren't defined. As was said before, [Purity.Initializer] control all reference between modules and module [Purity.Controls.Tools] (which is defined in [purityConfigurator]) uses and [Purity.Environment.Events], and [Purity.HTML]. That's why [Purity.Environment.Events] and [Purity.HTML] will be loaded automatically.

[Tasks.module.**mode**] Developer can define mode of loading resources. Available next:

debug – all resources load as links, like <SCRIPT>, <LINK> and etc. No compress.

release – all resources are compressed and if it possible integrate into page. For example, scripts will not be attached throw tag <SCRIPT>, but script will be generated and integrated into global namespace.

debug-release – all resources are compressed, but attached as link, like <SCRIPT>, <LINK> and etc.

noconnections – If this mode is active, [Purity.Initializer] will not load bodies of any resources. It means what all scripts, all CSS tables, all images and etc. will be attached via tags like <SCRIPT>, <LINK> or <IMG>. In this case, [Purity.Initializer] will not save resource into localStorage of browser to make second (and next) loading faster.

So, if developer use Purity's libraries without any changing of server side (read more "Addendum C. Purity's server") and don't want use any public Purity's server, developer should use mode "*noconnections*". But developer should remember that in this case, caching of resources will be switched off.

IMPORTANT	To use "*release*" and "*debug-release*" modes, Purity's server should support JSIC transport format and has several server-scripts, which control versions of modules (read more "Addendum C. Purity's server").

[Tasks.module.**useCache**] This flag control using of Purity's cache system. [true] – Purity's cache is on; [false] – off. Default value – [true].

IMPORTANT	Purity's cache system and browser's caching are not same things and do not depend from each other.

[Tasks.resources.**list**] This property consists list of developer's resources. For script or CSS file will be enough only link like:

```
{ url: "~/Program/JS/PROGRAM.Tests.js"},
```

Code listing 2.2.

NOTE	In url developer can use symbol "~", like next: [url:"~/resources/versions"]. In this case [Purity.Initializer] modify url like with this: "**domain**/resources/versions". [Purity.Initializer] will take domain information from [Purity.**domain**].

But, if developer want preload image, he should be used next format:

```
{ url: "http://url_to_image", type: "image", id: "developer_image_id"},
```

Code listing 2.3.

In this case images will be placed into virtual storage. To access it, developer can use [id]. Immediately after developer get necessary resource from virtual storage, such resource will be removed from storage.

In addition, developer can get any other data like text or binary data, using next record:

```
{ url: "http://url_to_resource", type: "jsic", id: "developer_resource_id"},
```

Code listing 2.4.

In this case resource will be placed in virtual storage like image. But such technique works only if server supports JSIC transport (read more "Addendum C. Purity's server").

In addition, developer can attach auto-initializing resources. For example:

```
{ url: "basic", name: "some.js ", path: "path", type: "jsic", cache: true, initas: "js"},
```

Code listing 2.5.

Code (2.5) means what [Purity.Initializer] will try get module via url [url + path + name] and will try initialize it as JavaScript. If flag [cache] is [true], [Purity.Initializer] will save such resource into localStorage and get resource from it in next time. Pay your attention in this case, we don't define [id] property. We don't do it, because such resource will be initiated automatically and will not be saved into virtual storage.

IMPORTANT	To use caching of developer's resources, server should support such technique and has version's controlling scripts.

[Tasks.resources.**loadAfterModules**] As was said before, [Purity.Initializer] can load developer's resources. But if such resources (for example script) use some Purity's library such resource should be load only after all Purity's libraries. To control ordering is flag [Tasks.resources.**loadAfterModules**]. If [Tasks.resources.**loadAfterModules**] in [**true**] – all developer's resources will be load only after Purity's libraries are loaded and are initialized. If [Tasks.resources.**loadAfterModules**] in [**false**], [Purity.Initializer] starts load developer's resources in same time as Purity's libraries. Use [**false**] only if you sure, that any Purity's libraries will not be necessary during developer's resources is initializing.

IMPORTANT	Developer should be careful with [Tasks.resources.**loadAfterModules**]. I strongly recommend always use it in [true]. It'll be much safely and gives guaranty, what application will load successfully. Use [Tasks.resources.**loadAfterModules]** in [true] only in case when you have very heavy resource that requires a lot of time to load.

[Tasks.resources.**versions**] If developer wont to use caching system of developer's resources, developer should define url of server's version's controlling scripts. [Purity.Initializer] will send via such url request and wait from server list of all actual versions of resources.

NOTE	In link developer can use symbol "~", like next: [Tasks.resources.**versions**] = "~/resources/versions". In this case [Purity.Initializer] modify url like with this: "**domain**/resources/versions". [Purity.Initializer] will take domain information from [Purity.**domain**].

[Tasks.compatibility.**localstorage**], [Tasks.compatibility.**transform**], [Tasks.compatibility.**animation**], [Tasks.compatibility.**xmlhttprequest**] [Purity.Initializer] can check browser's support for several tech. Code in listing means what browser should support: CSS3 animation, CSS3 transforms, xmlHTTPRequest and localStorage. If browser doesn't support such tech and object, [Purity.Initializer] will stop initialization and show message about tech, which doesn't support.

[Events.**onFinish**] and [Events.**onError**] In [Events.**onFinish**] developer defines script or method, which be run at the end of initialization procedure. Script (or method) which is defined in [Events.**onError**] will be run if some error will be.

IMPORTANT	Pay your attention, in code listing 2.1 use anonymous function, like: **onFinish : function () { PROGRAM.init(); }**. It's necessary because at the moment of defining [purityConfigurator] object [PROGRAM] isn't defined. And if developer make next definition **onFinish : PROGRAM.init();**, it'll call error, because PROGRAM === undefined.

[Visualization.**loadExternal**] and [Visualization.**urlExternal**] [Purity.Initializer] can show current progress of loading and initialization. To make in available developer should set value [true] for [Visualization.**loadExternal**] and defined link to visualization theme in [Visualization.**urlExternal**]. [Purity.Initializer] doesn't use direct access to visualization script, it allows use different themes to showing progress of loading.

[Visualization.**consoleLogs**] During initializing [Purity.Initializer] can write some information into browser console. It can be information about speed of loading or errors. To see such information [Visualization.**consoleLogs**] should be in [**true**].

[Debug.**allowCommandsViaHash**] In [true] [Purity.Initializer] will allow command via hash. Let's see next example.

- www.your_site.com/#mode=debug – Site will be loaded with [Tasks.module.**mode**] in value ["debug"];
- www.your_site.com/#doNotCloseConsoleOnFinish=true – site will be loaded with [Debug.**doNotCloseConsoleOnFinish**] in value [true].
- and etc.

Developer can define via hash next variables:

- [Tasks.module.**mode**];
- [Debug.**doNotCloseConsoleOnFinish**];
- [Debug.**showConsole**].

Value which was defined via hash has priority and change value, what was defined in configuration object.

IMPORTANT	Using hash for switching modes of loading and making available console is very necessary for final debugging process. Developer can initialize site in "release" mode and switch to "debug" in any time and from any device.

[Debug.**showServersResponse**] In [true] [Purity.Initializer] will show in browser's console and external console content of server's responses.

[Debug.**showLocalStorageStatus**] In [true] [Purity.Initializer] will show in browser's console and external console status of all operations with localStorage.

[Debug.**doNotCloseConsoleOnFinish**] In [true] [Purity.Initializer] will not run handle defined in [Events.**onFinish**] and user can read logs in external console. This option will be work only if external console is defined ([Visualization.**urlExternal**]) and supports manual closing.

[Debug.**showConsole**] Some external consoles support two modes: simply and console with logs of loading process. If [Debug.**showConsole**] is [true], [Purity.Initializer] tries switch external console to mode of logs viewing.

[Debug.**clearLocalStorage**] In [true] [Purity.Initializer] will clear localStorage object of browser each loading of site.

> **NOTE** In most PC browsers, we can clear localStorage without any problem. We can choose necessary command from menu or just write [window.localStorage.clear()] in console of browser's debugger. But as for mobile browsers and browsers for tablets, here situation is a little harder – in most cases we haven't browser's debugger. In such situation property [Debug.**clearLocalStorage**] will be very useful.

[Debug.feedback.**url**] Some external consoles support sending logs to developer. It's very useful for debugging application on mobile devices (where aren't console and keyboard). Property [Debug.feedback.**url**] should consist url to server script, what controls mail sending.

[Debug.feedback.**email**] It's destination address. On this address will be send logs.

At this moment, you as developer has one question (I'm sure): which properties of [purityConfigurator] are obligatory and which one can be undefined. Answer: all properties can be undefined, and all properties have default values.

2.3.2. Default values of configuration of [Purity.Initializer]

Default value will be defined if developer does not define some property in [purityConfigurator].

Property	Type	Default value	Possible values
Purity	object	-	-
domain	string	window.location.host	any
modules	object	-	-
url	string	"/modules/get"	any
JSICPrefix	string	"JSIC"	any
useOnlyJSIC	boolean	False	true / false
pathToJS	string	"/Kernel/JS/"	any
resources:	object	-	-
url	string	"~/Resources/get"	any
versions	string	"~/Resources/versions"	any
basicPath	string	""	any
Tasks	object	-	-
modules	object	-	-
list	array	Null	any
mode	string	"release"	"debug", "release", "debug-release", "noconnections"
useCache	boolean	True	true / false
resources	object	-	-

list	array	null	any
loadAfterModules	boolean	true	true / false
versions	string	“”	any
compatibility	object	-	-
localstorage	boolean	true	true / false
transform	boolean	true	true / false
animation	boolean	true	true / false
xmlhttprequest	boolean	true	true / false
Events	object	-	-
onFinish	function	null	
onError	function	null	
Visualization	object	-	-
loadExternal	boolean	false	true / false
urlExternal	string	“”	any
consoleLogs	boolean	true	true / false
Debug			
allowCommandsViaHash	boolean	false	true / false
showServersResponse	boolean	false	true / false
showLocalStorageStatus	boolean	false	true / false
doNotCloseConsoleOnFinish	boolean	false	true / false
showConsole	boolean	false	true / false
clearLocalStorage	boolean	false	true / false
feedback	object	-	-
url	string	“”	any
email	string	“”	any

Pay your attention, that defaults values of Purity are for C# server.

2.3.3. Configuration of [Purity.Initializer] for debug

To debug all script via browser’s debugger developer should do next:

1. Set value of [Tasks.module.**mode**] to “debug”.
2. Attach all script in resources section ([Tasks.resources.**list**]) just as link (like this: [{ url: "~/Program/JS/PROGRAM.Tests.js"}] or this: [{ url: "~/Program/JS/PROGRAM.Tests.js", type: “js”}].

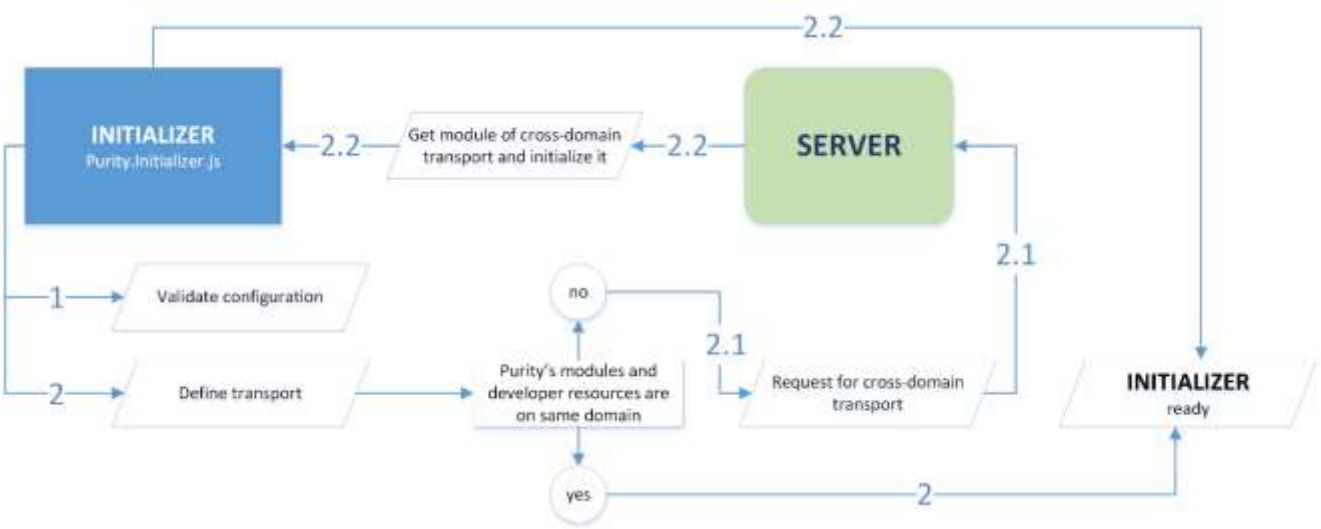
In this case, all module’s scripts and all resource’s scripts will be attached via tag <SCRIPT> and developer get possibility to debug scripts via browser’s debugger. Of course, in this case [Purity.Initializer] doesn’t cache any script.

2.4. Loading

It isn’t logic mistake. Procedure of loading should be describe after description of configuration of Purity’s initializer. It allows developer better understand procedure of loading and Purity’s modules and resources of project.

2.4.1. Preparing

Of course [Purity.Initializer] cannot start loading modules and developer’s resources immediately. Before start [Purity.Initializer] should do two activities: initialize configuration and apply transport for modules and resources.



Picture 2.1. – [Purity.Initializer] preparing

Stage 1. Configuration. On scheme (pic. 2.1.), you can see this stage by number 1. On this stage [Purity.Initializer] reads [purityConfigurator] (read more "2.3. Configuration of [Purity.Initializer]") and validates all properties of it. If developer didn't define something [Purity.Initializer] applies default settings.

Stage 2. Transport. On scheme, you can see this stage by number 2. Unfortunately we cannot use XMLHTTPrequest for any situations. For example, XMLHTTPrequest cannot be used for cross-domain requests. Of course, it isn't impossible. Some developers choose way with XMLHTTPrequest and make it available for cross-domain requests (it isn't so hard like can be seem). But in any case browsers don't like it and will "fight" with it always. And browser's position in this question makes using XMLHTTPrequest not effective (in this case we have to follow by browser's politic, update our transport time to time).

That's why Purity uses JSIC transport for cross-domain requests. This transport is based on using of tag <STRIPT> and will work in any browser. But it requires some server's code.

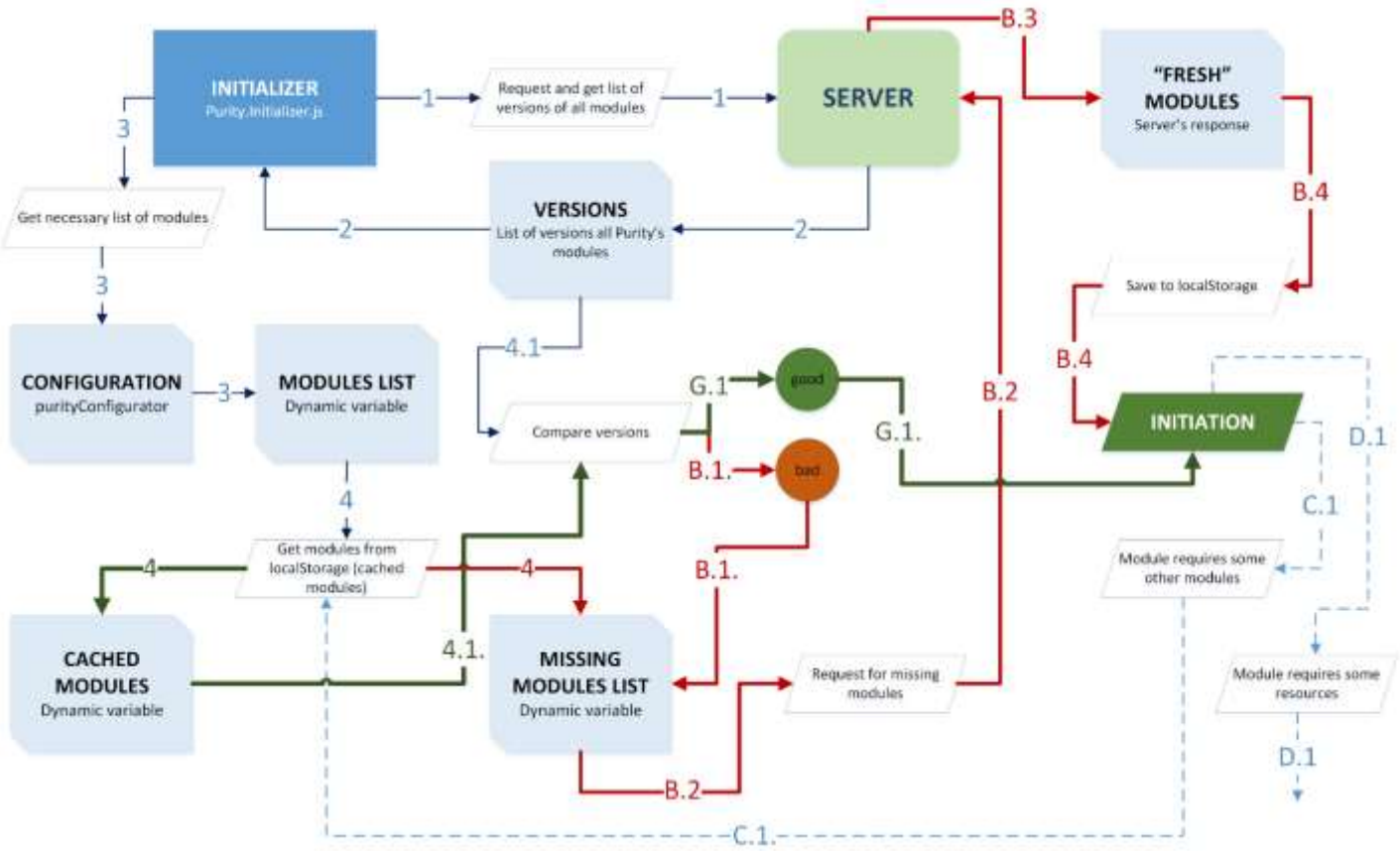
So on stage 2 [Purity.Initializer] discover all links (of modules and resources, and link of own domain) and if everything is on one server does nothing. But if own domain and Purity's domain (or resource's domain) aren't same, [Purity.Initializer] initialize JSIC module to make available cross-domain requests.

It should be clear: if JSIC transport module isn't available, [Purity.Initializer] will not continue loading.

2.4.2. Load and build Purity's modules

When configuration of [Purity.Initializer] is ready, [Purity.Initializer] can start load resources. First step: load all Purity's modules (which were defined in [purityConfigurator]) and build these modules (make it available for developer's scripts). This step was shown on scheme (pic. 2.0) by number 1 in "2.1. Basic principals".

So, let's see how it works via scheme on pic. 2.2.



Picture 2.2. – Loading Purity's modules

Steps 1 and 2. [Purity.Initializer] gets from server list of module's versions. It's necessary to understand, when [Purity.Initializer] should update module from own cache. Such list isn't heavy. It's array of names and versions on modules. This request to server will be done very fast.

Step 3. [Purity.Initializer] gets from [purityConfigurator] list of necessary modules. Such list was defined in [module] property (read more "2.1. Basic principals").

Step 4. [Purity.Initializer] try to find necessary modules in own cache (localStorage) and make two lists: cached modules and missing module.

> **IMPORTANT** If Purity's caching system is off ([Tasks.module.**useCache**] of [purityConfigurator] in [false] or [Tasks.module.**mode**] is "debug"), [Purity.Initializer] will place all defined (in [Tasks.module.**list**] property) modules in missing list.

Step 4.1. [Purity.Initializer] compare version of cached module with version form server.

Step G.1. If version of cached module is actuated, [Purity.Initializer] build module (start initialization of module).

Step C.1. If initializing module requires some other module, [Purity.Initializer] stops initialization procedure and back to request additional modules form step 4 (looking for it in localStorage).

Step B.1. If version of cached module is not actuated, [Purity.Initializer] adds such module in missing list.

Step B.2 and B.3. On these steps [Purity.Initializer] makes request to server for missing modules and get it from server using actuality transport.

Step B.4. On this point [Purity.Initializer] saves gotten from server modules in cache (localStorage). If Purity's caching system is off ([Tasks.module.**useCache**] of [purityConfigurator] in [false]), [Purity.Initializer] will not save any data in cache. Pay your attention [Purity.Initializer] will save data even if [Tasks.module.**mode**] is "debug".

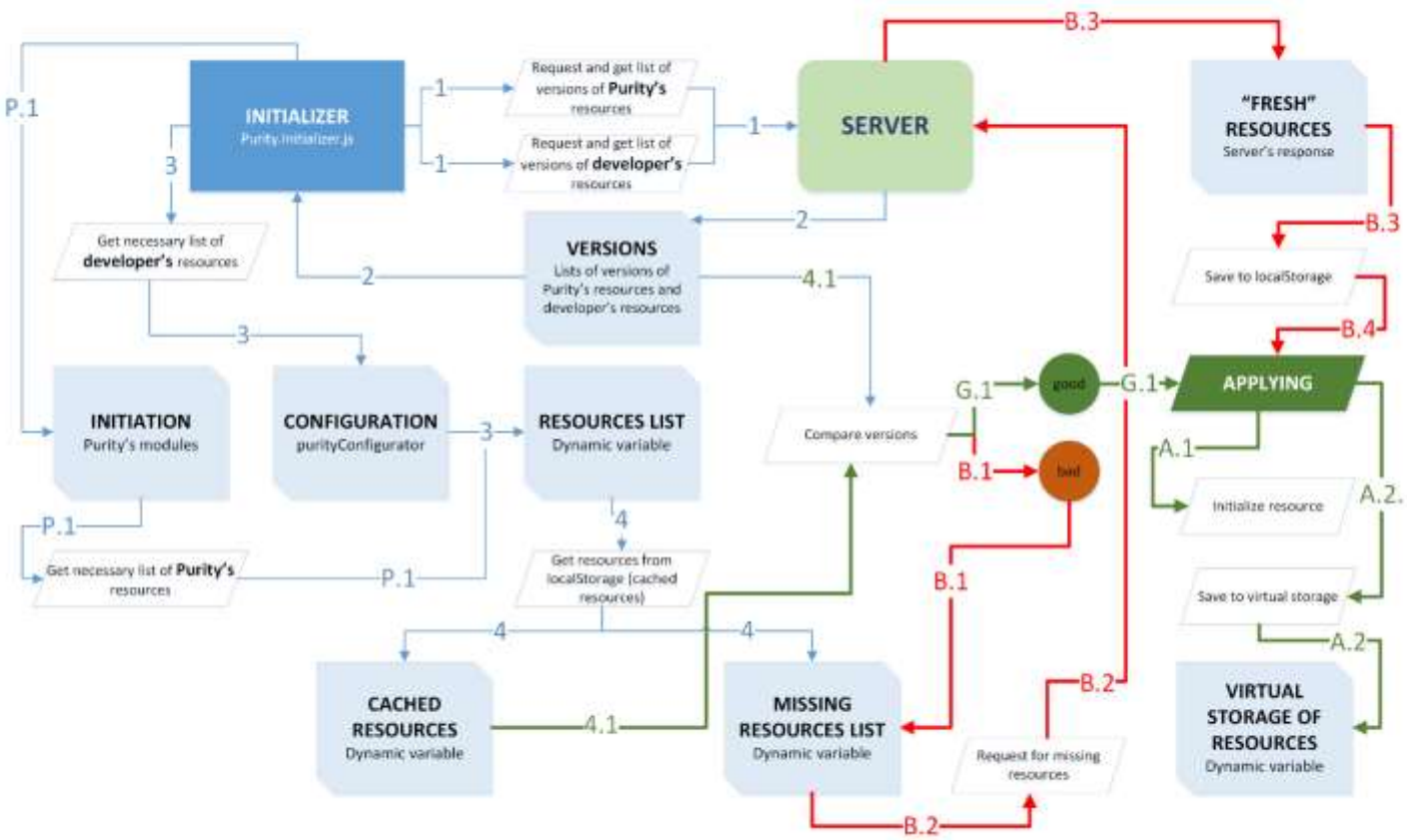
Step D.1. If initializing module requires some resources, [Purity.Initializer] doesn't stop initialization procedure, but start procedure of loading resource in parallel process. Take your attention, on step C.1. initialization will stop, but here (step D.1) initialization will continue.

As you can see, procedure of loading Purity's modules isn't so hard like it can be seem. By green line shows effective way of loading, when site was loaded several times before and cache is updated. Red way isn't bad way. Red way is first loading way or path of updating some module.

2.4.3. Load and applying resources

Firstly, can be Purity's resources and developer's resources (read more "2.1. Basic principals"). As for Purity's resources, such resources always load in parallel process with initialization. But developer's resources can be loaded after basic initialization process or parallel it (which mode will be used, should be defined in configuration property [Tasks.resources.**loadAfterModules**], read more "2.3. Configuration of [Purity.Initializer]").

On next scheme, you can see description of loading process.



Picture 2.3. – Loading resources

Step 1 and 2. [Purity.Initializer] makes two requests to get information about actual version of Purity's resources and developer's resources. As you see, developer can use different servers for Purity and own resources. When [Purity.Initializer] get necessary information from server, [Purity.Initializer] makes list of actual versions.

Step 3 and P.1. List of developer's resources [Purity.Initializer] get from configuration property [Tasks.resources.**list**] (step 3.). But list of Purity's resources is building dynamically (step P.1). As you remember from scheme on pic. 2.2. some module can request own resources during initialization (step D.1 on pic 2.2). As result of steps 3 and P.1. [Purity.Initializer] makes list of resources in real time mode.

Step 4. [Purity.Initializer] tries finding necessary resource in cache – localStorage. If resource is found – it's list of cached resources; and if isn't – list of missing resources.

> **IMPORTANT** [Tasks.module.**useCache**] is a global configuration's setting. If it's [false] caching will not apply not during Purity's resources are loading, not during developer's resources are loading.

Step 4.1. [Purity.Initializer] validate versions on cached resource with actual version in version's list.

Step G.1. If version of resource is actual, [Purity.Initializer] applies resource. Resource can be applied via different ways.

Step A.1. First way is initialization of resource. [Purity.Initializer] supports initialization of js-content and css-content. In this case, [Purity.Initializer] will apply resource, but will not save it in virtual storage.

Step A.2. Second way is saving resource in virtual storage. It allows get value of resource in any time after resource was loaded.

Step B.1. If version of cached resource isn't actual, such resource will be added into list of missing resources.

Step B.2. and B.3. [Purity.Initializer] makes request to server for missing resources and get it from server with actual version's information.

Step B.4. [Purity.Initializer] save (cache) updated resource into localStorage.

Step B.5. [Purity.Initializer] applies resources.

As you have seen, some resources can be applied. Purity supports applying for js-content and css-content. After applying css-content [Purity.Initializer] create styles via content of css using tag <STYLE> (it appends to <HEAD>). As for applying of js-content, [Purity.Initializer] run (without using of [eval]) js-content.

> **NOTE** It makes sense to use applying of js-content only if js-content is template of some module, because js-content will run only once.

2.5. Server's scripts

If you read "2.3. Configuration of [Purity.Initializer]" you know – different modes of [Purity.Initializer]'s work depend from server side. If developer uses any public Purity server – here is no problems. But sometime via some reasons developer has to share Purity on own server. In this case, developer has to add some functionality to server.

At the moment of creating this documentation for Purity's server were developed two packages of scripts:

- for PHP;
- for ASP.NET(C#) / ASP.MVC (C#);
- for node.js;

But here will be description only package for PHP. So package include next files:

NAME	DESCRIPTION
PUBLIC PART	**This folder should be shared for external requests**
Purity.Modules.Get.php	includes only calling of classes of Purity.Modules.Resources.php
Purity.Resources.Get.php	includes only calling of classes of Purity.Resources.Controller.php
Purity.Resources.Versions.php	includes only calling of classes of Purity.Resources.Controller.php
Purity.Configuration.php	get and validate configuration form [paths.xml]
Purity.SendLogs.php	allows send logs of loading to developer
paths.xml	configuration file
PRIVATE PART	**Strongly recommend deny external access to such folder**
Purity.JSICConvertor.php	includes classes of JSIC transport format
Purity.Strings.php	include classes for working with strings
Purity.Resources.Controller.php	manage resource's content and versions of resources
Purity.Modules.Resources.php	manage Purity's modules and versions of modules
Purity.Mail.php	manage mail sending

Take a look on code of [Purity.Modules.Get.php] (code listing 2.6). As you can see here is nothing except definition of Controller's class. Separation of access point and working code was done for effective secure.

```php
<?php
namespace Purity\Modules {
	session_start();
		ini_set('display_errors',1);
		error_reporting(E_ALL);
		require_once('Purity.Configuration.php');
		$Configuration = new \Purity\Configuration('paths.xml');
		$Configuration = $Configuration->get();
		if (is_null($Configuration) == false){
			require_once($Configuration['paths']->
					privatePHP.'Purity.Modules.Resources.php');
			$PurityModules = new Controller\Controller(
				$Configuration['paths']->PurityJS,
				$Configuration['paths']->privatePHP,
				$Configuration['urls']->PurityJS,
				$Configuration['urls']->protocol
			);
			$PurityModules->getModules();
		}else{
			echo "Purity modules::: cannot load paths file or bad file structure.";
		}
}
?>
```

Code listing 2.6.

Also developer should pay attention on constructor of [Purity\Configuration] class (see code listing 2.7).

```php
		$Configuration = new \Purity\Configuration('paths.xml');
		$Configuration = $Configuration->get();
```

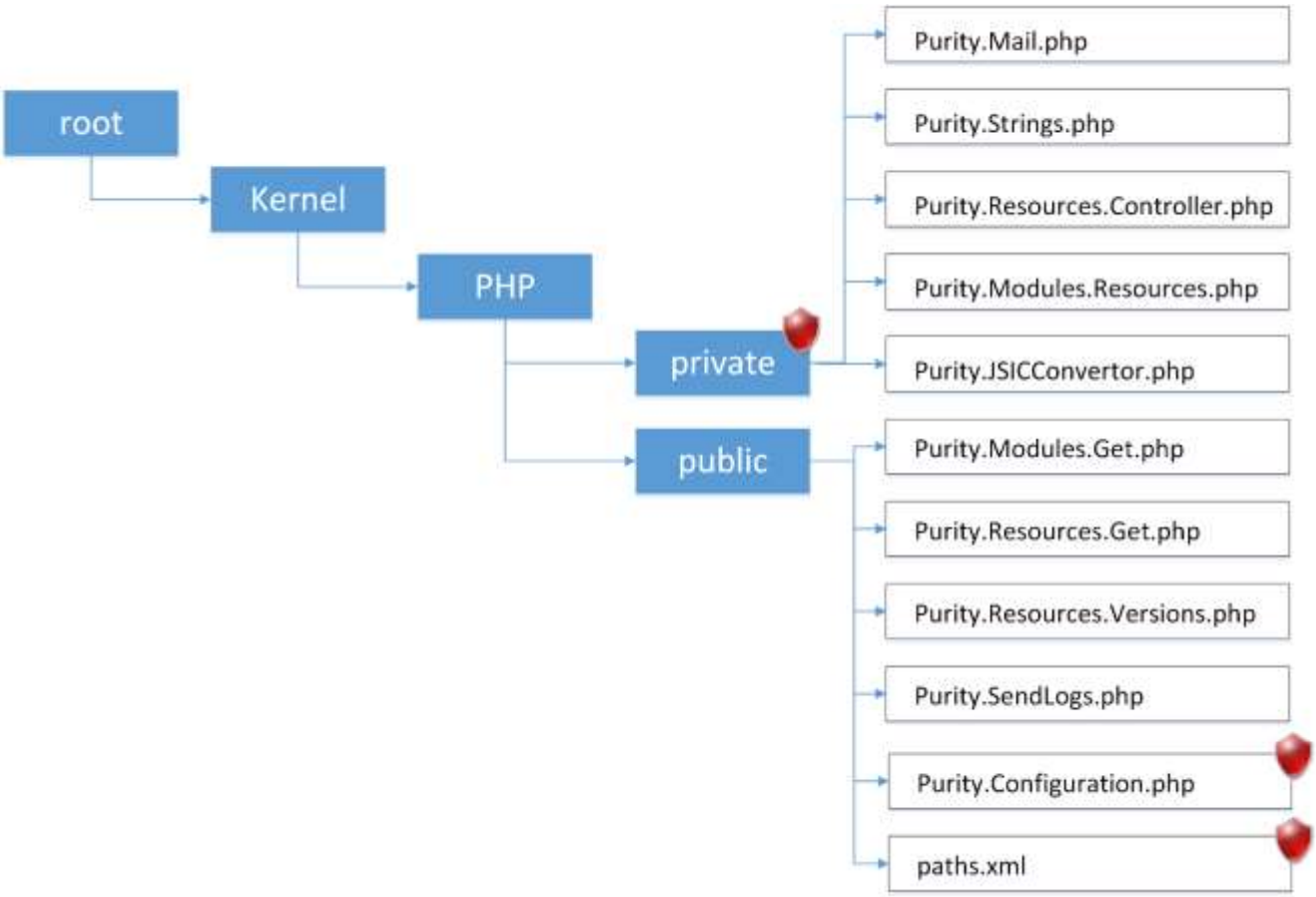
Code listing 2.7

Here is one important thing. If developer have decided change name of configuration file [paths.xml] or change his placement, developer should change name or/and path to this file here and only here.

IMPORTANT	Pay your attention, all paths are configured in configuration files only. Developer don't must discover php-resources – there are no any paths. But if developer have made decision change name of php-recourse, he has to modify all name of files in php-resources.

Strongly recommend place [Purity.Modules.Resources.php], [Purity.Resources.Controller.php], [Purity.JSICConvertor.php] and [Purity.Strings.php] in folder like [private] and deny free access to it. And [Purity.Modules.Get.php], [Purity.Resources.Get.php], [Purity.Resources.Versions.php], [Purity.Configuration.php] can be placed in [public] folder with external access. Also, recommend close free access to configuration files: [paths.xml] and [Purity.Configuration.php].

In any case, recommended folder structure for server's scripts is next:



Picture 2.4 – Server's folders structure.

If developer uses recommended structure, will not be necessary modify paths in PHP's configuration file, but developer can do it without any troubles.

Let's see structure of configuration file.

```
<?xml version="1.0" encoding="utf-8"?>
<data>
	<paths>
		<privatePHP>\Kernel\PHP\private\</privatePHP>
		<PurityJS>\Kernel\JS\</PurityJS>
```

```
            <addDOCUMENT_ROOT>yes</addDOCUMENT_ROOT> <!-- no || yes -->
        </paths>
        <urls>
            <PurityJS>/Kernel/JS/</PurityJS>
            <protocol>http</protocol>
        </urls>
        <cache>
            <resources_versions>versions.cache</resources_versions>
        </cache>
</data>
```

Code listing 2.8

As you can see it's simple *.xml file with a few sections.

Section <**paths**> includes data about physical path to files on server and consists:

<**privatePHP**> – path to private folder, like on pic. 2.4;

<**PurityJS**> – path to folder with *.js file of Purity's modules;

<**addDOCUMENT_ROOT**> – if this flag in [yes], script automatically adds path to root folder of server. Such path will be taken from `$_SERVER['DOCUMENT_ROOT']`. If this flag is in [no], script will use only paths, which were defined in <**privatePHP**> and <**PurityJS**> (without adding path to root folder).

Section <**urls**> includes data about URLs to Purity's files on server and consists:

<**PurityJS**> – URL to folder with *.js file of Purity's modules;

<**protocol**> – protocol of server: [http] or [https];

Section <**cache**> includes data about some setting of caching system:

<**resources_versions**> – this field consists name of file, which is created by scripts to storing information about versions of modules and resources.

IMPORTANT Pay your attention, for building URLs to Purity's js-resources and physically access to it can be used different paths (one of them in <**paths**> section and other in <**urls**> section). It allows developer to create virtual folders or make virtual URLs, what will be necessary for stronger security.

2.6. Folders structure on server side

Library "Purity" (or maybe framework, but not library or platform – how knows) includes files of two types: JavaScript-files and CSS-files.

NOTE This article can be useful for developer, if developer deploy Purity on own server and doesn't use any public Purity's server.

Count of JS-files and CSS-files is unlimited in two reasons: Purity is increasing and Purity is modular. It means what for each task can be use some module of Purity. Such module can be loaded or not. That's why here cannot be shown full list of Purity's modules, but most important (basic) modules are next:

Purity.Initializer.js – It's loader. This module loads and initialize other modules and developer's resources. It'll be truth, if we say – it's main module.

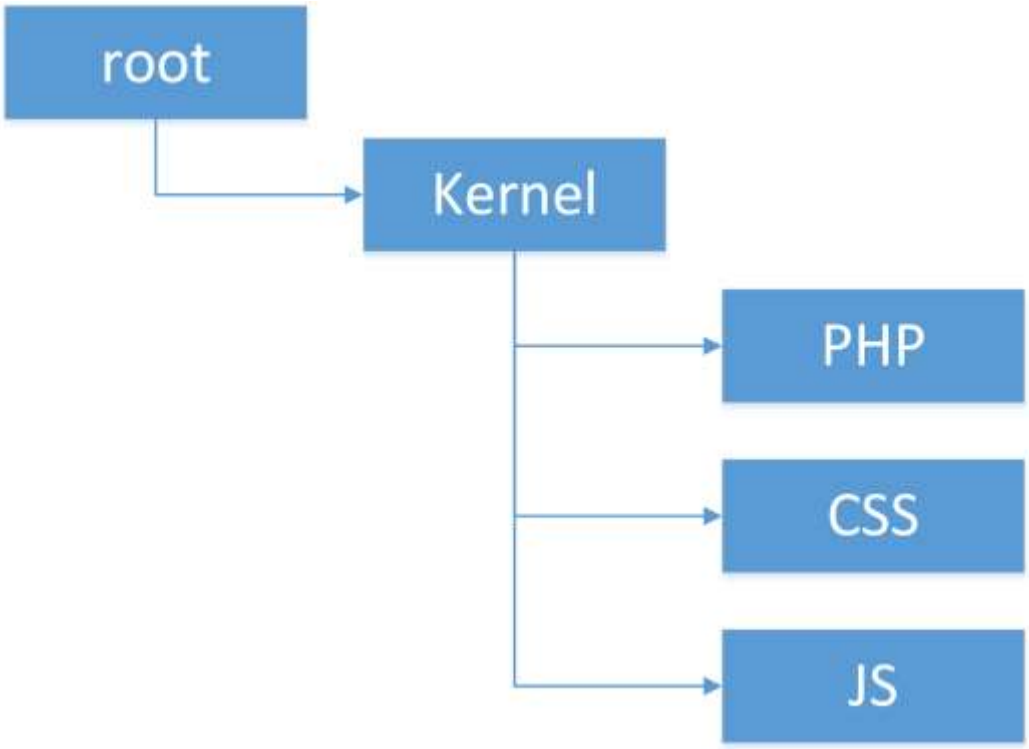
Purity.Environment.Events.js – this module controls all events.

Purity.Environment.Overhead.js – this module help manage properties of objects.

Purity.Tools.js – this is collection of most popular methods.

Purity.HTML.js – this is collection of most popular methods for work with DOM elements.

All Purity's sources can be storage on server in any folder, which should be defined in server's scripts. But if developer uses recommended folder structure, will not be necessary modify any server scripts.



Picture 2.5 – Server's folders structure of Purity's sources.

As you've read before, folder [PHP] consists server's scripts. If developer don't use PHP variant of serves's scripts, but .net MVC (C#), in this case such server's scripts will be storage according other rules. In any case, folder [PHP] will be in [Kernel], only if developer uses PHP variant of serves' scripts.

Folder [CSS] storages only CSS- files. And folder [JS] storages only JS-sources.

3. Purity's modules and developer's modules

3.1. Developing via modules

One of the best programming practice is using modules for group some code by functional of purpose of using. Developer should remember what Purity could be used for any practices, but it was created for using module's style first.

So, to make effective and clear application with Purity, developer should place own code into different module. As usual, one module should be created only for single functionality (uniform functionality) or for controlling only one-type element. For example, if we are developing some code and have to make some control center of events, we should ask our self:

1. Can we use such control center in a future in some other situations?
2. Is value of necessary code large?

In most cases, we will get two "yes". In this situation, we should make new module with necessary functionality.

3.2. Nice template of developer's module

Take a look on next code.

```
//Check global name
if (typeof GLOBALNAME         !== "object") { GLOBALNAME            = {}; }
if (typeof GLOBALNAME.SubName !== "object") { GLOBALNAME.SubName    = {}; }
//Initializer method
GLOBALNAME.SubName.NAMEOFMODULE = (function () {
//---< Declaration block          >--[begin]--
//---< Declaration block          >--[end]--
//---< Private part               >--[begin]--
//---< Private part               >--[end]--
//---< Public part                >--[begin]--
//---< Public part                >--[end]--
//---< Init part                  >--[begin]--
//---< Init part                  >--[end]--
//---< Return declaration         >--[begin]--
       return {
       }
//---< Return declaration         >--[end]--
}());
```

Code listing 3.0.

As you can see, template is self-extract function. It means what this module will initialize itself. It's very comfortable, because we shouldn't control initializing of module and memory usage.

To understand clear all segments of module let's see it with life example. Below you will find body of [PROGRAM.Widgets.Surface] (see full source PROGRAM.Widgets.Surface.js) from Widget's project.

```
//Check global name
if (typeof PROGRAM         !== "object") { PROGRAM            = {}; }
if (typeof PROGRAM.Widgets !== "object") { PROGRAM.Widgets    = {}; }
/*Build module*/
PROGRAM.Widgets.Surface = (function () {
    "use strict";
//---< Declaration block          >--[begin]--
              //Declaration references
       var    HTML                = new Purity.initModule("HTML"),
              Tools               = new Purity.initModule("Tools"),
```

```
            CSS                 = new Purity.initModule("CSS.Manipulation"),
            Events              = new Purity.initModule("Environment.Events"),
            DeveloperLogs       = new Purity.initModule("Developer.Logs"),
            controlsTools       = new Purity.initModule("Controls.Tools"),
            CSSManipulation     = new Purity.initModule("CSS.Manipulation"),
            //Declaration module's blocks
            Configuration       = {},
            Render              = {},
            Map                 = {},
            Grid                = {},
            Workspace           = {},
            Mounting            = {},
            Logs                = {},
            publicRender        = {},
            publicParameters    = {},
            publicMounting      = {},
            publicMap           = {};
//---< Declaration block          >--[end]--
//---< Private part               >--[begin]--
       Configuration = {
           Logs        : { },
           Grid        : { },
           CSS         : { },
           Animation   : { },
           Parameters  : { }
       };
       Render = {
           init    : function (params) { },
           Sizer   : {
               set : function () { }
           }
       };
       Map = {
           data    : { },
           methods : { }
       };
       Grid = {
           Render  : { },
           Sizer   : { },
           Select  : { },
           States  : { }
       };
       Workspace = {
           init: function () {},
           Sizer: {},
           Switcher: {
               Events      : { },
               State       : { },
               Render      : { },
               switchTo    : function (horizontal, vertical, onlyData) { },
               left        : function (onlyData) { },
               right       : function (onlyData) { },
               up          : function (onlyData) { },
               down        : function (onlyData) { },
               PoinerSwitch: { }
           }
       };
       Mounting = {
           Movement: { },
           Resize  : { },
           Events  : { },
           Mount   : { }
```

```
        };
        Logs = {};
//---< Private part                >--[end]--
//---< Public part                 >--[begin]--
        publicMap = {
            update          : Grid.Render.update,
            updateSelection : Grid.Render.updateSelection
        };
        publicRender = {
         init : Render.init
        };
        publicParameters = {
         get : Configuration.Parameters.methods.get
        };
        publicMounting = {
            append      : Mounting.Mount.append,
            avaliable   : function (rows, columns) {
                return Mounting.Mount.mountInFree(rows, columns, true);
            },
            position    : function () {
                return Grid.Select.Current.get();
            },
            isFree      : Mounting.Mount.isFree,
            mount       : function (row, column, rows, columns) {
                return Mounting.Mount.mount(row, column, rows, columns, "mount");
            },
            unmount       : function (row, column, rows, columns) {
                return Mounting.Mount.mount(row, column, rows, columns, "unmount");
            },
        };
//---< Public part                 >--[end]--
//---< Init part                   >--[begin]—
        Mounting.Events.attach();
//---< Init part                   >--[end]--
//---< Return declaration          >--[begin]--
        return {
            init        : publicRender.init,
            parameters  : publicParameters,
            mounting    : publicMounting,
            page        : {
                right   : Workspace.Switcher.right,
                left    : Workspace.Switcher.left,
                up      : Workspace.Switcher.up,
                down    : Workspace.Switcher.down,
                offset  : Configuration.Parameters.methods.getPageOffset
            },
            offset      : Configuration.Parameters.methods.getSurfaceOffset,
            map         : publicMap
        }
//---< Return declaration          >--[end]--
}());
```

Code listing 3.1.

Declaration block. Here we declare all global (in borders of module) variables. Pay your attention we declare links to Purity's modules. All necessary links are declared in function's body.

> **NOTE** Pay your attention, we declare only Purity's modules, but not other developer's module. Why? Here we see template of seft-initialize module, it means what our module's script will be run immediately when will be loaded. In such situation we cannot know is some other developer's module is available (initialized) or not. But Purity's modules have own constructor, what allow be sure – this module will be available.

Private part. It's body of module. Here is code of module, which is separated by functionality. JavaScript allows place here some object (myObj = {};) without declaration in declaration block and it will not be syntax error, but it'll be structure error,. Such object will be placed in global namespace.

Public part. It's like interface in C# or C++. JavaScript has not such structure like private or public as C# has or C++. We cannot secure our property from changing. But we can secure it via adding intermediate. This block necessary to make links on it in block "return declaration".

Init part. At this place we run some methods, which are necessary for work whole module. As you understand such methods will be run only once.

Return declaration. This block consists all what will be available for other module and any code outside of module. It's analog of public in C# or C++.

After developer's module is initialized, we get access to functionality of module.

```
    PROGRAM.Widgets.Surface.page.left();
    PROGRAM.Widgets.Surface.map.update();
    PROGRAM.Widgets.Surface.mounting.append();
```

Code listing 3.2.

A little more about public part and return declaration. Let's see a situation. We have two next functions:

```
function a() {
    PROGRAM.Widgets.Surface.mounting.append = null;
};
function b() {
    PROGRAM.Widgets.Surface.mounting.append();
};
a();
b();
```

Code listing 3.3.

Function [a] starts first. This function rewrite (change) property of PROGRAM.Widgets.Surface module, and set into property [mounting.append] null. So when function [b] starts, it will call error, because [mounting.append] isn't function and equals null. It means what after [a] was run, nobody can call [mounting.append].

But let's see code listing 3.1 once more. Function [a] change property of [publicMounting], but [Mounting.Mount.append] still works inside of [PROGRAM.Widgets.Surface]. That's why public part is necessary.

3.3. Template of Purity's module

Purity's module has own template (see code listing 3.4).

```
/// <reference path="~/Kernel/JS/Purity.Initializer.js" />
/// <module>
///     <summary>
```

```
///
///     </summary>
/// </module>
(function () {
    //Check Purity.Initializer. Purity should be inited at this moment.
    if (typeof Purity !== "undefined") {
        //Init module prototype and init function
        Purity.createModule("ModuleName",
            //Check references
            ["", ""],
            //Prototype part
            function () {
            /// <summary>Discription of library</summary>
//---< Declaration block          >--[begin]--
                var name        = "Purity::: ",
                    version     = "1.0",
                    lastUpdate  = "29.05.2013",
                    author      = "Dmitry Astafyev";
                    //Declaration module's blocks
                    //Declaration references
//---< Declaration block          >--[end]--
//---< Private part               >--[begin]--
//---< Private part               >--[end]--
//---< Public part                >--[begin]--
//---< Public part                >--[end]--
//---< Init part                  >--[begin]--
//---< Init part                  >--[end]--
//---< Return declaration         >--[begin]--
                return {
                    AboutModule: {
                        getName         : function () { return name;        },
                        getVersion      : function () { return version;     },
                        getLastUpdate   : function () { return lastUpdate;  },
                        getAuthor       : function () { return author;      }
                    },
                }
//---< Return declaration         >--[end]--
            },
            //Init function
            function () {
            }
        );
    }
}());
```

Code listing 3.4.

As you can see, structure of supporting module is same like developer's module has, but it isn't self-initializing module.

Initialization of support module is controlled via [Purity.**createModule**] function. This function gets three arguments:

1. [**name**] - name of current module.
2. [**references**] - array of names other modules, which are necessary for work current module.
3. [**body**] - Body of current module.
4. [**initializing method**] - Initializing function.

So, how does it work?

1. [Purity.**createModule**] check availability of each module in [**references**]. If some module isn't initialized, [Purity.**createModule**] initialize it before continue. If [Purity.**createModule**] cannot find or initialize some module from [**references**], procedure of initialization will be stopped.
2. On second step [Purity.**createModule**] creates prototype of module and save it in virtual storage.
3. And last step [Purity.**createModule**] run [**initializing method**].

As you can see, here is only three simple steps to create new supporting module. But I'm sure, you have a question: what is differences between [**initializing method**] and [**init part**] (in body of module). [**initializing method**] will be run only once during module's initialization. All methods in [**init part**] runs each time developer initialize access to module.

And last thing about supporting module is access to such modules. Developer cannot make simple link like he can do it with working modules. As was said before, supporting modules use prototype. That's why each link to supporting module should be created like object in JavaScript (see code listing 3.5).

```
var HTML = new Purity.initModule("HTML");
```

Code listing 3.5.

After such procedure object [**HTML**] will has all properties, which were defined in [Return declaration] block of module's body.

4. Useful Purity's modules (libraries)

4.1. Storage of properties

4.1.1. Principals

JavaScript if very flexible. This can help create very effective application and in time very poor application, because JavaScript doesn't control developer. Developer can do everything with JavaScript, everything, what he wants. In such situation, developer should develop very carefully and deep.

One of important moment is a storage of properties. Sometime developer has to save some value of some property. For example current state of element. In any other syntax developer create some variable (as property of class or global) and save value into such variable. But JavaScript is a freedom and everything in JavaScript is object, object without open access. So, JavaScript allow add and remove own properties to any object. And developer can add new property for example to any DOM element. It's very useful and comfortable – we should not create any global variable, but save all as property of object.

For example. We have widget. When we activate resize/movement functionality, we should save such state of our widget. JavaScript allows us just make new property of <DIV> (container of widget) and read / write it. It's very nice. Isn't it?

But here are hiding two potential problems.

First. Memory. JavaScript allows everything and does not control developer. As result, nobody doesn't control memory usage.

Second. Conflicts. Who knows, does some property exists or not before developer add new one.

Developer has very low risk to meet such problems in little projects. But if we have big project with several developers such risks are very-very high.

So, how we can organize storage of developer's properties? Answer is very simple: make our own standard and follow it all time in anywhere. Here are several points:

- Use common root-name for all properties.
- Remove property if it isn't necessary.

It means, if we want create some new property and attach it to some <DIV>, firstly we create root-name, for example: [myRootStorage]. And after attach our property to [myRootStorage].

```
var layerDIV = document.createElement("DIV");
document.body.appendChild(layerDIV);
layerDIV.myRootStorage               = {};
layerDIV.myRootStorage.newProperty0 = "something here";
layerDIV.myRootStorage.newProperty1 = "something here";
layerDIV.myRootStorage.newProperty2 = "something here";
```

Code listing 4.0.

Let's see on code in listing 4.0. We create <DIV>, create root-property and save into it everything, what we want. JavaScript allows use other syntax to do same (see code listing 4.1).

```
var layerDIV    = document.createElement("DIV"),
    properyName = "newProperty2";
document.body.appendChild(layerDIV);
layerDIV["myRootStorage"]                    = {};
layerDIV["myRootStorage"].newProperty0       = "something here";
layerDIV["myRootStorage"]["newProperty1"]    = "something here";
```

```
layerDIV["myRootStorage"][properyName]      = "something here";
```

Code listing 4.1.

Pay your attention on second example. We can use variable to get access to some property:

```
layerDIV["myRootStorage"][properyName]      = "something here";
```

Code listing 4.2.

Such technique can be very helpful in a lot of situations. So, using of object's body to storage some data is a very clear, understandable and effective way of developing. Via using common root-property we have decided second problem with potential conflicts, but first problem is a little harder.

4.1.2. Realization

[**Purity.Environment.Overhead**] is a Purity's module, which controls properties of objects. This module has several methods.

```
var Overhead = new Purity.initModule("Environment.Overhead");
Overhead.Properties.Set(targetObject, propertyName, propertyValue, rewriteFlag[false]);
Overhead.Properties.Get(targetObject, propertyName, deleteFlag[false]);
Overhead.Properties.Del(targetObject, propertyName);
```

Code listing 4.3.

Method [Properties.**Set**] create new property with name [propertyName] and value [propertyValue], and attach this property to object [targetObject]. If flag [rewriteFlag] will be in [true], method [Properties.**Set**] rewrites value, if property with same name was defined before (default value is [false]).

Method [Properties.**Get**] try read in object [targetObject] and return value of property with name [propertyName]. If flag [deleteFlag] in [true], method [Properties.**Get**] will delete property after read its value (default value is [false]).

And last method [Properties.**Del**] just remove property with name [propertyName] from object [targetObject].

As you can see developer get some useful functional. Firstly, developer doesn't care about common root-property – everything does [Purity.Environment.Overhead]. Secondly, developer can remove any property via name of property or remove it immediately after reading.

4.2. Events

4.2.1. DOM's events

Control of events one of problem place in HTML and DOM. And problem isn't attach some event to some element. Problem – detach such event after. And as you understand for hard one-page application detaching event is very important moment.

Besides, different browsers has different methods to manage DOM events. Take a look on code listing 4.4.

```
//Add / remove group #1
    element.addEventListener    (eventType, handle, false);
    element.removeEventListener (eventType, handle, false);
//Add / remove group #2
    element.attachEvent(("on" + eventType), handle);
    element.detachEvent(("on" + eventType), handle);
//Add / remove group #3
    element[("on" + eventType)] = handle;
    element[("on" + eventType)] = null;
```

Code listing 4.4.

- [**element**] is any DOM element, for example, node like <DIV> or <INPUT>.
- [**eventType**] is a name of event, like [click], [mouseDown] and etc.
- [**handle**] is a method, which will be called on event.

Pay your attention on group #3. It oldest way to attach and detach events and it works in any browsers and today. But here is problem: handle is saved as property of DOM's node. It means that we cannot attach to one node more than one handle (for one event).

Result: it's a **bad** way.

Methods from group #1 and #2 works similar. In this case, we can attach any count of handles for one event to one DOM's node – no problem. But, pay your attention on detach / remove methods. Here you cannot define some ID of event. But, how does browser detach necessary event? Browser looks at handle. So, to successfully detach / remove event developer should store link to all handle which was attached before. In this case, handle is ID, ID is handle. It seems nice idea. But, stop. Let's see one moment. JavaScript automatically clear memory from unused objects and functions and other data. For example, if we have some function JavaScript will remove this function from memory only when all links to this function will be deleted. But developer should store them to detach / remove events, remember? Yes, using such way developer has a risk of not optimally using of memory.

Result: it's a **bad** way.

And what is **good** way? Event's manager should:

1. Allow attach any count of handle to one element on same event;
2. Allow detach any handle by event ID;
3. Clear memory after event detach.

Whole such functional is released in [Purity.Environment.Events] library. This library controls events, attach it, detach it and clear memory. To get access to this module developer should define this module in [purityConfigurator] via property [modules] (read more: "2.1. Basic principals") and define link:

```
Events = new Purity.initModule("Environment.Events");
```

Code listing 4.5.

After this definition object [Events] gets access to next properties:

```
Events.DOM.Add          ({  element    : DOMObject,
                            type       : string,
                            handle     : function,
                            onetimerun : boolean,
                            id         : string});
Events.DOM.AddListener  (element: DOMObject, type: string, handle: function, id: string);
Events.DOM.Remove       ({  element    : DOMObject,
                            type       : string,
                            handle     : function,
                            id         : string});
Events.DOM.Call         (element: DOMObject, type: string);
Events.DOM.isAvaliable  (type: string, tagName: string, doEmulation: boolean);
Events.Window.Add       (type: string, handle: function, id: string);
Events.Window.Remove    (type: string, id: string);
Events.Window.IsSet     (id: string);
Events.CSS.Animation.attach.start       (element: DOMObject, handle: function, onceRun: bool);
Events.CSS.Animation.attach.iteration   (element: DOMObject, handle: function, onceRun: bool);
```

```
Events.CSS.Animation.attach.end          (element: DOMObject, handle: function, onceRun: bool);
Events.CSS.Animation.remove.start        (element: DOMObject);
Events.CSS.Animation.remove.iteration    (element: DOMObject);
Events.CSS.Animation.remove.end          (element: DOMObject);
Events.CSS.Transition.attach.start       (element: DOMObject, handle: function, onceRun: bool);
Events.CSS.Transition.remove.start       (element);
```

Code listing 4.6.

DOM group

Methods [Events.DOM.Add] and [Events.DOM.AddListener] are same, but [Events.DOM.Add] allows define property [onetimerun]. [onetimerun] in [true] means what handle will be automatically removed (detached) after handle will be run once.

IMPORTANT Don't use prefix "on" in [type]. Just define "click" instead "onClick".

To detach some handle developer should use method [Events.DOM.Remove]. Doesn't necessary define [id] and [handle]. Developer can define or [id], or [handle] and handle will be detached by [id] or [handle]. If developer defines and [id] and [handle], [Events.DOM.Remove] will remove handle by [id].

To call some event (manually calling) developer can use method [Events.DOM.Call] and define [element] and event type [type] (like, "click" or "keyPress").

Sometime developer have to check availability of some event. To solve this task, developer should use method [Events.DOM.isAvaliable]. Property [tagName] is string and looks like: "div", "scritp" and etc. If property [doEmulation] has value [true] event will be emulated. In this case, [Events.DOM.isAvaliable] create event [type], call this event and try to see – does it work or not. Emulation will be helpful in some situations when simple checking doesn't work. It depends of browser.

IMPORTANT Pay your attention, types [boolean] and [bool] are same. [bool] used only to make string of code shorter.

Windows group

To attach and detach events to global element [window] developer can use DOM group, but let's imaging situation. We have some <DIV> on page, what should change size according browser's window size. In this case we do:

```
Events.DOM.AddListener  (window, "resize", our_method_resize_our_div, "DIVEventResize_1");
```

Code listing 4.7.

Each time user changes browser's window size, will be called our handle [our_method_resize_our_div]. And our handle in this case do next:

```
function our_method_resize_our_div() {
    ourDIV.style.width  = window.innerWidth     + "px";
    ourDIV.style.height = window.innerHeight    + "px";
}
```

Code listing 4.8.

Everything is okay. But we are developing big and hard one-page application. And in this case our DIV can be removed from page and mount in page again. But what will be if our DIV is removed after handle was attached? Yes. Handle [our_method_resize_our_div] will be called (when user changes browser's window size) and our handle will call error because [ourDIV] === undefined. To solve such problem each time our DIV should

be removed, we should detach event's handle before. It isn't very comfortable for developer, isn't it? Yes. That's why better way – use windows group and attach events using [Events.Window.Add(...)].

In this case processor of [Events.Window] always checks returning from handle. If we modify our [our_method_resize_our_div] with next:

```
function our_method_resize_our_div() {
    if (typeof ourDIV !== "undefined") {
        ourDIV.style.width  = window.innerWidth     + "px";
        ourDIV.style.height = window.innerHeight    + "px";
        return true;
    }
}
```

Code listing 4.9.

[our_method_resize_our_div] will be detached automatically if [our_method_resize_our_div] doesn't return [true]. Here is only one case when it can be: when ourDIV is removed. So developer should return [true] and it means handle should "live" and return nothing, if handle should be removed.

CSS events

CSS3 has animations. It's very nice. But developer can meet a lot of situation when he have to know moment of end animation, for example. And such task can be huge problem, because CSS3 at this moment "fresh" tech and each browser has own methods of controlling necessary events.

But blocks [Events.CSS.Animation] and [Events.CSS.Transition] have cross-browsers methods, which allow attach and detach necessary events.

Here is only one property [onceRun] what needs some comments. [onceRun] in [true] means what handle will be automatically removed (detached) after handle will be run once, like [onetimerun] in [Events.DOM.Add].

IMPORTANT	[Events.CSS.Animation] and [Events.CSS.Transition] allow attach only one handle to one node (for same event).

Addendum A. Running PHP applications in Tomcat 6

Proof: http://php-java-bridge.sourceforge.net/doc/tomcat6.php

This guide shows how to install and run PHP applications like Moodle, mediaWiki, Joomla as Tomcat 6 web applications. And how to install PHP 5.x for all existing web applications.

Download and install Java, Tomcat and PHP

Use a graphical packet installer to install PHP. Or download JavaBridge.war, which contains an embedded PHP binary.

- Download and install Java 6 or above.
- Download and install the original tomcat servlet engine.

Install a PHP web application into Tomcat

1. Copy the PHP web application JavaBridgeTemplate.war or the demo JavaBridge.war to the Tomcat webapps directory.
2. Wait two seconds until Tomcat has loaded the web application.
3. Browse to http://127.0.0.1:8080/JavaBridgeTemplate621 and http://127.0.0.1:8080/JavaBridgeTemplate621/test.php to see the PHP info page.
4. Rename the file file:webapps/JavaBridgeTemplate621 directory, for example to file:webapps/Moodle, and download and install your PHP application to this directory.
5. Browse to the name of your web application, for example http://127.0.0.1:8080/Moodle, to run it from your internet browser.

PHP support for all existing Tomcat web applications

If you want to enable PHP for all of your web applications, move the Java libraries from the local web application folder to the Tomcat library folder and edit the Tomcat web configuration as follows:

1. Stop Tomcat.
2. Move the libraries JavaBridge.jar, php-servlet.jar and php-script.jar from thewebapps/JavaBridgeTemplate621/WEB-INF/lib directory over to the tomcat lib directory.
3. Edit the Tomcat conf/web.xml. Add the following lines:

```
  <listener>
    <listener-class>php.java.servlet.ContextLoaderListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>PhpJavaServlet</servlet-name>
    <servlet-class>php.java.servlet.PhpJavaServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>PhpCGIServlet</servlet-name>
    <servlet-class>php.java.servlet.fastcgi.FastCGIServlet</servlet-class>
    <init-param>
      <param-name>prefer_system_php_exec</param-name>
      <param-value>On</param-value>
    </init-param>
    <init-param>
      <param-name>php_include_java</param-name>
      <param-value>Off</param-value>
    </init-param>
  </servlet>
```

```
  <servlet-mapping>
    <servlet-name>PhpJavaServlet</servlet-name>
    <url-pattern>*.phpjavabridge</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>PhpCGIServlet</servlet-name>
    <url-pattern>*.php</url-pattern>
  </servlet-mapping>
```

Code listing A.1

IMPORTANT Lines from code listing A.C.3 should be added into block <web-app></web-app>, via next rules:

- <listener> should be added after all other <listener> inside <web-app></web-app>,
- <servlet> should be added after all other < servlet > inside <web-app></web-app>,
- and etc.

4. Start Tomcat again. Now you can add PHP scripts to tomcat.
5. Add a PHP test file, like next (or any other):

```
<?php
    include_once("java/Java.inc");
    echo phpInfo();
?>
```

Code listing A.2

6. Check the process list using the Unix/Linux or Windows task manager. There should be 5 php-cgi FastCGI executables waiting for requests from Java.

Please see our FAQ for more information how to set up a load balancer or how to create a distributable PHP/Java web application.