Developer Manual

Copyright

This document is provided for information purpose only. NHN Corp. has endeavored to verify the completeness and accuracy of information contained in this document, but it does not take the responsibility for possible errors or omissions in this document. Therefore, the responsibility for the usage of this document or the results of the usage falls entirely upon the user, and NHN Corp. does not make any explicit or implicit guarantee regarding this.
Software products or merchandises mentioned in this document, including relevant URL information, conform to the copyright laws of their respective owners. The user is solely responsible for any results occurred by not complying with applicable laws. NHN Corp. may modify the details of this document without prior notice.

Important Information regarding Open Source Licenses

There are several types of open source licenses. XE is licensed under the GNU Lesser General Public License (LGPL) v2. As there is a slight difference between LGPL v2 and LGPL v3, it is important to keep the version in mind. LGPL is basically the same as GPL, but its scope of application is more restrictive. Like GPL, LGPL has the effect of forcing all software that includes any LGPL-licensed software to have the same license. While GPL requires all software that includes any GPL-licensed software to unconditionally open its source codes, LGPL-licensed programs do not need to open their source code when they are used under specific conditions. Therefore, LGPL-licensed software can also be used for developing proprietary software. For more information, see the website below.
LGPL License: http://www.gnu.org/copyleft/lesser.html
GPL License: http://www.gnu.org/licenses/gpl.html

Introduction to Document

Document Overview

This document describes how to develop XE additional features including modules, addons and widgets. The information in this document is based on XE core 1.5.x.

Audience

This intended audience of this document is users who want to develop XE additional features. This document does not cover the basic knowledge of using web servers and PHP, which you need to know to better understand how to develop XE additional features. Please refer to the related books and documents, if necessary.

Contact

For any comments or inquiries regarding the document, contact via email below.

Email: developers@xpressengine.com

Revision History

Ver.	Date	Changes Mode
1.1	Dec. 30, 2011	Updated based on XE core 1.5
1.0	Jul. 29, 2011	1.0 version distributed.

Conventions

Note Symbol

> **Note**
>
> A note provides information that is useful for readers.

Caution Symbol

> **Caution**
>
> A caution provides information that you should know in order to prevent system damages.

Window, Site, Menu, and Field Names/Selected Value and Symbol

Window, site, menu and field names, a selected value, and a symbol are marked as follows:

- Window name: In bold type such as **window name** window. Note that this convention is not applied in source code.
- Site name: Enclosed in single quotations such as 'Naver Desktop Download' site.
- Menu name: In bold type such as **Menu > Submenu**.
- Input value: In italic type such as *homepage*.

Source Code

Source code is written in black on a gray background in this document.

```
COPYDATASTRUCT st;
st.dwData = PURPLE OUTBOUND ENDING;
st.cbData = sizeof(pp);
st.lpData = &pp;
::SendMes(GetTargetHwnd(), WM_COPYDATA, (WPARAM)this->m_hWnd, (LPARAM)&st);
```

Table of Contents

List of Tables and Figures

List of Tables

List of Figures

1. Understanding XE core

This chapter describes what XE core is, and XE's request lifecycle and directory structure.

1.1 Overview

XE core represents a framework on top of which developers can build customized applications.

Among the features that come with the core are member management, article and comment management as well as other features to ease development such as a powerful database abstraction layer. Also, XE was built using the MVC (Model-View-Controller) architecture for a clean separation of concerns.

All the incoming requests to an XE application are being handled by index.php. This page is responsible for initializing the request context, identifying the appropriate module and sending back a response to the client (browser).

In XE, almost everything is a module. Most core functionality of XE (documents, pages, modules, addons, etc.) resides in modules that get initialized by XE's core classes.

XE knows what module to be used when a request comes based on two parameters: a module name and an action name (if these are not given, it fall backs to defaults). For example, to display an admin page, the URL would be <root_url>?module=admin&act=dispBoardAdminContent.

In this chapter, you can understand XE's request lifecycle and directory structure which you need to know in order to extend XE.

1.2 Request lifecycle overview

The request lifecycle represents the sequence of steps that XE goes through from the moment when a URL is accessed to the moment when a response is sent back to the client.

You can see an overview of the request life-cycle in the figure below:

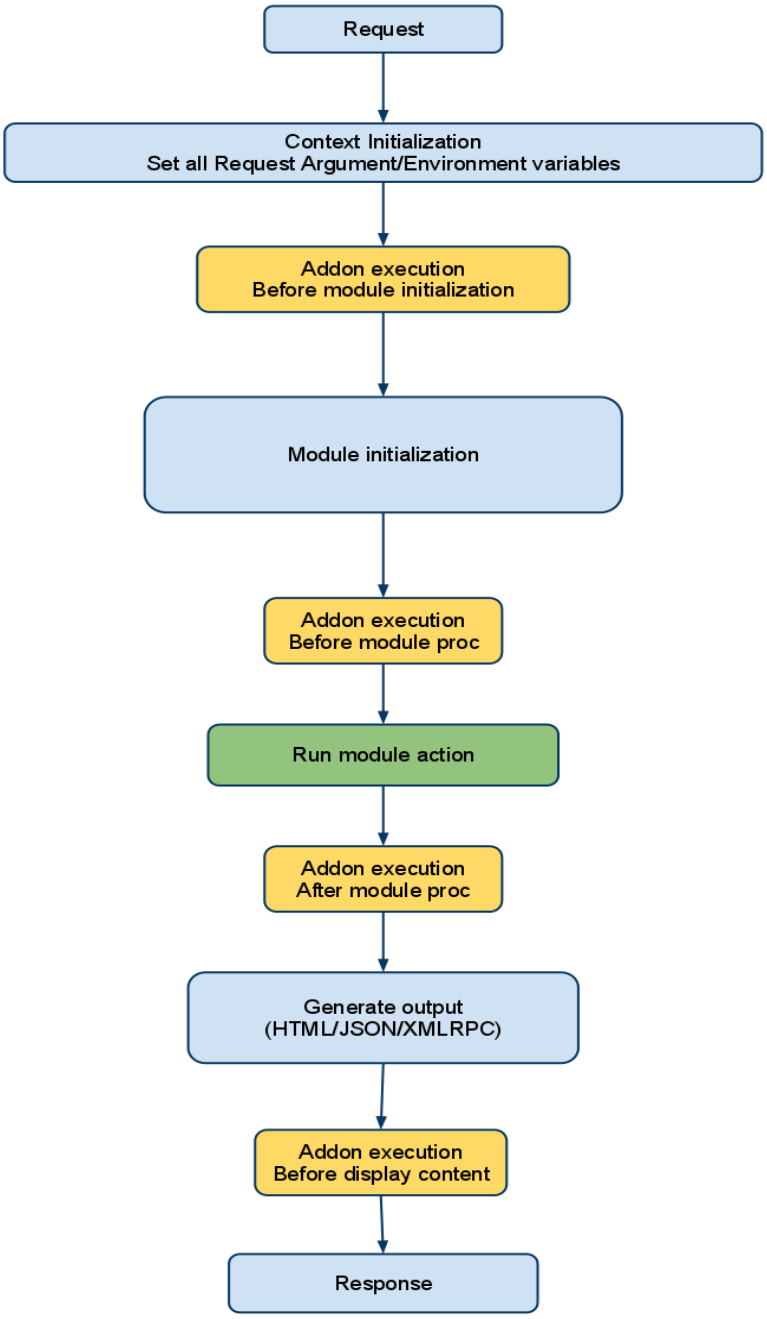

Figure 1-1 XE request lifecycle

As you can see, the main steps are:

1. Context initialization
2. Module initialization
3. Running the requested module action
4. Generating response output

Developers can run custom code at certain moments of this lifecycle through the use of addons. Addons are a type of XE additional features that work through the PHP include mechanism - code is directly included in the Core methods, thus giving developers full power to override the context in which requests are handled. For more information about addons and how to create an addon, see "2.2 Addons".

1.2.1 Context Initialization

Context initialization is handled by the Context class. This class encapsulates the environment in which XE actions are run. Among its responsibilities are:

- setting context variables in $GLOBALS (to use in display handler)
- including language files according to language types
- setting authentication information in Context and session
- checking if server uses rewrite module
- setting locations for JavaScript use

Context class location: .\classes\context\Context.class.php

1.2.2 Module Initialization

Module initialization is handled by the init() method of the ModuleHandler class. This method is responsible for:

- executing addons before module initialization (before_module_init hook).
- setting variables from request arguments.
- validating variables to prevent XSS.
- finding the requested module based on module_srl, mid and/or document_srl.
- setting current module info into context.

ModuleHandler class location:.\classes\module\ModuleHandler.class.php

1.2.3 Running the requested module action

All modules are run through the procModule() method of the ModuleHandler class.

- This module executes the addons hooked before module execution (before_module_proc hook).
- This module executes current module action.

1.2.4 Generating response output

DisplayHandler class is responsible for generating output. Depending on the request type, it can display either HTML or XML/JSON content. In the case of HTML, this class first retrieves the appropriate template file and populates it with response values (properties of the ModuleObject). For XML/JSON the ModuleObject properties are simply serialized as XML/JSON, without any other formatting.

1.3 Directory structure

Files and folders below are created after the installation in the root of XE.

Table 1-1 XE directory structure

Folders/Files	Description
addons	Includes all XE addons.
classes	Includes XE core classes.
common	Contains static files and templates common to all XE modules. This is also where the global language files reside.
config	Contains default configuration and common function used.
files	Is created during installation, and saves uploaded files, internal cache files and DB & environment configuration files.
layouts	Contains all default and custom XE layouts.
libs	Includes all libraries used by XE core (e.g. ftp, tar).
m.layouts	Includes mobile layouts.
modules	Includes all modules (XE core and custom modules).
themes	Contains themes (layouts and skins for modules).
widgets	Contains all XE widgets.
widgetstyles	Contains all widgetstyles required to decorate the widgets.
index.php	Contains functions as a gateway for all inputs and outputs in XE.
.htaccess	Contains configuration information to use the rewrite mod of Apache Web Server.
LICENSE	Contains the original license of XE.

1.3.1 addons Directory

The addon can be configured simply as active or inactive, and when there is the need for additional configurations, it can interact with the modules.

Table 1-2 addons directory structure

Folders/Files	Description
addons	A root folder of addon
addon_name	A folder whose name is the same as the addon's name
conf	Contains configuration information for an addon.
info.xml	Contains description, creator, version and creation date of an addon.
addon_name.addon.php	Contains an addon's execution code which will be inserted when the addon is executed.
queries	Contains a collection of queries to be used for an addon.
queryID.xml	A query file. The query schema is the same as the one used in the module.

For more information, see "2.2 Addons".

1.3.2 classes Directory

This directory contains the library classes commonly used by each component, such as module, addon, widget, and others in XE.

The default distribution version has the following classes:

Table 1-3 classes directory structure

Folders	Description
cache	Contains all cache classes that XE core can use (CacheAPC, CacheMemcache, and CacheHandler). The default class is CacheHandler.
context	Contains Context class that manages Context such as request arguments/environment variables.
db	Contains all Databases supported by XE core: CUBRID, MySQL, Firebird, MySQL Innodb, MySQLi, PostgreSQL, SQLite2, and SQLite3 with PDO.
display	Contains classes which are responsible for displaying the execution result (depending on the request type: HTML, JSON or XMLRPC).
editor	Contains an editor handler class.
extravar	Contains a class to handle extra variables used in posts, member and others.
file	Contains classes which handle the common use with files and folders.
handler	Contains an abstract class of (*)Handler.
httprequest	Contains a class that is designed to be used for sending out HTTP request to an external server and retrieving response.
mail	Contains a class for mailing.
mobile	Contains a class for mobile optimization.
module	Contains module handler and module object classes.
object	Contains a base class that is designed to pass the Object instance between XE modules.
page	Contains a base class that handles page navigation.
template	Contains a class that compiles template file by using regular expression into PHP code, and XE caches compiled code for further uses.
widget	Contains a handler class for widget execution.
xml	Contains classes to parse and generate XMLs.

1.3.3 common Directory

This directory contains the resources essential for XE.

Table 1-4 common directory structure

Folders/Files	Description
common	Contains common JS and CSS files used in XE.
css	Contains common CSS files used in XE.
default.css	Defines basic styles and XE-specific styles.
button.css	Defines basic button styles being used in XE.
js	Contains common JS files used in XE.

Folders/Files	Description
common.js	Defines various types of JavaScript functions being used in XE.
jquery.js	A jQuery (http://jquery.com) file, which is the JavaScript framework being used in XE.
js_app.js	A JAF file, which is a JavaScript application framework being used in XE.
x.js	A JavaScript library file for cross-browsing. It is not recommended to use this file, as it will be removed in the future.
xml_js_filter.js	An XML JS filter file being used in XE.
lang	Contains the language files supported by XE.
tpl	Contains common layout and template files used in XE.
common_layout.html	A common layout being used in XE.
default_layout.html	An empty layout that only displays contents when there is no layout skin in use.
mobile_layout.html	A layout used in the XE mobile environment.
popup_layout.html	A layout used when opening a popup window in XE.
redirect.html	A template file used when there is the need to redirect to another page.
refresh.html	A template file used to refresh.

1.3.4 config Directory

This directory contains the files with the default configuration and the collection of functions that are frequently used.

Table 1-5 config directory structure

Files	Description
config.inc.php	Stores XE version and debug configuration for developers.
config.user.inc.php	Stores debug configuration, which you need to create manually.
func.inc.php	Contains the functions that are frequently used in XE.

1.3.5 files Directory

The file directory is automatically created by XE during installation. It contains cache files, uploaded files and other files needed by the modules.

Table 1-6 files directory structure

Folders/Files	Description
_debug_message.php	Displays PHP error messages, database errors and others based on the value set in ./config/config.inc.php for the __DEBUG_OUTPUT__ option. This file is deactivated by default.
attach	Is used for attachments (uploaded files).

Folders/Files	Description
binaries	Stores attached files with extension names other than gif, jpg, jpeg, png, swf and mpeg (a target having directly spread the contents to fpassthru() to avoid malicious attacks).
images	Stores image and video files that can be directly accessed from the browser. The subfolder naming convention is ./$module_srl/$document_srl/$file_name.
cache	A cache folder
addon	Contains cache files related to an addon.
mobileactivated_addons.cache.php	Contains the PHP code to execute activated addons. (Mobile environment)
pcactivated_addons.cache.php	Contains the PHP code to execute the activated addons. (PC environment)
document_category	Contains XML and PHP cache files for the document category.
editor	Contains information cache files of the editor component.
js_filter_compiled	Contains cache files of XE's XML JS filter.
lang_defined	Contains cache files of user-defined language code.
layout	Contains layout cache files of XE. The modified layout contents will be stored by editing layouts.
menu	Contains XML and PHP cache files for the menu information created by the menu module of XE.
module_info	Stores information cache files for each of XE modules.
opage	Contains cache files for the external page module of XE.
optimized	Contains optimized cache files to reduce traffic and increase page loading speed by integrating CSS and JS files.
page	Contains cache files for page module of XE.
queries	Contains cache files for XML Query compile of XE.
template_compiled	Contains template cache files of XE.
thumbnails	Contains document thumbnail images of XE.
widget	Contains cache files for widget information of XE.
widget_cache	Contains cache files to store and utilize the information of the created widgets. When the caching time is specified in the widget, the cache file will be stored.
triggers	Contains cache files for the trigger function of XE.
widgetstyles	Contains cache files to store and utilize the information of the widgetstyles.
newest_news.language.cache.php	Contains temporarily stored files of the latest news on the administrator page.
config	Contains configuration information of site administrators such as DB and FTP.
db.config.php	Contains DB configuration information.

Folders/Files	Description
ftp.config.php	Contains FTP information saving files of the server where XE is installed.
lang_selected.info	Saves a language list for a certain site that the administrator wants to work on.
member_extra_info	Contains files used for extra variables of member information.
image_mark	Contains image files for the marks in front of the member's name.
image_name	Contains name files for the member's images.
profile_image	Contains profile image files registered by members.
signature	Contains the signatures of members are stored.
point	Contains point scores for each member are stored.
new_message_flags	Contains the location of temporary files whether used to store new messages from certain members or not.
agreement.txt	Stores the terms and the conditions configured by the member management module.
ruleset	Contains dynamic ruleset files.
theme	Stores the current theme information.

1.3.6 layouts Directory

The layout is the shell surrounding content (module). The layout can be used by itself or through interaction with a menu specified by the layout author. You can edit a layout template file by using **Edit Layout** in the **Layout Management** menu.

Table 1-7 layouts directory structure

Folders/Files	Description
Layout Name	Layout root directory
conf	Contains the configuration file with layout information.
info.xml	Defines layout author, description, additional variables and the number & name of the interworking menu.
layout.html	Defines the layout template.

1.3.7 modules Directory

Modules directory explanation and directory rules can be found on module directory file structure.

Table 1-8 modules directory structure

Folders/Files	Description
module_name	Module root directory named after the module.
conf	Includes module description, action setup and permission setup.
info.xml	Contains creator information and description of the module.
module.xml	Contains action module definitions, including information related to the behavior of the module.
lang	Contains language pack files.
en.lang.php	Contains English language pack.
schemas	Contains database table schema used for module installation, optional folder used only when the current module uses a new database.
table.xml	Contains table schema (creating files by using table names).
queries	Contains XML syntax files for defining queries used for inserts, selects and updates.
ruleset	Contains ruleset XML files to be used by modules.
tpl	Contains template files that are used for the administrator view of the module.
css	Contains style sheets.
images	Stores template images.
js	Stores template JavaScript files.
filter	Declares nodes and parameters from forms that will be passed to processing files (will be covered in next paragraphs).
template_files.html	Contains skins (including admin screen of a module) created by using XE template syntax of a screen with no skin.
skins	Contains skin files that are displayed on the front-end of the module.
Skin Name	Skin name
css	Contains style sheets.
images	Stores skin images.
js	Stores skin JavaScript files.
skin.xml	Contains skin creator information and extra variables declaration for skin.
template_files.html	A skin file created by using XE template syntax.

Folders/Files	Description
module_name.class.php	Contains base class of the module which contains the installation, update and deletion functions.
module_name.view.php	Contains view functions that display the front-end part of the module.
module_name.model.php	Defines module model class and functions.
module_name.controller.php	A controller for user interface.
module_name.admin.view.php	Contains view class and functions used to display the back-end part of the module.
module_name.admin.model.php	Declares model class and functions for admin.
module_name.admin.controller.php	Contains controller actions for administration functions.
module_name.api.php	Similar to the view - it prepares data for display; more precisely, it usually removes internal data from the output, can return JSON or XML, used for instance to create other types of apps, not only web. e.g. iPhone app.
module_name.wap.php	Contains classes for WAP mobile phones as they have different outputs.
module_name.smartphone.php	Contains special classes for smart phones, including iPhones.

For more information, see "2.1 Modules".

1.3.8 themes Directory

Themes are used to manage layouts and module skins, for unified website design.

Table 1-9 themes directory structure

Folders/Files	Description
Theme Name	Themes root directory
conf	Contains the configuration file which has themes information.
info.xml	Contains creator information and description of the theme, and definition of skins included in the theme.
layouts	Layout skin root directory
Layout Name	Layout name
conf	Contains the configuration file which has layout information.
info.xml	Contains creator information and description of the layout, extended variables, and the number of connected menus and their names.
layout.html	Layout template file
modules	Root directory of a set of module skins
Module Name	Name of the module to which the skin is applied.
css	Style sheet
images	Contains skin images.

Folders/Files	Description
js	Contains skin JS files.
skin.xml	Contains skin creator information and declaration of extended variables of skins.
template_files.html	Skin file which is created with XE template syntax.

1.3.9 widget Directory

A widget is a small program displayed on the screen. Some widgets interact with recent posts or member information (login form), while other widgets communicate with external open APIs.

The widget folder should be named the same as the widget itself. Widget folder structure is as follows:

Table 1-10 widget directory structure

Folders/Files	Description
Widget Name	Widget root folder
widget_name.class.php	The widget's class file - used for processing data and specifying template files
conf	Configuration folder
info.xml	Defines widget information (name, descriptions) and the variables available to the widget class
skins	Skin folder
Skin Name	Contains the files for a widget skin; the folder should have the same name as the skin itself
skin.xml	A configuration file for the information on the names, description, authors and colorsets of skins.

1.3.10 widgetstyle Directory

This directory contains widgetstyles. Widgetstyles are used to decorate the widget container and allows the user to change widget appearance such as background, borders and title of a widget.

Widgetstyles need to use the following folder structure:

Table 1-11 widgetstyle directory structure

Folders/Files	Description
widgetstyles	Widgetstyles root folder
Widgetstyle names	Name of the widget style
widgetstyle.html	A template file for the widgetstyle.
skin.xml	A configuration file for titles, descriptions, authors and additional variables of the widgetstyle.
preview.gif	Widgetstyle preview

2. Extending XE

This chapter describes how to develop XE additional features including modules, addons and widgets.

2.1 Modules

XE is a Contents Management System (CMS) that can be upgraded using different types of extensions. The most important extension is the Module, which is a collection of files that add new functionality to the platform.

There are three rules to follow in order to create a minimal working module:

- Module must be a folder under 'modules' directory. Folder name is the same with module name. Think about a unique name if you want to publish your module, as it may conflict with other modules named by other developers.
- The info.xml file contains generic information about the author, functionality description and optional extra variables if the module requires so.
- Module.xml contains configuration parameters, actions definition etc. They are covered below.

2.1.1 How to create config/info.xml

First, let's give an example on what the file should look like.

```
<?xml version="1.0" encoding="UTF-8"?>
<module version="0.2">
   <title xml:lang="en">Module name</title>
   <description xml:lang="en">Module description </description>
   <version>1</version>
   <date>2011-05-01</date>
   <category>service</category>
   <author email address="author@authorland.com" link="http://www.authoria.com/">
   <name xml:lang="en">Author name</name>
   </author>
</module>
```

The <category> tag refers to the module classification in the admin menu. The options that can be entered are: service | member | content | statistics | construction | utility | interlock | accessory | migration | system | package, you can enter.

- service: Services Management
- member: Membership Management
- content: Information Management
- statistics: Statistics View
- construction: Construction Set
- utility: Features Settings
- interlock: Interlocking Set
- accessory: Addons Set
- migration: Data Management / Restoration
- system: System Administration / Settings
- package: Package module such as cafeXE and textyle

2.1.2 Creating actions

In Xpress Engine all inputs and outputs are processed through index.php. The action request argument is determined by Module Handler and usually the $act variable is used. The actions of the module are declared in the conf/module.xml file. For a better understanding let's give an example:

```
<?xml version="1.0" encoding="utf-8"?>
<module>
   <grants>
      <grant name="post" default="guest">
```

```
        <title xml:lang="en">Post</title>
  </grants>
  <permissions>
     <permission action="dispForumAdminInsertForum" target="manager" />
     <permission action="dispForumAdminForumInfo" target="manager" />

     <permission action="procForumAdminInsertForum" target="manager" />
     <permission action="procForumAdminInsertListConfig" target="manager" />
  </permissions>
  <actions>
     <action name="dispForumIndex" type="view"   />
     <action name="dispForumContent" type="view" index="true"/>
     <action name="dispForumNoticeList" type="view" />
     <action name="dispForumContentList" type="view" />
     <action name="dispForumContentView" type="view" />
     <action name="dispForumCatogoryList" type="view" />
     <action name="dispForumContentCommentList" type="view" />
     <action name="dispForumContentFileList" type="view" />

     <action name="procForumInsertDocument" type="controller" />
     <action name="procForumDeleteDocument" type="controller" />

     <action name="dispForumAdminContent" type="view" standalone="true"
admin index="true" menu name="forum" menu index="true" />
     <action name="dispForumAdminForumInfo" type="view" standalone="true"
menu_name="forum" />
     <action name="dispForumAdminExtraVars" type="view" standalone="true"
menu name="forum" />
     <action name="dispForumAdminForumAdditionSetup" type="view" standalone="true"
menu_name="forum" />
     <action name="procForumAdminDeleteForum" type="controller" standalone="true"
menu_name="forum" ruleset="deleteForum" />
     <action name="procForumAdminInsertListConfig" type="controller" standalone="true"
menu name="forum" ruleset="insertListConfig" />

     <action name="dispForumCategory" type="mobile" />
     <action name="getForumCommentPage" type="mobile" />
  </actions>
  <menus>
     <menu name="forum">
        <title xml:lang="en">Forum</title>
        <title xml:lang="ko">포럼</title>
     </menu>
  </menus>
</module>

<action>
```

The properties used in conf/modules.xml are described in the following table.

Table 2-1 Properties used in actions

Properties	Description
name	The name of the action that also contains the name of the module. If the action name contains the string “Admin” then it should have administrative rights.
type	Defines what the type of the action is, and so in what file should be located: view, model or controller. If the name contains the string “Admin” then it should be located in the admin view, model or controller PHP files.
standalone	The current action is not dependent on the rest of modules if standalone is set to "true". If standalone is set to "false" and the request is not received, the module will output an error when running. This property will be deprecated.
index	Should only be applied to only one action and sets the default action of the

Properties	Description
	module.
admin_index	Is only applied to one action and represents the default action of the module back-end.
setup_index	Module Settings page is used. <Permissions ... /> - action permissions
menu_name	Name of the menu to which the action belongs.
menu_index	If this property is set as "true," it means that this action is the initial action of the current menu.
ruleset	Name of the ruleset to be applied to the action.
action	Name of the action for which the permission is being declared.
target	The permissions supported are: • member: Member • manager: Manager

2.1.3 Using action forward

In general, the actions are owned by XE modules. However there are some cases when an action is used in various modules. This method is called action forward.

The most typical case is the RSS module. The RSS action is not the action defined by the board module, but is called and executed by the Action Forward feature.

```
?mid=board&act=rss
```

Action Forward can be used to process the module with an independent feature.

In case of the request above, XE looks for a mid called “board”, and in case this mid does not contain the rss action, XE searches for rss registered through Action Forward table in the database. Since rss action is registered in DB as the view type of rss modules, XE configures all mid information for board, and then creates the view object of rss modules to execute the rss method.

This Action Forward will be needed when XE wants another method while maintaining the layout or the information of the currently requested module. As another example, the action to see a friend list works using dispCommunicationFriend action of Communication module. This action replaces the contents with the friend list while maintaining the layout of the current module.

In other words, the display of the content area can be changed by the appointed action, and different results can be induced by the information of the requested module.

Registration of Action Forward

In general, the Action Forward is stored when processing moduleInstall() in the module.class.php. It can be registered as follows:

```
$oModuleController = &getController('module');
$oModuleController->insertActionForward('module', 'type(Ex:controller)', 'action_name');
```

Verification of Action Forward

You can confirm the registration of the Action Forward by using the code below. In general, it is used in the checkUpdate() method of the module.class.php.

```
$oModuleModel = &getModel('module');
if(!$oModuleModel->getActionForward('action_name')) ...
```

Deletion of Action Forward

You can delete the Action Forward when it has no use.

```
$oModuleModel = &getModel('module');
$oModuleModel = &getController('module');
if($oModuleModel->getActionForward('Action Name'))
   $oModuleController->deleteActionForward('Module Name','Type','Action Name');
```

Otherwise, if an action name does not consist of (disp|proc|get) module name, you will have to register the Action Forward.

2.1.4 Using triggers

A trigger is used when a module has to provide actions that are already implemented in other modules. For example in the forum module we want to use a view for the admin that is already found in triggerDisplayDocumentAdditionSetup of the document module.

For inserting a trigger in the DB:

```
$oModuleController->insertTrigger('forum.dispForumCommentSetup', 'comment', 'view',
'triggerDispCommentAdditionSetup', 'before');
```

For getting a trigger:

```
if(!$oModuleModel->getTrigger('forum.dispForumAdditionSetup', 'document', 'view',
'triggerDispDocumentAdditionSetup', 'before')) return true;
```

For calling a trigger:

```
ModuleHandler:: triggerCall ('Trigger Name', 'call time (Called Position)', the trigger
will be used as a parameter of the object);
```

For deleting a trigger:

```
$ OModuleController-> deleteTrigger ('Trigger Name', 'module name', 'call the method
belongs to the type of instance', 'call the method (Called Method)' + ',' call time
(Called Position) ');
```

2.1.5 Using ruleset

Ruleset is used for both client and server sides to check the validity of information when the information of HTML form is delivered to the processing method of PHP. It is contained in the XML files in the ruleset directory of each module directory. You can use ruleset as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<ruleset version="1.5.0">
   <customrules>
   </customrules>
   <fields>
      <field name="user_id" required="true" length="3:20" />
      <field name="user_name" required="true" length="2:40" />
      <field name="nick_name" required="true" length="2:40" />
      <field name="email_address" required="true" length="1:200" rule="email" />
   </fields>
</ruleset>
```

The elements and properties used in ruleset are described in the following table.

Table 2-2 Elements and properties used in ruleset

Elements	Properties	Description
customrules		Defines custom rules.
rule		Custom rule

Elements	Properties	Description
	name	Name of custom rule
	type	Type of custom rule. You can use one of "regex", "enum" and "expression". • "regex": when creating regular expression • "enum": when selecting only one from given values • "expression": when an expression is required.
	test	Test code for custom rule
fields		A set of fields to check the validity.
field		A field to validate.
	name	Name of a form element
	rule	Rule to apply
	required="true"	Means that this field is mandatory.
	length	Length of field. You can set the value as "minimum:maximum."
	default	Default value
	equalto	Means that the value entered for 'equalto' should be the same as the one for the current element (like password and confirmation of password).
	modifier	Used to change the value entered before using the rule or change the result after validation.

For more information, refer to "4 Working with Forms."

2.1.6 Using form filters

The filters are used in XE to pass information from an HTML form to a processing method in PHP and for specifying JavaScript callback functions. The filters are contained in XML files inside the tpl folder. For XE 1.5 or higher version, it is recommended to use the ruleset rather than form filters.

To study better the syntax of filters let's have a look below:

```
<filter name="insert_contest" module="contest" act="procContestAdminInsertContest"
confirm msg code="confirm submit">
  <form>
     <node target="mid" required="true" maxlength="40" filter="alpha_number" />
     <node target="browser_title" required="true" maxlength="250" />
  </form>
  <parameter>
     <param name="contest name" target="mid" />
     <param name="module_srl" target="module_srl" />
     <param name="module_category_srl" target="module_category_srl" />
     <param name="layout_srl" target="layout_srl" />
     <param name="skin" target="skin" />
     <param name="browser title" target="browser title" />
     <param name="header text" target="header text" />
     <param name="footer_text" target="footer_text" />
  </parameter>
  <response callback_func="completeInsertContest">
     <tag name="error" />
     <tag name="message" />
     <tag name="module" />
     <tag name="act" />
     <tag name="page" />
     <tag name="module_srl" />
```

```
   </response>
</filter>
```

The following table shows the elements and attributes used in the form filter.

Table 2-3 Attributes used in form filters

Elements	Attributes	Description
form		Verifies the input value.
node		Verifies an HTML form.
	required	Checks if the current input element is a required value. If it is set as 'true' and a value is not entered for the element, an alert is generated.
	filter = "filter type"	The types you can use for the filter are email (email_address), userid (user_id), url (homepage), korean, korean_number, alpha, number, and alpha_number.
	equalto = "target person"	Means that the value entered for 'equalto' should be the same as the one for the current element (like password and confirmation of password).
	maxlength	Maximum length
	minlength	Minimum length
parameter		Changes the name of a form element to be sent to the server or sends the value entered for the parameter only to the server. If you don't use this element, all the form elements are sent to the server by default.
param		Writes information for the form element which you want to re-define or send to the server.
	name	Form element name
	target	Element name to re-define
response		
	callback_func	JavaScript callback function. This attribute must be actually implemented.
tag		Defines a variable to be passed to the callback function.
	name	A name of the variable to be added to the arguments of the callback function. The variables are used to execute the action in the controller and implement the values to be passed to the callback function with $this->add('variable name', 'value')

For more information, see "4 Working with Forms".

2.1.7 Defining database queries

XE uses a custom query language in order to define queries. The XML code is parsed by the XmlQueryParser.class.php that resides in the folder ./classes/xml. For example:

```
<query id="getCounterStatus" action="select">
<tables>
<table name="counter_status" />
</tables>
<columns>
<column name="sum(unique visitor)" alias="unique visitor" />
<column name="sum(pageview)" alias="pageview" />
</columns>
```

```
<conditions>
<condition operation="more" column="regdate" var="start date" notnull="notnull"pipe="and"
/>
<condition operation="less" column="regdate" var="end_date" notnull="notnull"pipe="and"
/>
</conditions>
</query>
```

2.2 Addons

In Xpress Engine, an addon performs hooking, which is an action that grabs other normal actions.

Hooking uses the 'include,' which is available in interpreter-based languages such as PHP. The exclusion of addons in the form of a function or a class in XE was intentional, to allow them to be inserted to the normal Context of XE as native code. For this reason, the addons of XE can be used to powerful effect from the moment they are called. However, they must be created with caution so as not to overload the overall operation of XE.

The following are the minimum rules that must observed when creating an addon:

- Location: addons/addon_name
- Addon operation file: Addon_name.addon.php
- The info.xml file in which the creator information, description of the addon and the addon variable from an administrator (when necessary) are to be stored.

2.2.1 When to Call Addons

The four points of time at which an addon can be called are as follows:

- before_module_init - before creating a module object: After finding a necessary module upon a user request and before creating the object of that module.
- before_module_proc – before executing a module: After initializing the object of a module and before executing the module.
- after_module_proc – after executing a module: Immediately after executing a created module object and obtaining the result.
- before_display_content – before displaying result: Immediately before displaying the result of a module to which a layout has been applied.

To better understand what these hooks really mean, and why some addons use only specified moments in the Xpress Engine control path let's give some practical examples.

Tag list - After module proc

Let's take an addon that will display the list of tags of all documents on a page. For the tag list to be generated we need to first get the documents which have the module_srl of the current page. In order to do this we need to get the module_srl. For this to be possible we have to choose as called position the after_module_proc . After the module information is being processed all of the operations defined previously are viable.

Meta Tag - Before module proc

This addon will have the role to insert in every page meta tags, including meta description, meta keywords, meta author, etc. The addon will use the before_module_proc position as a hook because it needs to insert the meta tags before the content is generated in the module processing operation.

Point Level Icon- before display content

We need an addon that will display an icon for each user depending on the point level that the specified member accumulated. This addon will use the before_display_content position as a hook because it has to replace some HTML code inside the content depending on some parameters that were already processed.

Counter- Before Module Init

This addon was developed to make a statistic of visits on a website built with Xpress Engine.

It uses of course the counter module. The counter addon just uses the information in the $is_logged variable to count the number and visits. In order to do this the module uses before_module_init, the first chronological hook, because it doesn't need any more information from the module processing operation.

2.2.2 Variables to Be Passed When Calling an Addon

The following common variables can be transferred to an addon at the four calling points.

- $called_position: Contains the information of the time of calling. It can have one of the following four values: before_module_init, before_module_proc, after_module_proc, or before_display_content.
- $addon_path: Contains the path of the called addon.
- $addon_info: The addons of XE can be configured independently, and they can specify a target module in which they will be operated. The $addon_info variable contains the information of extra_vars in info.xml declared by an addon, and such information defers depending on each addon.

2.2.3 Creating an addon

The 'addons' directory may include files with different names. Classes can be used in the directory. However, the declaration of a function is not allowed because it uses the include structure to be operated as native code.

config/info.xml

To create an info.xml file, use the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<addon version="0.2">
 <title xml:lang="en">Addon title</title>
 <description xml:lang="en">Addon description</description>
 <version>Addon version</version>
 <date>Year-Month-Date</date>
 <author email_address="The email address of an author" link="The homepage address of an
author">
 <name xml:lang="en">Author name</name>
 </author>
 <extra vars>
 <var name="Variable name" type="textarea">
         <title xml:lang="en">Variable name (for output)</title>
         <description xml:lang="en">Variable description</description>
 </var>
 </extra vars>
</addon>
```

Create extra_vars if necessary. Omit the details if there aren't any, by using the "<extra_vars/>" command. Save the file above as info.xml, and move it to the conf/info.xml directory.

addon_name.addon.php

Create an addon file in PHP if it is intended to perform any actions. However, functions cannot be declared, as an addon is usually called within the method of a class object. Note that you can define and utilize classes in an addon.

The beginning section of any addon file should look like the following:

```
  <?php

 /**
```

```
        * @file addon name.addon.php
        * @author author name (email address)
        * @brief description
        **/
        if(!defined('__ZBXE__')) exit();
```

Double-check the __ZBXE__ constant so that it will not be executed by an external request; it must be called by XE. That is, XE additional features check if the __ZBXE__ constant is set as 'true' before executed. The actions of an addon can be controlled by called_position; this must be done manually in the addon.

Let's take for example an addon that displays the tag list of all documents at the bottom of the page. Firstly we have to establish what the proper hook for us to call the addon. In order for us to get the list of documents, the page module has to be processed so we are going to use 'after_module_proc' like in the following code:

```
<?php
if(!defined("__ZBXE__")) exit();
/**
* @file tag_list.addon.php
* @author Author (author@authorland.com)
* @brief Description of the addon
**/

if($called_position != 'after_module_proc' || Context::getResponseMethod()!=='HTML')
return;

$obj->module srl=Context::get('module srl');
$document list=executeQueryArray('addons.tag list.getModuleDocumentTags',$obj);
$tags='';
foreach ($document_list->data as $val) {
 $tags=$tags.','.$val->tags;
}
$tags=explode(',', $tags);
for($i=1;$i<count($tags);$i++) {
 $tags[$i]='<a href="'.getUrl('act','TS','is_keyword',$tags[$i]).'">'.$tags[$i].'</a>';
}
$tags=implode(' ', $tags);
$tags='<div class="tags" align="center">'.$tags.'</div>';
$content=Context::get('page_content');
$content=$content.$tags;
Context::set('page_content',$content);
?>
```

In the above example, we insert the tag list code to the current HTML page.

2.2.4 How to Use XE XML Query

In XE Addon, the data in a DB that has been created by another module can be utilized by using XML Query.

In this case, make a subdirectory named 'queries' under the 'addon' directory, and save an XML file in which XML query statements are defined. The query can be executed like in the example above:

```
$document_list=executeQueryArray('addons.tag_list.getModuleDocumentTags',$obj);
```

2.2.5 What to Consider When Creating Addons

The considerations to create an addon are listed below:

- Make sure that there is no space before or after <?php ... ?>, because the addons of XE will be inserted into many parts of all modules. If there is any space, malfunction will occur even when before_display_content is called.

- The XE core does not separately handle exceptions that may occur while programming addons. Therefore, the routine to check the current call situation must be well-established to prevent such conflicts from occurring.
- If a serious error occurs on the Web site due to erroneous addon coding, edit the files/cache/activated_addons.cache.php file and upload it again.

XE Addon can perform powerful actions. However, the inappropriate use of code may result in an unintended outcome or may even stop XE. It is recommended to refer to the default addons.

2.3 Widgets

Widgets are components used to display data on screen. Widgets can work together with existing modules - such as recent posts, member profiles - or extract data from external APIs. Widgets can be added on any page or directly in layouts and allow for easy customization of the content displayed.

Widgets are manually entered by the administrator on the page module and are stored in <img/> tags. When calling a web page to be display, the trigger widgetController::triggerWidgetCompile() executes the code between <img/> tags using widgetproc() and transforms it into the correct html code.

2.3.1 Creating config/info.xml

The info.xml file holds information about the widget author, version and other configuration variables.

```
<?xml version="1.0" encoding="UTF-8"?>
<widget version="0.2">
   <title xml:lang="en">Widget title</title>
   <description xml:lang="en">Widget description</description>
   <version>Widget version</version>
   <date>Widget creation date</date>
   <author email_address="..." link="...">
      <name xml:lang="en">Author name</name>
   </author>
   <extra vars>
      <var id="extensionVariableName">
         <name xml:lang="en">Extension variable name</name>
         <type>Type of extension variable: text | textarea | select | select-multi-order
| mid | mid-list | menu </type>
      </var>
   </extra vars>
</widget>
```

2.3.2 Creating widget class

What a widget does is implemented in a class file named widgetName.class.php. All classes that implement a widget must inherit from WidgetHandler and must implement the proc() method:

```
<?php
class myWidget extends WidgetHandler {
   function proc($args) {
           // .. Widget implementation ..

           // Template, specify the path of the skin (skin, colorset according to the
value)
           $tpl_path = sprintf('%sskins/%s', $this->widget_path, $args-> skin);
           Context::set ('colorset', $args->colorset);

           // Template file name
           $tpl file = 'HTML template file except the extension ';

           // Template compilation
           $oTemplate = &TemplateHandler::getInstance();
           return $oTemplate->compile($tpl path, $tpl file);
   }
}
?>
```

2.3.3 Extra Vars

These are variables that a widget uses to get data in the admin part of the widget just before inserting the widget into a page. For each of these variables you can set the type of input to get their value which will be automatically created on the page:

- text: Generic text type
- textarea: Text type containing paragraphs
- select: Select one from several items
- select-multi-order: Used to select one from several items and change the order of them as in the following figure.

- mid: Select only one module.
- mid_list: Select multiple modules.
- menu: Select one of the site menus

3. Working with DB

This chapter describes how to work with the database.

3.1 Introduction

XE has a database-agnostic database abstraction layer. This means that you can use XE with many different database management systems and you can easily switch your XE from one provider to another. XE supports MySQL, MS SQL, CUBRID, Postgres, SQLite3 and Firebird.

To handle this, you write the entire database schema and the queries in XML - using XE's XML Schema Language and XML Query Language.

Here is an example of an XML Schema file:

```
# Excerpt from ./modules/member/schemas/member.xml
<table name="member">
   <column name="member_srl" type="number" size="11" notnull="notnull"
primary key="primary key" />
   <column name="user id" type="varchar" size="80" notnull="notnull"
unique="unique user id" />
   <column name="find_account_question" type="number" size="11" />
   <column name="allow_mailing" type="char" size="1" default="Y" notnull="notnull"
index="idx_allow_mailing" />
   <column name="limit date" type="date" />
   <column name="regdate" type="date" index="idx regdate" />
   <column name="description" type="text" />
   <column name="list_order" type="number" size="11" notnull="notnull"
index="idx_list_order" />
</table>
```

If there is a table.xml in the modules included when you first installed XE, the table will be automatically created. If there is a table.xml when you install additional modules after the installation of XE, you can see **Install Module** button on the admin page. You can then create queries for this table through XML files:

```
#./modules/member/queries/getMemberInfo.xml
<query id="getMemberInfo" action="select">
   <tables>
      <table name="member" />
   </tables>
   <columns>
      <column name="*" />
   </columns>
   <conditions>
      <condition operation="equal" column="user_id" var="user_id" notnull="notnull" />
   </conditions>
</query>
```

Calling this query from PHP is as simple as:

```
         $args->user_id = $user_id;
         $output = executeQuery('member.getMemberInfo', $args);
```

3.2 XML Schema Language Reference

The schema of XE's database tables is defined through XML files. These are found inside the schemas folder of each module.

An XML schema file consists of one root <table> tag and one or more children <column> tags. Here are the attributes you can use with each tag:

Table 3-1 Attribute of <table> element

Attribute	Description
name	Name of the table to be created. The table prefix (xe_) will be automatically added and doesn't need to be specified. Note: this has to be the same as the name of the XML file.

Table 3-2 Attributes of <column> element

Attributes	Description
name	Name of the column.
type	Data type that the column will store. Has to be one of: • number • bignumber • varchar • char • text • bigtext • date • float The parser will automatically map this data type to a database-specific data type. For instance, bignumber corresponds to bigint in MySQL. Read more about how each data type is mapped to database specific data types, see "Table 3-4 Data type mapping between XE and DBMSs".
size	Defines the size of the column. This is used for numeric or character types. For numeric: it can represent the precision. For character: it represents the number of characters that the string can hold.
default	Specifies a default value for the column.
notnull	Specifies whether column allows null values. If a column allows null values, just omit this attribute. Otherwise, add it like this: e.g.) notnull = "notnull"
primary_key	Specifies if this column should be used as a primary key for the table. You can set primary_key="primary_key" for each column to bind the two attributes into primary_key.

Attributes	Description
index	Creates an index for the column. The value represents the name of the index to be created. If the value is used in more than one column, a combined index will be created. e.g.) index="idx_list_order"
unique	Creates a unique index for the column. The value represents the name of the index to be created.
auto_increment	Specifies if column value should be auto incremented. e.g.) auto_increment="auto_increment"

3.3 XML Query Language

Xpress Engine does not use direct SQL queries. Instead, database queries are written in XML in order to support a variety of DBMSs.

3.3.1 How to use

XML Query can be used in modules, addons, widgets and others as follows:

```
$args->name = "zero";
$output = executeQuery("member.getMemberInfo", $args);
```

The executeQuery() function is the alias for the DB::executeQuery() function in ./classes/db/DB.classs.php. This function manipulates the actual DB data and receives the output after the XML Query is parsed as native SQL according to the database used.

```
function executeQuery($xml_query_name, $args = null);
```

- The first parameter is the name of the XML Query to be executed. The name is decided according to module name and query ID.
- The second argument is a type of stdClass and is used to pass extra data to the query. This parameter can be null.
- The result is returned as an object of the object class.
  - A query failure can occur when $output->toBool() is FALSE, but if it's TRUE, it means the query has been normally executed.
  - The result data of a select statement is put in $output->data variable and returned.

3.3.2 XML elements used

```
<query id="query id" action="select|update|delete|insert">
 <tables>
         <table name="tableName" alias="alias" />
 </tables>
 <columns>
         <column name="columnName" alias="alias" />
 </columns>
 <conditions>
         <condition operation="doSomething" column="column1" var="variable"
filter="filterType" default="default" notnull="notnull" minlength="minimumLength"
maxlength="maximumLength" pipe="TheConcatenationOperator "/>
         <group pipe="pipe">
                 <condition operation="anotherOperation" column="column" var="variable"
filter="filterType" default="default" notnull="notnull" minlength ="minimumLength"
maxlength="maximumLength" pipe="TheConcatenationOperator"/>
         </group>
 </conditions>
 <navigation>
         <index var="var" default="default" order="desc|asc" />
         <list_count var="var" default="default" />
         <page_count var="var" default="default" />
         <page var="var" default="default" />
 </navigation>
 <groups>
         <group column="GroupBy daesang" />
 </groups>
 </query>
```

The following table shows the XML elements and attributes used in the XML queries.

Table 3-3 XML elements used

Elements	Attributes	Description
<query>		Query XML's root element

Elements	Attributes	Description
	id	ID for searching the query. Use module.query_id to search and use query XML files.
	action	There are four types of actions, which are select, update, delete and insert.
	alias	Alias name of the query statement when using a subquery.
<tables>		Joining tables allows the use of multiple <table>.
	name	Original table name (Ignore the prefix in ZeroBoardXE)
	alias	When the original table name is changed and used for join or other services.
<columns>		Columns to be used in the query.
	name	Column name
	alias	Specifies when you want to change it to another name
<conditions>		It is used to form a conditional clause. If you want conditional clauses to be multiple groups, you can bind them by using <group> tags.
<group> ... </group>		When conditional clauses are used as groups, you can specify the conditions between the groups by using pipe="and|or"
<condition>		Creates a conditional clause.
	operation	Its operation is processed by the operators below: • equal : column = (var|default) • more : column >= (var|default) • excess : column > (var|default) • less : column <= (var|default) • below : column < (var|default) • notequal : column != (var|default) • notnull : column is not null • null : column is null • like_prefix : column like '%var|default' • like_tail : column like 'var|default%' • like : column like '%var|default%' • in : column in (var|default) • notin : column not in (var|default)
	column	Specifies the column name.
	var	Specifies the key value of the second parameter in the executeQuery() function.

Elements	Attributes	Description
	filter	Filters the condition of the var value. Supports the following filters: • email, email_address: mail format • homepage: homepage format such as http|https:// • userid, user_id: user id format of XE zeroboard (The first two characters must be alphabet. From third characters, it must be the number+alphabet+ _ format.) • number: numbers allowed • alpha: alphabetical characters allowed • alpha_number: both numbers and alphabetical characters al lowed
	default	It will be replaced with the default value when the var value is null. The following function values are available: • ipaddress(): IP address • unixtime(): Unix time (time() function in php) • curdate(): YYYYMMDDHHIISS • plus(int count): column = column + count • minus(int count): column = column – count • multiply(int arg): column = column * arg • sequence(): Executes getNextSequence() of XE.
	notnull	Does a not null check.
	minlength	Checks the minimum length.
	maxlength	Checks the maximum length.
	pipe	Specifies the condition, such as and|or.
<navigation>		Navigation supports the sort order (order by) or paging.
<index>		Specifies the columns to be sorted and the sorting methods.
	var	A variable that contains the column names
	default	A column name to sort by default; in case that the var value is n ot specified.
	order	asc|desc
<list_count>		Makes it possible to receive the results of paging.
	var	Rows of list
	default	A default rows value when the var value is not specified.
<page_count>		Specifies the number of navigations on the bottom page when ca lculating paging.
	var	The number of paging navigations
	default	The number of default paging navigations when the var value is not specified.
<page>		Specifies the current page number.
	var	A variable to specify the n-th page
	default	A default page number used when a var value is not specified.
<groups>		Enables the use of the group by clause.

Elements	Attributes	Description
	column	The name of group by column.

3.3.3 Examples of using XML subquery

In XE 1.5 or higher version, you can use a subquery. The following examples show how to create a subquery for each type.

SELECT statement

Using SQL

```
select *,
(select count(*) as "count"
         from "xe_documents" as "documents"
          where "documents"."user_id" = "member"."user_id"
     ) as "totaldocumentcount"
  from "xe member" as "member"
  where "user_id" = 7
```

Using XML subquery

```
<query id="getStatistics" action="select">
   <tables>
      <table name="member" alias="member" />
   </tables>
   <columns>
      <column name="*" />
      <query id="getMemberDocumentCount" alias="totalDocumentCount">
         <tables>
                  <table name="documents" alias="documents" />
         </tables>
         <columns>
                  <column name="count(*)" alias="count" />
         </columns>
             <conditions>
                <condition operation="equal" column="documents.user_id"
default="member.user_id" />
             </conditions>
      </query>
   </columns>
   <conditions>
      <condition operation="equal" column="user_id" var="user_id" notnull="notnull" />
   </conditions>
</query>
```

WHERE clause

Using SQL

```
SELECT *
FROM xe_member as member
WHERE regdate = (SELECT MAX(regdate) as regdate
                           FROM xe_documents as documents
                           WHERE documents.user_id = member.user_id)
```

Using XML subquery

```
<query id="getMemberInfo" action="select">
   <tables>
      <table name="member" alias="member" />
   </tables>
   <columns>
      <column name="*" />
   </columns>
   <conditions>
      <condition operation="equal" column="regdate" notnull="notnull">
       <query alias="documentMaxRegdate">
```

```
         <table>
                 <table name="documents" alias="documents" />
         </table>
         <columns>
                 <column name="max(regdate)" alias="maxregdate" />
         </columns>
                    <conditions>
                       <condition operation="equal" column="documents.user_id"
var="member.user_id" notnull="notnull" />
                    </conditions>
       </query>
      </condition>
   </conditions>
</query>
```

FROM clause

Using SQL

```
SELECT m.member_srl, m.nickname, m.regdate, a.count
FROM (
 SELECT documents.member srl as member srl, count(*) as count
 FROM xe_documents as documents
 GROUP BY documents.member_srl) a
         INNER JOIN xe_members m on m.member_srl = a.member_srl
```

Using XML subquery

```
<query id="getMemberInfo" action="select">
   <tables>
      <table query=nfo" action="selec             <table>
                 <table name="documents" alias="documents" />
         </table>
         <columns>
                 <column name="member_srl" alias="member_srl" />
                 <column name="count(*)" alias="count" />
         </columns>
            <groups>
              <group column="member srl" />
            </groups>
      </table>
      <table name="member" alias="m" type="inner join">
          <conditions>
             <condition operation="equal" column="m.member" default="a.member srl" />
          </conditions>
      </table>
   </tables>
   <columns>
      <column name="m.member srl" />
      <column name="m.nickname" />
      <column name="m.regdate" />
      <column name="a.count" />
   </columns>
</query>
```

3.4 Data Type Mapping

The following table shows the data type mapping between XE and each DBMS.

Table 3-4 Data type mapping between XE and DBMSs

XE	MySQL	CUBRID	MS SQL
number	bigint	integer	int
bignumber	bigint	numeric(20)	bigint
varchar	varchar	character varying	varchar
char	char	character	char
text	text	character varying(1073741823)	text
bigtext	longtext	character varying(1073741823)	text
date	varchar(14)	character varying(14)	varchar(14)
float	float	float	float
tinytext		character varying(256)	

3.5 XML Query Parser

The XML Query Parser class receives an XML Query file as input, parses it, and generates a PHP file containing related class objects of all the info that is necessary for creating an SQL query (the type of query - select, update, insert, delete -, the expressions used, the join and filtering conditions, the group by and order by clauses). This PHP file is then used as input for each of the DB classes, which will create specific SQL for each DBMS (using the appropriate escape characters and custom language constructs).

For example, say we have the following XML query:

```
# ./modules/document/queries/getCategory.xml
<query id="getCategory" action="select">
   <tables>
      <table name="document_categories" />
   </tables>
   <conditions>
      <condition operation="equal" column="category srl" var="category srl"
filter="number" notnull="notnull" />
   </conditions>
</query>
```

When the query is called (using the executeQuery function) the engine checks to see if the corresponding PHP file was created, and if not invokes the XML Query Parser class, generates the PHP file and saves it under ./files/cache/queries.

```
# ./files/cache/queries/document.getCategory.1.5.0.8.cache.php
<?php if(!defined('__ZBXE__')) exit();
$query = new Query();
$query->setQueryId("getCategory");
$query->setAction("select");
$query->setPriority("");

$category_srl1_argument = new ConditionArgument('category_srl', $args->category_srl,
'equal');
$category srl1 argument->checkFilter('number');
$category srl1 argument->checkNotNull();
$category_srl1_argument->createConditionValue();
if(!$category_srl1_argument->isValid()) return $category_srl1_argument->getErrorMessage();
if($category_srl1_argument !== null) $category_srl1_argument->setColumnType('number');

$query->setColumns(array(
new StarExpression()
));
$query->setTables(array(
new Table('`testtesttest_document_categories`', '`document_categories`')
));
$query->setConditions(array(
new ConditionGroup(array(
new ConditionWithArgument('`category_srl`',$category_srl1_argument,"equal")))
));
$query->setGroups(array());
$query->setOrder(array());
$query->setLimit();
return $query; ?>
```

Then, the database specific executeQuery method is invoked and the output of the file above will be used as input to that method. The DB class generates the sql query and executes it. For instance, for the query above, the SQL will be:

```
select * from "xe_document_categories" as "document_categories" where ("category_srl" =
15)
```

You can see that the cache.php file also contains information about the column types. This info is extracted from the table schema files. XE first searches for the schema file inside ./modules/<module_name>/<table_name> and if none is found, it looks inside each module for a file named <table_name> until one is found.

3.6 XE Database Classes

XE has a custom class for every DMBS it supports. They are the ones responsible for generating custom SQL syntax for each DBMS.

For example, the classes that come with XE core are:

```
DB.class.php
DBMysql.class.php
DBCubrid.class.php
DBMssql.class.php
….
```

All these are stored under ./classes/db.

All custom DB classes inherit from a common DB class. In your code, you will only use the generic DB class and XE can figure out at runtime what DB class implementation to use.

4. Working with Forms

This chapter describes how to work with forms.

4.1 Introduction

A form is used to send a user input to the server. In addition to the general form submission, XE supports a ruleset to check the validity of the input values for form submission. With this ruleset function, you don't have to create a script to verify input values.

In XE, all form submission is handled through AJAX calls. This means that for each form we need:

- the form markup and design
- a server side method to be called on form submission

The XE parts that work together for this are:

- The form template file: defines the layout and fields of the form
- The controller method that will handle the form submission (inside the controller file)
- The ruleset XML file to validate forms

4.2 Example of XE Form

The form we are going to build is very simple - the user will be asked to enter his name and then the page will greet him by it. The module will have only one view - that will display the hello message after the user entered his name or a form for entering the name otherwise.

You can download a working version of the module: hello.zip. In order to follow the tutorial you should download just the start files: hello-tutorial.zip.

4.2.1 Creating the form view

Let's first create the look of the form - it will contain only an input box and a submit button. Name the file name.html and place it under ./modules/hello/tpl/.

```
<h1>Enter your name:</h1>
<form id="name_form" action="./" method="post" ruleset="say_hello">
       <input type="hidden" name="module" value="hello" />
       <input type="hidden" name="act" value="procHelloGreet" />
       <input type="text" name="name" id="name" value="" />
       <br />

       <input type="submit" value="OK" />
</form>
```

To submit forms in XE, you need to specify the module and the action to be used to submit data, and set the ruleset to validate the data. In the example above, 'procHelloGreet' action of 'hello' module is used to submit data, and 'say_hello' ruleset file is used to validate the data.

> **Note**
>
> The ruleset property of the form element is the name of ruleset file to be applied. Attaching '@' to the property value means that it should refer to the ruleset created dynamically in XE. For example, XE 1.5 needs users to select user_id or email_address as a login account. In this case, the dynamic ruleset is required since the method to validate login data should differ depending on the login account type. The dynamic ruleset file is saved in the files/ruleset directory.

Let's create a view method for displaying the template file.
Inside ./modules/hello/hello.view.php add the following method:

```
      /**
       * @brief Display form for entering a name
       **/
      function dispHelloName() {
         $this->setTemplateFile('name');
      }
```

Then, let's make it available to end-users by listing it inside ./modules/hello/conf/module.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<module>
  <grants />
  <permissions />
  <actions>
     <action name="dispHelloName" type="view" standalone="true" index="true" />
  </actions>
</module>
```

You should now be able to see the form by accessing /?module=hello:

Figure 4-1 A form to enter a name

4.2.2 Adding the XML ruleset file and the controller action

Right now our form doesn't do anything, so let's add a method for retrieving the username and displaying the hello message.

Inside ./modules/hello/hello.controller.php add the following method. Using the ruleset file, you need to specify the next action to be executed. In the following example, we used setRedirectUrl to redirect to dispHelloName page after procHelloGreet is executed.

```
 /**
  * Action for handling the name input form submission
  * Retrieves the name given by the user and passes it on for displaying the greeting
screen
  */
 function procHelloGreet(){
         $name = Context::get('name');
         $this->setRedirectUrl(getNotEncodedUrl('', 'module', 'hello', 'act',
'dispHelloName', 'name', $name));
 }
```

Then, make it known by adding this line, inside ./modules/hello/conf/module.xml under the <actions> tag:

```
<action name="procHelloGreet" type="controller" standalone="true" />
```

In order for XE to validate the form data, we need to add an XML ruleset file. Name it say_hello.xml and place it under ./modules/hello/ruleset:

```
<?xml version="1.0" encoding="utf-8"?>
<ruleset version="1.5.0">
   <fields>
      <field name="name" required="true" />
   </fields>
</ruleset>
```

For more information on the elements and attributes used in the ruleset file, see "2.1.5 Using ruleset"

4.2.3 Showing the greeting message

Update the dispHelloName method, inside ./modules/hello/hello.view.php:

```
      /**
       * @brief Display form for entering a name
       **/
      function dispHelloName() {
                  $name = Context::get('name');
                  if(isset($name)){
```

```
                        $hello message = "Hello " . $name;
                        Context::set('hello message', $hello message);
                }
    $this->setTemplateFile('name');
     }
```

Now, let's also update the name template file (./modules/hello/tpl/name.html):

```
<h1 cond="isset($hello_message)">{$hello_message}</h1>
<block cond="!isset($hello message)">
   <h1>Enter your name:</h1>
   <form id="name form" action="./" method="post" ruleset="say hello">
      <input type="hidden" name="module" value="hello" />
      <input type="hidden" name="act" value="procHelloGreet" />
      <input type="text" name="name" id="name" value="" />
      <br />

      <input type="submit" value="OK" />
   </form>
```

</block>If you reload the page in the browser, you should now see the greeting message:

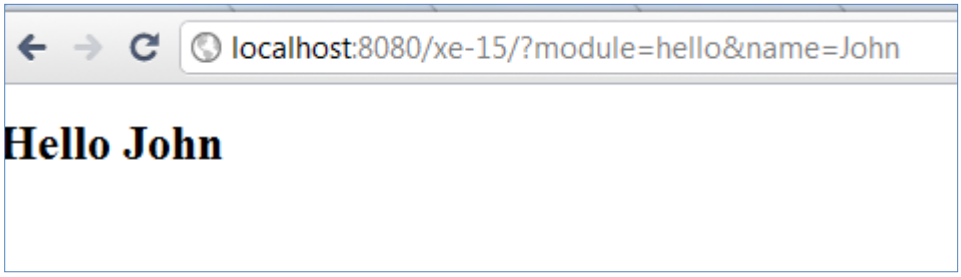

Figure 4-2 Display the greeting message

This is it!

5. Using Document Module

This chapter describes how to use the document module which is basically provided by XE.

5.1 Introduction

XE was built with a modular structure in mind. This way, you can easily extend its core functionality while taking advantage of the existing building blocks.

Among the most important bricks when building your own additional features are the document module. This is the recommended module you should use when creating custom modules that work with content.

Some of the features that come bundled with this are:

- built-in functions for creating and retrieving content
- information about number of comments, number of views and other useful statistics
- revision history
- ability to easily organize content - through categories or tags
- batch editing
- easy integration with other modules in XE

To better grasp how you can take advantage of these modules, you should take a look at the forum, wiki, textyle and issue tracker modules - they all use documents to store their content.

5.2 Document Module

5.2.1 Creating documents

The method for creating documents is in the documentController - .\modules\document\document.controller.php . Here is an example of creating one:

```
$obj->title = "My sample document";
$obj->content = "Hello World!";
$obj->tags = "demo, hello";
$document_srl = getNextSequence();
$obj->document_srl = $document_srl;
$obj->module_srl = $this->module_srl;
$obj->allow comment = 'Y';
$obj->allow trackback = 'Y';
$oDocumentController = &getController('document');
$output = $oDocumentController->insertDocument($obj);
```

All documents are stored in the [DB flag]_documents table. Besides the default fields, you can easily add your own custom fields with the extra_vars feature. These represent attributes that can be associated to all documents belonging to a certain module type, so that just your custom documents share them.

The name of the custom fields and info about their type is stored in the [DB flag]_document_extra_keys table. You can add new keys using the insertDocumentExtraKey method in the documentController. The values of the new keys are stored in the [DB flag]_document_extra_vars table. In order to add these you can use the insertDocumentExtraVar method o the documentController class.

5.2.2 Document attributes

The following table describes the document attributes.

Table 5-1 Document attributes

Attributes	Description
document_srl	Represents the unique ID of a document.
module_srl	Represents the module instance that the document is linked to.
category_srl	Represents the ID of the document category (document categories are stored in the [DB flag]_document_categories table).
lang_code	Language code of the document - this is for having different versions of the same document in different languages.
is_notice	Marks a document as being important. You can use this attribute for displaying notices at the top of a document list, for instance.
title	Document title
content	Document content
readed_count	Number of times that the document has been viewed.
voted_count	Number of votes the document received. This is achieved through integration with the point module.
blamed_count	Number of times the document has been marked as inappropriate.
comment_count	Number of comments associated to the document.
trackback_count	Number of trackbacks for the document.
uploaded_count	Number of attachments of the document.

Attributes	Description
document_srl	Represents the unique ID of a document.
password	Used for secret documents, to allow access to a document only to the users who know the password.
user_id, user_name, nick_name, member_srl	Information on the document owner
tags	Document tags, stored as comma separated values.
regdate	Date when the document was created.
last_updated	Date when the document was last modified.
ipaddress	IP address of the user who created the document.
comment_status	Whether to allow comments to the document (ALLOW: allow comments, DENY: not allow comments)
status	Document status (PRIVATE: private, PUBLIC: public, SECRET: secret, TEMP: temporarily saved)

These attributes represents the fields of [DB flag]_documents table. The model class of document is document.item.php.

5.2.3 Document URLs

Documents can be accessed in many ways.

First of all, they all expose a permalink with the following structure.

```
http://<xe_name>/<document_srl>
```

There is also a more user friendly name of accessing any document in XE:

```
http://<xe_name>/entry/<document_title>
```

If the document title is very long or contains spaces, you can always define a document alias by selecting **Contents > Document** from **Dashboard**. A document can have more than one alias. The URL structure for accessing a document by its alias is:

```
http://<xe_name>/entry/<alias>
```

Besides these built-in ways to access a document, you can define your own view methods in your custom module.

> **Note**
>
> The examples above are available only when the mod_rewrite is enabled.

5.2.4 Document categories

Each document can belong to a category. Categories are saved in the [DB flag]_document_categories table and can be either hierarchical or nonhierarchical.

Managing categories is done through the documentController and documentModel class. documentController contains following methods related to managing categories:

- insertCategory
- deleteCategory
- moveCategoryUp

- moveCategoryDown
- procDocumentMoveCategory
- updateCategory
- updateCategoryCount

documentModel contains following methods related to managing categories:

- getCategory
- getCategoryChildCount
- getCategoryDocumentCount
- getCategoryHTML
- getCategoryList
- getDocumentCategories
- getCategoryTplInfo

5.2.5 Document revision history

The document module has a mechanism for keeping the revision history of documents. On each update, a log entry is added automatically by the updateDocument method of the documentController class.

By default, revision history is disabled. In order to enable it, you need to check the **Use History** option from the document partial configuration page.

Revision history is saved in the [DB flag]_document_histories table. In order to retrieve the log for a document you can use the following methods from the documentModel class:

- getHistories
- getHistory

5.2.6 Retrieving documents

The method used for searching through documents is getDocumentList from the documentModel. This allows you to filter documents based on:

- module srl
- category
- member who created the document
- title
- content
- tags
- type: notice, secret
- number of views, votes, etc.
- date created, date modified

6. API Reference

This chapter describes XE's utility functions: global functions and functions for each class.

6.1 XE Global Functions

The global functions of XE are defined in the XE_ROOT/config/func.inc.php file.

debugPrint(mixed OBJECT)

This is the debugging function.

The value of __DEBUG__ must be defined as one or higher in the XE_ROOT/config/config.inc.php file. You can select the method by which to obtain the result value in accordance with the __DEBUG_OUTPUT__ value.

- 0: Connect to files/_debug_message.php to display
- 1 : Display as a comment at the bottom of HTML (When the response method is HTML)
- 2: Display on the Firebug console (PHP >= 5.2.0. Firebug/FirePHP plugin required)

instance getController(string MODULE_NAME)

The function imports the controller instance of a module.

```
// If you want to get the document.controller.class instance
$oDocumentController = &getController('document');
```

instance getAdminController(string MODULE_NAME)

This function imports the Admin controller instance of a module.

```
// If you want to get the documentAdminController instance
$oDocumentAdminController = &getAdminController('document');
```

instance getView(string MODULE_NAME)

This function imports the view instance of a module.

```
// If you want to get the rssView instance
$oRssView = &getView('rss');
```

instance getAdminView(string MODULE_NAME)

This function imports the Admin view instance function.

```
// If you want to get the adminAdminView instance
$oAdminAdminView = &getAdminView('admin');
```

instance getModel(string MODULE_NAME)

This function imports the model instance of a module.

```
// If you want to get the documentModel instance
$oDocumentModel = &getModel('document');
```

instance getAdminModel(string MODULE_NAME)

This function imports the Admin model instance of a module.

```
// If you want to get the documentAdminModel instance
$oDocumentAdminModel = &getAdminModel('document');
```

instance getAPI(string MODULE_NAME)

This function imports the API instance of a module.

```
// If you want to get the boardAPI instance
$oBoardAPI = &getAPI('board');
```

instance getWAP(string MODULE_NAME)

This function imports the WAP instance of a module.

```
// If you want to get the boardWAP instance
$oBoardWAP = &getWAP('board');
```

instance getClass(string MODULE_NAME)

This function imports the class instance of a module.

```
// If you want to get the documentClass instance
$oDocumentClass = &getClass('document');
```

Object executeQuery(string QUERY_ID, stdClass PARAM)

This function executes an XML Query. The result data is returned to the object of the object class.

The query failure shows that Object::toBool() is FALSE. If the value is TRUE, it shows that the query was normally executed.

The result data of the select statement is put in the Object::data variable and returned to the object.

Object executeQueryArray(string QUERY_ID, stdClass PARAM)

This function has the same feature as executeQuery(), but it returns in array even though the result of the Object:data variable is a single row when it is selected.

int getNextSequence()

This function imports the next sequence number.

XE uses one sequence internally, and all keys, such as member_srl, module_srl and document_srl, are set by using this function. The document_srl does not increase by +1 gradually, but XE uses the sequence in such a way because it has many advantages.

string getUrl(['',] string KEY, string VALUE [,string KEY, string VALUE ...])

This function creates URLs.

XE changes a URL with a given parameter value, and then returns the URL to the currently requested RequestURI. If the value of the first parameter is '', XE creates a URL to RequestURI with the added args_list.

```
// domain : www.example.com
// xe install path : /xe
// request url : www.example.com/xe/index.php?module=sample&act=dispSampleAct

$reset_url = getUrl('', 'module', 'reset');
print r($reset url);
// result : /xe/index.php?module=reset

$update_url = getUrl('module', 'update');
print_r($update_url);
// result : /xe/index.php?module=update&amp;act=dispSampleAct
```

string getFullUrl(['',] string KEY, string VALUE [,string KEY, string VALUE ...])

This function creates a URL starting with http://.

```
// domain : www.example.com
// xe install path : /xe
// request url : www.example.com/xe/index.php?module=sample&act=dispSampleAct

$reset url = getFullUrl('', 'module', 'reset', 'mid', 'samplemid');
print r($reset url);
// result : http://www.example.com/xe/index.php?module=reset&amp;mid=samplemid
```

string getNotEncodedFullUrl(['',] string KEY, string VALUE [,string KEY, string VALUE ...])

This function creates an unencoded URL. It has the same feature as getFullUrl().

```
// domain : www.example.com
// xe install path : /xe
// request url : www.example.com/xe/index.php?module=sample&act=dispSampleAct

$reset url = getNotEncodedFullUrl('', 'module', 'reset', 'mid', 'samplemid');
print r($reset url);
// result : http://www.example.com/xe/index.php?module=reset&mid=samplemid
```

string getAutoEncodedUrl([‘’,], string KEY, string VALUE [,string KEY, string VALUE …])

This function creates a URL which is encoded automatically but not repeatedly.

```
// domain : www.example.com
// xe install path : /xe
// request url : www.example.com/xe/index.php?module=sample&act=dispSampleAct

$reset url = getAutoEncodedUrl('', 'name', '<script>', ‘title’, ‘&lt;title’);
print r($reset url);
// result : http://www.example.com/xe/index.php?name=&lt;script&rt;&amp;title=&lt;title
```

string getSiteUrl(string DOMAIN, ['',] string KEY, string VALUE [,string KEY, string VALUE ...])

This function creates a URL for a virtual site. The first parameter domain gets a domain or vid.

```
// domain : www.example.com
// xe install path : /xe
// request url : www.example.com/xe/index.php?module=sample&act=dispSampleAct

$reset url = getSiteUrl('site id', '', 'module', 'reset');
print_r($reset_url);
// result : http://www.example.com/xe/index.php?module=reset&vid=site_id
```

string getNotEncodedSiteUrl(string DOMAIN, ['',] string KEY, string VALUE[,string KEY, string VALUE...])

This function creates an unencoded URL. It has the same feature as getSiteUrl().

string getFullSiteUrl(string DOMAIN, ['',] string KEY, string VALUE [,string KEY, string VALUE ...])

This function creates a URL starting with http:// for a virtual site.

int ztime(string STR)

This function changes the time value of YYYYMMDDHHIISS format to Unix time.

string getTimeGap(string DATE, string FORMAT)

This function displays the YYYYMMDDHHIISS time format as a string showing the remaining minutes/hours to the current time. If the time gap is more than 24 hours, it displays the time in the FORMAT.

string getMonthName(int MONTH, bool SHORT)

This function displays month names.

```
print_r(getMonthName(3, true));
// result : Mar

print r(getMonthName(10, false));
// result : October
```

string zdate(string STR, string FORMAT, bool CONVERSION)

This function changes the time value of YYYYMMDDHHIISS format into a desired time format.

```
print_r(zdate('19830310123644', 'Y-m-d H:i:s'));
```

```
// result : 1983-03-10 12:36:44
```

string cut_str(string STRING, int CUT_SIZE, string TAIL)

This function cuts a string to a certain size and then adds a tail to the back of the string.

```
print_r(cut_str('All roads lead to XE', 3, '...'));
// result : All…
```

string removeHackTag(string CONTENT)

This function removes code suspected as hacking attempts.

bool isCrawler(string AGENT)

This function inspects login user agent and its IP to check whether it is a robot or not.

6.2 Context Class

Context receives the value of GET/POST and passes variables and diverse information to a template. In addition, it identifies whether the request is XMLRPC, JSON or GET/POST.

Context::set(string KEY, mixed VALUE)

This function sets the variables to be passed to a template.

```
Context::set('user_id','user'');
```

Once the template has been set, the {$user_id} format can be used.

mixed Context::get(string KEY)

This function retrieves the variable that has been passed to Request or the value of the set result.

```
$user_id = Context::get('user_id');
```

stdClass Context::gets(string KEY1 [, string KEY2 ...])

This function retrieves multiple values at once and returns them to stdClass.

stdClass Context::getRequestVars()

This function returns the variable passed from a request to stdClass.

Context::addJsFile(string FILE_PATH, bool OPTIMIZED ,string TARGETIE, int INDEX)

This function adds JavaScript files to a template. It only adds the files with an extension ending with js.

Context::addCSSFile(string FILE_PATH, bool OPTIMIZED ,string TARGETIE, int INDEX)

This function adds CSS files to a template.

Context::addJsFliter(string FILTER_NAME)

This function loads a filter written in XML to a template.

Context::setBrowserTitle(string TITLE)

This function specifies the title value of an HTML.

Context::loadJavascriptPlugin(string PLUGIN_NAME)

This function loads JS plugins to a template.

Context::addHtmlHeader(string HEAD)

This function adds a string between <head> and </head> of an HTML.

6.3 Extravar Class

The Extravar class is often used in extended variables and for a module such as a bulletin board.

ExtraItem::setValue(string VALUE)

This function specifies the value of an extended variable.

ExtraItem::getValueHTML()

This function processes and displays the value of the extended variable specified by the ExtraItem::setValue() function as an HTML file according to the type of the extended variable.

ExtraItem::getFormHTML()

This function displays the input form of an HTML result file according to the type of the extended variable.

6.4 Mail Class

The Mail class is used to send mails in XE. XE can send mails only when the server is configured to send mails.

Mail::setSender(string NAME, string EMAIL)

This function specifies the sender of mail.

Mail::getSender()

This function returns the sender specified by the Mail::setSender() function.

- It encodes the sender in base64, and returns it if a sender name exists.
- It returns empty string (' ') if a sender name does not exist.

Mail::setReceiptor(string NAME, string EMAIL)

This function specifies the recipient of mail.

Mail::getReceiptor()

This function returns the recipient specified by the Mail::setReceiptor() function.

- It encodes the recipient in base64 and returns it if a recipient name exists.
- It returns empty string (' ') if a recipient name does not exist.

Mail::setTitle(string TITLE)

This function specifies the title of mail.

Mail::getTitle()

This function returns the title of mail that has been encoded in base64.

Mail::setContent(string CONTENT)

This function specifies the body of mail.

Mail::replaceResourceRealPath(mixed MATCHES)

This function converts the address of the image included in the body to an absolute path.

Mail::getPlainContent()

This function returns the body of mail in text.

Mail::getHTMLContent()

This function returns the body of mail in HTML format.

Mail::setContentType(string MODE)

This function specifies the format of the mail body. The default is HTML format.

Mail::send()

This function actually sends a mail.

Specify the sender, recipient, and the body of mail by using the Mail::setSender(), Mail::setReceiptor() and Mail::setContent() functions before sending it.

Mail::checkMailMX(string EMAIL_ADDRESS)

This function checks the validity of a mail address. It returns false if the mail address is incorrect.

Mail::isVaildMailAddress(string EMAIL_ADDRESS)

This function quickly checks the validity of a mail address in regular expression. It returns the $email_address variable unchanged if the address is valid.

6.5 Object Class

The Object class is used to exchange data between modules. Modules inherit from Object and transfer values and states by means of the errors, messages and variables of Object.

Object::Object([int ERROR, string MESSAGE])

An Object creator.

- ERROR: Error code (not an error if this value is 0)
- MESSAGE: Error message (not an error if this value is success)

bool Object::toBool()

This function checks whether the Object is an error. If the return value is true, the object is not an error.

```
$output = executeQuery('document.insertDocument', $obj);
if(!$output->toBool()) {
         $oDB->rollback();
         return $output;
}
```

Object::add(string KEY, mixed VALUE)

This function adds a variable in which the value of the key is KEY to Object.

Object::adds(stdClass OBJECT)

This function adds all the variables in the stdClass to Object.

```
$oObj = new Object();
$params->key1 = "value1";
$params->key2 = "value1";
$oObj->adds($obj);
```

mixed Object::get(string KEY)

This function returns Object's variables whose key is KEY.

stdClass Object::gets(string KEY[, string KEY , ...])

This function returns Object's variables whose key is KEY, in the form grouped by stdClass.

```
$obj = $oObj->gets('key1','key2','key3');
// $obj->key1, $obj->key2, $obj->key3
```

6.6 FileHandler Class

This class defines methods to handle folders and files.

FileHandler::copyDir(string SOURCE_DIR, string TARGET_DIR [, string FILTER] [, string TYPE])

This function is used to copy directories from SOURCE_DIR to TARGET_DIR.

- FILTER: When you copy the subdirectories and files in a directory using a regular expression, the matching files will not be copied.
- TYPE: If the option is 'force,' the function will overwrite any duplicate files that exist in the subdirectory.

FileHandler::copyFile(string SOURCE _FILE, string TARGET_FILE [, string FORCE])

This function copies files from SOURCE_FILE to TARGET_FILE.

- FORCE: If the option is 'Y,' the function will overwrite any duplicate files.

string FileHandler::readFile(string FILE_NAME)

This function reads the content of a file and returns it.

FileHandler::writeFile(string FILE_NAME, string BUFFER [, string MODE])

This function writes the content of BUFFER to a file.

- FILE_NAME: A file to be saved
- BUFFER: Content to be saved
- MODE: 'w' - Save new, 'a' - Update previous save

FileHandler::makeDir(string PATH)

This function creates a directory and its sub-directories of PATH in a recursive way.

```
FileHandler::makeDir(_XE_PATH_ . 'files/cache/nhn/openuitech/sol');
```

FileHandler::removeDir(string PATH)

This function deletes a directory and its subdirectories of PATH in a recursive way.

```
FileHandler::removeDir(_XE_PATH_ . 'files/cache/openiuthech');
```

bool FileHandler::getRemoteFile(string URL, string TARGET_FILE)

This function saves remote files.

- URL: Enter a path starting with 'Http://'.
- TARGET_FILE: Files to be saved

bool FileHandler::createImageFile(string SOURCE_FILE, string TARGET_FILE ,int WIDTH, int HEIGHT, string FILE_TYPE, string THUMBNAIL_TYPE)

This function uses the existing image file to create a thumbnail by specifying size and creation type (ratio or crop).

- SOURCE_FILE: Original image file
- TARGET_FILE: Image files to be saved
- WIDTH: Width of the image to be saved
- HEIGHT: Height of the image to be saved
- FILE_TYPE: Type of Image to be saved
- THUMBNAIL_TYPE: 'ratio,' 'crop,' or thumbnail