

Contents

- 01 Background
- 01 Skill Requirements for Coding in Web Store
- 01 Reference Sites
- 02 Directory Structure
- 03 Code Execution
- 04 Reserved Namespace
- 04 Global Variables
- 04 Extending Custom Logic
- 04 Objects & Functions
- 04 Stack Functions
- 05 Debugging
- 05 Changing Templates
- 05 Developing Custom Modules
- 06 Shipping Modules
- 11 Payment Modules
- 18 Sidebar Modules
- 18 Admin Panel
- 21 Internationalization

Background

Web Store 2.0 is written using qcodo PHP5 web application framework (see <http://www.qcodo.com>). It follows the MVC (Model-View-Controller) application methodology and PHP5's object oriented programming features.

Skill Requirements for Coding in Web Store

- Advanced knowledge of PHP5
- Understanding of Object-Orientated programming, class inheritance, method overloading
- Javascript and jQuery
- CSS3
- HTML

Reference Sites

- www.php.net
- qcodo.com
- api.qcodo.com
- examples.qcodo.com
- jquery.com
- forums.xsilva.net



For more information on customization of the look and feel of Web Store, please see the *Web Store 2.0 Transition Guide* article on the Training page of www.xsilva.com.

Directory Structure

Color coding:

Files/folders that are locked by xsilva and shouldn't be modified.

Files/folders that you can modify but no guarantee given that they won't be overwritten by Xsilva in version upgrades.

Files/folders you can modify and will remain untouched in future versions of Web Store (known in this document as *WS*).

Files that be extended in *custom_includes* folder.

- TT0154M_.TTF (Font file used for drawing Captcha text)
- includes (Contains qcodo installation)
 - data_classes (contains ORM definitions – DO NOT modify)
 - configuration.inc.php (Configuration files used for running qcodo)
 - configuration_install.inc (Default config file used by installer to create *configuration.inc.php*)
 - qcodo (qcodo core logic files)
 - *i18n (You can put language translation files in this folder)
- index.html (Redirects to index.php)
- assets (Contains all core javascript, css, and images used by qcodo)
- index.php (Site root)
- templates (contains templates)
 - admin (Template files used by admin panel)
 - install (Template files used by installer)
 - deluxe
 - basic
 - framework
- custom_includes (extend any file from xlsws_includes folder here)
 - prepend.inc.php (Put your functions and other definitions here. Do not delete this file!)
 - payment (Custom payment modules)
 - shipping (Custom shipping modules)
 - sidebar (Custom sidebar modules)
 - admin (Custom Admin Panel)
- info.php (For PHP info page)
- db_backup (Database backups are stored here)
- install.php (Installer)
- verify-img.php (Displays captcha image)
- favicon.ico (Browser's favicon image)
- xls_admin.php (Admin panel)
- photos (Holds all product images)
- xls_jumper.php
- htaccess/ .htaccess (Rename this file to .htaccess . Apache directive to control SEO details)
- php.ini (PHP configuration required for WS. This file may need to be included in your PHP config)
- xls_payment_capture.php

- `image-validation.gif` (Background image to captcha)
- `xls_soap.php`
- `xlsws_includes` (Contains WS core logic)
 - `payment` (Payment modules)
 - `shipping` (Shipping modules)
 - `sidebar` (Sidebar modules)
 - `cart.php`
 - `category.php`
 - `checkout.php`
 - `crumbtrail.php` (used for printing category or page breadcrumbs. This file cannot be extended)
 - `custom_page.php`
 - `customer_register.php`
 - `family.php`
 - `functions.php` (contains functions specific to WS. This file cannot be extended)
 - `gift_registry.php`
 - `gift_search_detail.php`
 - `gift_search.php`
 - `login.php`
 - `minicart.php` (This file cannot be extended)
 - `minigift.php` (This file cannot be extended)
 - `msg.php` (displays error message)
 - `myaccount.php`
 - `order_track.php`
 - `product.php`
 - `prepend.inc.php`
 - `qextend.php` (contains class and objects that are extended from qcodo. This file cannot be extended)
 - `relations.txt` (ORM relations used in QCodo ORM generation. Do not touch/edit this file unless you are planning on advanced development of WS via modification of database models)
 - `searchbox.php` (Search box displayed on each page. This file cannot be extended)
 - `searchresults.php`
 - `sitemap.php`
 - `skeleton.php` (Common class used by all other files to utilize basic building functions. This file cannot be extended)
 - `sro_track.php`

Unless otherwise specified, please **NEVER** delete a file or folder in Web Store 2.0. Web Store 2.0 may stop working completely if it does not find a referenced file.

Code Execution

Everything in Web Store 2.0 executes via `index.php` as far as visitors and customers are concerned. `index.php` works based on GET parameters and directs requests to files in `custom_includes/xlsws_includes` folder.

- `search=[Search term]` - goes to `searchresults.php`
- `xlspg=[file name]` - executes given filename in `custom_includes` or `xlsws_includes`. If the given file name (parameter is without the `.php` extension but file searched is with `.php`) is found in `custom_includes` folder it will take precedence over `xlsws_includes` folder.
- `product=[Product code]` - dispatch to `product.php` in `custom_includes` or `xlsws_includes` folder)

- `customer_register=` - Customer registration page
- `family=[Family name]` - Show family page
- `cpage=[page key]` - display custom page using `custom_page.php`

Files in `xlsws_includes` folder will include other modules like sidebars, shipping and payment gateways as required. Finally, it will render generated controls and components using defined template in `templates` directory.

Reserved Namespace

Xsilva reserves the right to modify or change any function, class, file or directory names prefixed with `xls`, `_xls`, `xlsws`. Please refrain from utilizing these names.

Global Variables

Global variables are not used in Web Store 2.0 extensively. There are only two global variables that are exclusive to Web Store 2.0

- `$customer` holds the current logged in customer
- `$XLSWS_VARS` is a combination of `$_GET` and `$_POST`

Extending Custom Logic

Any execution file in `xlsws_includes` folder can be extended in `custom_includes` folder. For example, if you want to overwrite product display logic (which is handled by `product.php` in `xlsws_includes` folder), create `product.php` in `custom_includes` folder. Content of the file should be as following:

```
<?php
// Following will stop XLSWS's own form from executing
define('CUSTOM_STOP_XLSWS' , true);
// Include the native product display page
include(XLSWS_INCLUDES . '/product.php');
class dummy_product extends xlsws_product{
// define methods here
}
dummy_product::Run('dummy_product' , templateNamed('index.tpl.php')); // Run the code
?>
```

To find out more on what functions are available on what file and class, please contact Xsilva Support at support@xsilva.com.

Objects & Functions

Along with all the qcodo objects and ORMs, WS has several objects of its own. `qextend.php` contains many of these custom objects. `functions.php` contains several utility functions. Please contact Xsilva Support at support@xsilva.com for more information.

Stack Functions

Since session maintenance of variables can be quite hectic, Web Store 2.0 uses a stack variable system which is maintained across page requests.

- `_xls_stack_add(VariableName , Value)` : Will save Value under VariableName in the stack. You can keep calling this function with the same VariableName to keep adding more values on top of the old one
- `_xls_stack_get(VariableName)` : will return the last added value under VariableName
- `_xls_stack_pop(VariableName)` : will return the last added value under VariableName and remove it from the top of the stack.

Debugging

WS can be debugged with PHP debuggers like Zend. However, for quick debugging and step logging, please use `function _xls_log(object or string or array as parameter)`. `_xls_log` will accept a single variable of any type. And it will print it in Apache's error log or wherever PHP's error logging is done. It will also insert this in the database for viewing by *Admin Panel* system logs.

Changing Templates

Web Store 2.0 ships with a number of templates. For maintenance purposes, we recommend that you duplicate one of the existing template folder and give it a new name. E.g. duplicate deluxe folder and rename it as `my_deluxe` folder. Go to *Admin Panel* to change the template to your new template folder.

You can change the almost anything in the templates.

A few notes on changing templates:

- All css classes are defined in the `templates/[your template]/css/westore.css` file. Feel free to modify any css definitions here.
 - Use firefox's firebug or Safari's developer menu to find out what elements are being styled using what css class.
- `index.tpl.php` is the main server of all html content. It has a `mainPnl` element which pulls contents from other template files.
 - All template file names should match their counterpart file names in `xls_ws_includes` folder.
 - Templates used in emailing are prefixed `email_*`
 - There are some template files which are included as repeating elements. E.g. product image and data in search list page, cart items etc. These template files are named suffixed as `_item`.
- You will find that often a whole component is rendered using a statement like `<?php $this->someItem->Render(); ?>`. These components can be modified on the fly during render. Since they are qcontrols, there are several ways you can modify them:
 - Find out there class type by doing `_xls_log(get_class($this->someItem))`
 - Go to <http://api.qcodo.com> to look properties and methods available on the object
 - You can modify the member variables on the fly `<?php $this->someItem->Render('CssClass=MyClass' , 'Text=My text'); ?>`
 - You can also modify before Render. E.g. the above statement can be changed as `<?php $this->someItem->CssClass = 'MyClass' ; $this->someItem->Text = 'My Text' ; $this->someItem->Render(); ?>`

Developing Custom Modules

Web Store 2.0 contains three types of modules

- Shipping
- Payment
- Sidebar

Several pre-built modules are already included in Web Store 2.0. These files are located in the **xlsws_includes/shipping** , **xlsws_includes/payment** and **xlsws_includes/sidebar** folders respectively. It is possible to alter functionality of existing modules or develop new modules from scratch. Any modules placed in the *custom_includes* folder should have a class name and filename exactly the same without the *.php* extension. For example, in the example below, *my_shipping_module* should be saved in a file called *my_shipping_module.php*.

Modules can have an administrative name in admin panel and also a name that is shown the customer. Hence most modules have a `admin_name()` and `name()` function. It is recommended that as a developer you allow your customers to define the name of your module they would show to their web visitors.

Shipping Modules

Place your shipping file in *custom_includes/shipping* folder.

Here is a sample (non-working) shipping module:

```
<?php

// Include generic Web Store 2.0 generic shipping class
include_once (XLSWS_INCLUDES . 'shipping/xlsws_class_shipping.php');

class my_shipping_module extends xlsws_class_shipping {

    /**
     * The name of the shipping module that will be displayed in the checkout page
     * @return string
     */
    public function name() {
        $config = $this->getConfigValues(get_class($this)); // Get the config values

        if(isset($config['label'])) //if there is a label defined
            return $config['label']; // return the label
        return $this->admin_name(); //otherwise return administrative name
    }

    /**
     * Return the administrative name of the module for WS Admin Panel.
     * It is different than the module name returned in front of the customer.
     * @return string
     */
    public function admin_name() {
        return _sp("My shipping module");
    }

    /**
     * Check if the module is valid or not.
     * Returning false here will exclude the module from both admin panel and checkout page
     *
     * @return boolean
     */
}
```

```
*/
public function check(){
    if(defined('\XLSWS_ADMIN_MODULE'))
        return true;
    $vals = $this->getConfigValues(get_class($this));

    // if nothing has been configured return null
    if(!$vals || count($vals) == 0)
        return false;
    return true;
}

/**
 * Return config fields (as array) for user configuration.
 * The array key is the variable value holder
 * For example if you wanted to have an admin-editable field called Message which is a textbox
 *     $message = new XLSTextBox($objParent);
 *     $message->Text = "Default text"; /// this will be over-written by the user
 *     $message->Required = true; // This is optional. However if you wanted to make a
 *     field compulsory, this is what you would do.
 *     return array('message' => $message);
 *
 *
 * @param QPanel $objParent
 * @return array
 */
public function config_fields($objParent){
    $ret= array();

    $ret['label'] = new XLSTextBox($objParent);
    $ret['label']->Name = _sp('Label');
    $ret['label']->Required = true;
    $ret['label']->Text = $this->admin_name();

    $ret['product'] = new XLSTextBox($objParent);
    $ret['product']->Name = _sp('LightSpeed Product Code');
    $ret['product']->Required = true;
    $ret['product']->Text = 'SHIPPING';

    $ret['markup'] = new XLSTextBox($objParent);
    $ret['markup']->Name = _sp('Mark up ($)');
    $ret['markup']->Required = true;
    $ret['markup']->Text = 3.00;
}
```

```
    return $ret;
}

/**
 * Check config fields
 *
 * The fields generated and returned in config_fields will be passed here for validity.
 * Return true or false
 *
 * Admin panel will ONLY save field configs if all the fields are valid.
 *
 * @param $fields[]
 * @return boolean
 */
public function check_config_fields($fields){
    //check that postcode exists
    $val = $fields['product']->Text;
    if(trim($val) == ''){
        QApplication::ExecuteJavaScript("alert('Must enter a product to be saved in light-speed!')");
        return false;
    }
    return true;
}

/**
 * Return customer fields (as array) that will be shown in the checkout page. Fields can be any
 * qcontrol elements
 * The array key is the variable value holder
 * For example if you wanted to have a service list box where customer will be choosing a type
 * of service
 *
 * $service = new XLSListBox($objParent);
 *
 * $service->Name = _sp('Choose your service type');
 *
 * $service->AddItem(_sp('Service 1') , 'service1');
 *
 * $service->AddItem(_sp('Service 2') , 'service2');
 *
 * $service->SelectedValue = $config['defaultproduct'];
 *
 * return array('service' => $service);
 *
 *
 * @param QPanel $objParent
 * @return array
 */
```

```

public function customer_fields($objParent){
    $ret = array();
    $config = $this->getConfigValues(get_class($this));

    $ret['service'] = new XLSListBox($objParent);
    $ret['service']->AddItem(_sp('Service 1') , 'service1');
    $ret['service']->AddItem(_sp('Service 2') , 'service2');
    $ret['service']->Name = _sp('Choose Service');
    $ret['service']->SelectedValue = 'service1';
    return $ret;
}

/**
 * Check customer fields
 *
 * The fields generated and returned in customer_fields will be passed here for validity.
 * Return true or false
 * s
 * Checkout panel will ONLY continue to checkout if all the fields are valid.
 *
 * @param $fields[]
 * @return boolean
 */
public function check_customer_fields($fields){
    return true;
}

/**
 * Return total for shipping.
 *
 * The return value must be an array
 * Array('msg'=> , 'product'=> , 'markup' => 'price' => ) - Return an array with these keys for
detailed analysis.
 * msg => put error messages in this key. This field is optional
 * product => the shipping product lightspeed should allocate the cost to when order is down-
loaded in POS system
 * markup => Markup (if any) that is has been applied.
 * price => The shipping cost. Return FALSE here if there has been some sort error and you do
not want checkout to go through..
 *
 *
 * @param Cart $cart
 * @param string $country
 * @param string $zipcode
 * @param string $state

```

```
* @param string $city
* @param string $address2
* @param string $address1
* @param string $company
* @param string $lname - last name
* @param string $fname - first name
* @return mixed
*/
public function total($fields , Cart $cart , $country = '' , $zipcode = '' , $state = '' , $city
= '' , $address2 = '' , $address1= '' , $company = '' , $lname = '' , $fname = '' ){
    $config = $this->getConfigValues(get_class($this));

    $weight = $cart->total_weight();
    if(_xls_get_conf('WEIGHT_UNIT' , 'kg') != 'kg')
    $weight = $weight / 2.2; // one KG is 2.2 pounds

    $weight = round($weight *1000 , 0);

    $length = $cart->total_length();
    $width = $cart->total_width();
    $height = $cart->total_height();

    if(_xls_get_conf('DIMENSION_UNIT' , 'cm') != 'cm'){
        $length = round($length *2.54);
        $width = round($width *2.54);
        $height = round($height *2.54);
    }

    // Do your calculations here
    // ...
    // ...

    // if some sort of error had happened
    if($error)
    return array(
        'price' => FALSE
    ,   'msg' => ''
    ,   'markup' => floatval($config['markup'])
    ,   'product' => $config['product']
    );
}
```

```

// if no error happened, then return an array like below
return array(
    'price' => 10.00 // Say shipping cost was $10
    ,
    'msg' => ''
    ,
    'markup' => floatval($config['markup'])
    ,
    'product' => $config['product']
);
}
}
?>

```



- All shipping modules must start with `"include_once (XLSWS_INCLUDES . 'shipping/xlsws_class_shipping.php') ;"` as they need to extend the `xlsws_class_shipping` class
- They must have `name()` and `admin_name()` methods that returns customer front-end module name and backend admin panel name
- `check()` methods shows the module both admin panel and checkout page. *Example code above checks whether this function is being called from admin panel or not. Because you may want to allow the panel in admin panel for config but have it disabled in checkout page.*
- `config_fields()` should return an array of fields that are to be used for admin panel configuration, where the string indices of the array are field names. You can use any qcontrol elements for return. But textbox of `XLSTextBox` and List Box `XLSListBox` is preferred. Please note that argument to this function is a `QPanel`, which all your qcontrol elements should use as parent.
- `check_config_fields()` should validate all the fields the `config_fields()` generated. It's passed parameter is the fields returned by `config_fields()`. The indices are maintained hence you can validate against those.
- Web Store 2.0 admin panel saves chosen config settings. Function like this `$config = $this->getConfigValues(get_class($this)) ;` will return the saved values. It is a returned array which has the config fields keys as indices and there saved values as value.
- `customer_fields()` and `check_customer_fields()` work in similar fashion as above
- The `total()` function should return an array as per code documentation for the code. Please note that all the elements in the returned array format are required! Failing to return appropriate data type may cause checkout page to crash!
- Shipping classes are not serialized between requests. Hence if you have member variables, they won't be instantiated in page reload. You should use the stack system (See **Stack Functions**) to maintain variables across page requests.

Payment Modules

Place your payment file in `custom_includes/payment` folder. There are usually two forms of payment modules. Those which processes payment while customer is retained in the store. Secondly those who are referred to a third party payment processing page which will process customer payment and return the customer to your website on completion (or failure). We call this **jumper payment processing**.

Here is a sample (non-working) payment module:

```
<?php

include_once(XLSWS_INCLUDES . 'payment/xlsws_class_payment.php');
class my_payment_module extends xlsws_class_payment {
    /**
     * Return the administrative name of the module for WS Admin Panel.
     * It is different than the module name returned in front of the customer.
     * @return string
     */
    public function admin_name(){
        return "My payment module";
    }

    /**
     * The name of the module that will be displayed in the checkout page
     * @return string
     */
    public function name(){
        $config = $this->getConfigValues(get_class($this));
        if(isset($config['label']))
            return $config['label'];
        return $this->admin_name();
    }

    /**
     * Return config fields (as array) for user configuration.
     * The array key is the variable value holder
     * For example if you wanted to have a admin-editable field called Message which is a textbox
     *     $message = new XLSTextBox($objParent);
     *     $message->Text = "Default text"; /// this will be over-written by the user
     *     $message->AddAction(new QFocusEvent(), new QAjaxControlAction('moduleActionProxy')); //
     * You do not have to add action to a field. But if you wanted to this is how you would do it.
     *     $message->Required = true; // This is optional. However if you wanted to make a
     * field compulsory, this is what you would do.
     *     return array('message' => $message);
     *
     * @param QPanel $objParent
     * @return array
     */
    public function config_fields($objParent){
        $ret = array();

        $ret['label'] = new XLSTextBox($objParent);
    }
}
```

```

$ret['label']->Name = _sp('Label');
$ret['label']->Required = true;
$ret['label']->Text = 'My payment module';
$ret['service'] = new XLSTextBox($objParent);
$ret['service']->Name = _sp('Enter your Service Details');
$ret['service']->Text = _sp('Service ABCD...') ;

$ret['ls_payment_method'] = new XLSTextBox($objParent);
$ret['ls_payment_method']->Name = _sp('LightSpeed Payment Method');
$ret['ls_payment_method']->Required = true;
$ret['ls_payment_method']->Text = 'Cash';
$ret['ls_payment_method']->ToolTip = "Please enter the payment method (from LightSpeed) you
would like the payment amount to import into";

    return $ret;
}

/**
 * Check config fields
 *
 * The fields generated and returned in config_fields will be passed here for validity.
 * Return true or false
 *
 * Admin panel will ONLY save field configs if all the fields are valid.
 *
 * @param $fields[]
 * @return boolean
 */
public function check_config_fields($fields){
    return true;
}

/**
 * Return customer fields (as array) that will be shown in the checkout page. Fields can be any
qcontrol elements
 * The array key is the variable value holder
 * For example if you wanted to have a service list box where customer will be choosing a type of
service
 *
 * $service = new XLSListBox($objParent);
 *
 * $service->Name = _sp('Choose your service type');
 *
 * $service->AddItem(_sp('Service 1') , 'service1');
 *
 * $service->AddItem(_sp('Service 2') , 'service2');
 *
 * $service->SelectedValue = $config['defaultproduct'];
 *
 * return array('service' => $service);
 *
 */

```

```
*
* @param QPanel $objParent
* @return array
*/
public function customer_fields($objParent){
    $ret= array();
    $config = $this->getConfigValues(get_class($this));
    $ret['bank'] = new QLabel($objParent);
    $ret['bank']->Text = $config['bank'];
    return $ret;
}

/**
 * Check customer fields
 *
 * The fields generated and returned in customer_fields will be passed here for validity.
 * Return true or false
 *
 * Checkout panel will ONLY continue to checkout if all the fields are valid.
 *
 * @param $fields[]
 * @return boolean
 */
public function check_customer_fields($fields){
    return true;
}

/**
 * Return the paid amount that is actually going to come to store.
 * Returned value here will go into paid amount/deposit of LS POS.
 *
 *
 * @param Cart $cart
 * @return float
 */
public function paid_amount($cart){
    return 0;
}

/**
 * Whether this payment method uses a jumper page or not
 * If it uses a jumper page then process() function must return a HTML FORM string.
 * @return bool
 */
public function uses_jumper(){
```

```

    return true;
}

/**
 *
 * Process payment
 *
 * Return string to be stored as part of the payment to WS if successful (e.g. Reference number)
 * If you are going to do a jumper page, you should return a full HTML FORM that will be execut-
ed in users' browser.
 * Please provide a gateway_response_process() function which should take care of the returned
$_GET or $_POST variables from the third party website
 *
 * Return false if processing has failed. Error can be returned as part of the $errortext vari-
able (ByRef)
 *
 * @param $cart
 * @param $fields
 * @param $errortext
 * @return string|boolean
 */
public function process($cart , $fields , &$errortext){
    global $customer;
    $config = $this->getConfigValues(get_class($this));

    $str = "";

    $str .= "<FORM name=\"_third_party_process\" action=\"https://www.thirdpartysite.com/
payprocess\" method=\"POST\">";
    $str .= _xls_make_hidden('currency_code', _xls_get_conf('CURRENCY_DEFAULT' ,
'USD'));
    $str .= _xls_make_hidden('item_name', "Web store Checkout");
    $str .= _xls_make_hidden('first_name', $customer->Firstname);
    $str .= _xls_make_hidden('last_name', $customer->Lastname);
    $str .= _xls_make_hidden('address1', $customer->Address1);
    $str .= _xls_make_hidden('address2', $customer->Address2);
    $str .= _xls_make_hidden('city', $customer->City);
    $str .= _xls_make_hidden('state', $customer->State);
    $str .= _xls_make_hidden('zip', $customer->Zip);
    $str .= _xls_make_hidden('country', $customer->Country);
    $str .= _xls_make_hidden('email', $customer->Email);
    $str .= _xls_make_hidden('phone1', $customer->Mainphone);
    $str .= _xls_make_hidden('return_url', _xls_site_dir() . "/" . "xls_payment_capture.
php");
    $str .= _xls_make_hidden('amount', round($cart->Total , 2));

```

```

        $str .= (</FORM>');
        return $str;
    }

/**
 *
 * this function processes silent or hosted payment responses
 *
 * Payment methods such as Authorize.net AIM or SIM uses this function to process payment status
in WS.
 *
 * return false if not applicable to you
 * Other wise return an array containing
 * - order_id => Order Id
 * - amount => paid amount
 * - data => payment data to store
 * - success => true| false
 * - output =>
 */
public function gateway_response_process() {
    global $XLSWS_VARS;

$config = $this->getConfigValues(get_class($this));

// means it failed
if(!isset($XLSWS_VARS['transId']) )
return false;

if(!isset($XLSWS_VARS['cartId']))
    return false;
    else{
        $cart = Cart::LoadByIdStr($XLSWS_VARS['cartId']);
$order_id = $cart->IdStr;
    }

if(empty($XLSWS_VARS['transId'])) {
    // failed order
    _xls_log("My Payment modules failed order payment recieved " . print_r($XLSWS_VARS ,
true)) ;
    return false;
}

return array(
    'order_id' => $order_id
    ,
    'amount' => isset($XLSWS_VARS['authAmount'])?$XLSWS_VARS['authAmount']:0

```

```

,      'success' => isset($XLSWS_VARS['transId'])?true:false
,      'data' => isset($XLSWS_VARS['transId'])? $XLSWS_VARS['transId']:''
);

}

}

?>

```



All payment modules must start with `"include_once(XLSWS_INCLUDES . 'payment/xlsws_class_payment.php');"` as they need to extend the `xlsws_class_payment` class

- They must have `name()` and `admin_name()` methods that returns customer front-end module name and backend admin panel name
- `Check()` methods shows the module both admin panel and checkout page. *Example code above checks whether this function is being called from admin panel or not. Because you may want to allow the panel in admin panel for config but have it disabled in checkout page.*
- `Config_fields()` should return an array of fields that are to be used for admin panel configuration, where the string indices of the array are field names. You can use any qcontrol elements for return. But textbox of `XLSTextBox` and List Box `XLSListBox` is preferred. Please note that argument to this function is a `QPanel`, which all your qcontrol elements should use as parent.
- `Check_config_fields()` should validate all the fields the `config_fields()` generated. It's passed parameter is the fields returned by `config_fields()`. The indices are maintained hence you can validate against those.
- WS2.0 admin panel saves chosen config settings. Function like this `$config = $this->getConfigValues(get_class($this));` will return the saved values. It is a returned array which has the config fields keys as indices and there saved values as value.
- `Customer_fields()` and `check_customer_fields()` work in similar fashion as above
- The `process()` function should return a string on successful processing. If payment fails, it should return false. If there is error messages to be displayed, it can be placed in the `$errortext` variable which is a passed by reference.
- Since payment processing often can occur as a jumper (through third party site), `process()` function should return a full HTML FORM in such cases (as per the example above). You should capture the payment return status by `gateway_response_process()` function.
- `Gateway_response_process()` has to work on `$_GET` and `$_POST` variables (or `$XLSWS_VARS`). WS2.0 will go through all payment modules and call the `gateway_response_process()` function on it when there is a page request comes to `xls_payment_capture.php`. Return FALSE unless you are expecting a payment using your module.
- Please note that payment classes are not serialized between requests so if you have member variables, they won't be instantiated in page reload. You should use the stack system (See **Stack Functions**) to maintain variables across page requests.

Sidebar Modules

Sidebar modules work similar to payment and shipping modules except there are no processing functions here. Rather, they are meant to return a QPanel object. Please note that the minicart on Web Store 2.0 is not a sidebar module hence cannot be extended by sidebar modules.

Following is an example for a sidebar module

```
<?php

include_once(XLSWS_INCLUDES . 'sidebar/xlsws_class_sidebar.php');

class my_sidebar extends xlsws_class_sidebar{

    public function name(){
        return _sp("My sidebar");
    }

    /**
     * Return the panel for the sidebar
     *
     * @param $parent the parent to the QPanel
     * @param $id If you want to have a DIV id (auto-generated if none given)
     * @return QPanel
     */
    public function getPanel($parent , $id = null){
        $qp = new QPanel($parent , $id);
        $qp->Template = templateNamed('sidebar_my.tpl.php'); // Define your sidebar's template in the
        templates directory.
        return $qp;
    }
    public function check(){
        return true;
    }

}

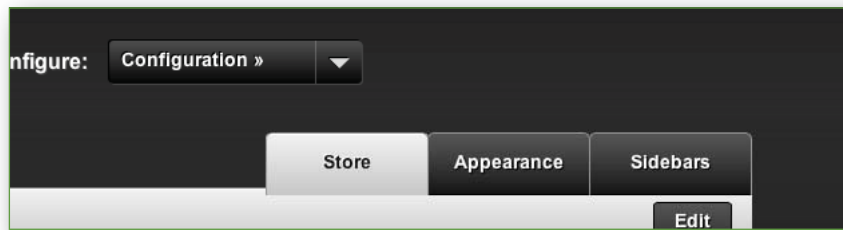
?>
```

Admin Panel

Starting from Web Store 2.0.2, admin panel can be extended.

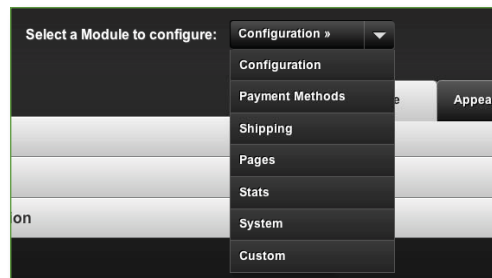
Admin Panel extensions are in the *custom_includes/admin* folder. In this folder, you can create QForm classes where the class name inside the file must match the filename. It must also have a public variable *\$this->strTemplate* which will be used for rendering the QForm with. Define *\$this->arrTabs* for the tabs that will be displayed for jumping in the Custom section. *\$this->arrTabs* contain the QForm class names as key and the label

is value. E.g. for these tabs:



```
$this->arrTabs = array('store' => _sp('Store') , 'appear' => _sp('Appearance') , 'sidebars' =>
_sp('Sidebars'));
```

To create a custom admin panel, create a folder called "admin" in *custom_includes*. If admin panel detects existence of this folder, it will automatically add the Custom drop-down in the *Configure* menu:



Place a .php file in the admin folder where the class you define must have the same filename as the class name. E.g. Following is an admin panel custom module which shows all the gift registries in the Web Store – it must be saved as *my_gift_registry.php*

You can define your template files or use an existing template file from ws distribution. For example *xlsws_admin_generic_edit_form* is a general database table editor class in WS that allows editing of many tables in the admin panel (e.g. countries, states, credit cards, destinations etc..). Our example below uses the edit form and the common edit template file (edit.tpl.php).

```
<?php
class my_gift_registry extends xlsws_admin_generic_edit_form{

    public static $strTemplate = "templates/admin/edit.tpl.php"; // the template file this form
    will be displayed using.

    protected function Form_Create(){

        $this->arrTabs = array('my_gift_registry' => 'Gift Registry'); // Define the tabs
        $this->currentTab = 'my_gift_registry'; // choose yourself in the tabs

        $this->className = "GiftRegistry";
        $this->blankObj = new GiftRegistry();
        $this->qqn = QQN::GiftRegistry(); // the QQN object for query building

        $this->arrFields = array();
```

```
$this->appName = "Gift Registries";

$this->arrFields['RegistryName'] = array('Name' => 'Registry Name');
$this->arrFields['RegistryName']['Field'] = new XLSTextBox($this);
$this->arrFields['RegistryName']['Width'] = 150;
$this->arrFields['RegistryName']['Field']->Required = true;

$this->arrFields['RegistryPassword'] = array('Name' => 'Password');
$this->arrFields['RegistryPassword']['Field'] = new XLSTextBox($this);
$this->arrFields['RegistryPassword']['Field']->Required = true;
$this->arrFields['RegistryPassword']['Width'] = 70;

$this->arrFields['EventDate'] = array('Name' => 'Event Date');
$this->arrFields['EventDate']['Field'] = new XLSTextBox($this);
$this->arrFields['EventDate']['Field']->Required = true;
$this->arrFields['EventDate']['Width'] = 70;

$this->arrFields['CustomerId'] = array('Name' => 'Customer ID');
$this->arrFields['CustomerId']['Field'] = new XLSListBox($this);
$this->arrFields['CustomerId']['Field']->Required = true;
$this->arrFields['CustomerId']['Width'] = 150;
$this->arrFields['CustomerId']['DisplayFunc'] = "RenderCustomer";
$customers = Customer::LoadAll();
foreach($customers as $cust)
    $this->arrFields['CustomerId']['Field']->AddItem( sprintf("%s %s (%s)" , $cust->Firstname , $cust->Lastname , $cust->Email) , $cust->Rowid);

parent::Form_Create();
}

public function canFilter(){
    return true;
}

public function RenderCustomer($val){
    $cust = Customer::Load($val);

    if(!$cust)
        return '';
}
```

```
        return sprintf("%s %s (%s)" , $cust->Firstname , $cust->Lastname , $cust->Email);
    }

}

?>
```

Note that you can write anything in your admin module and display using any template as we you like. HOWEVER, it is recommended that you maintain `$this->strTemplate`, `$this->$arrTabs` and the configure dropdown in your template file for return back to other admin menu items.

Internationalization

Web Store 2.0 supports i18n translation standards. Qcodo's QI18n class takes care of all translation requirements and more about it can be learnt from

<http://examples.qcodo.com/examples/communication/i18n.php>

For any coding development done in Web Store 2.0, it is recommended that you output texts using `_p()` or `_sp()` function. For example if `$str` is a variable that you want to output to the browser, use `_p($str)`. To assign to another variable, use `$var = _sp($str)`.

To setup a new language, follow the steps below:

- Specify language code in administration panel under **Configuration->Store->Localization**.
- Create a file as your language code .po (e.g. if your language code was 'en', then your filename would be en.po) in `includes/qcodo/i18n`. Please note that depending on web server's filesystem, this file may need to be case-sensitive! E.g. by default Web Store 2.0 has EN as language code. Hence if you want to override the default English texts, you need to create a file called `EN.po`.
- For every text that Web Store 2.0 prints other than those printed in templates already, goes through the translation system. In the file created in the above step, you need to create a `msgid` and `msgstr`. E.g. to convert the text "Invalid Zip/Postal Code" to "Invalid Post Code", enter the following two lines :
 - `msgid "Invalid Zip/Postal Code"`
 - `msgstr "Invalid Post Code"`
- Sometimes strings have variables in them. These variables are replaced by `%s` on the fly. E.g.
 - `"Sorry, we cannot ship to %s %s %s"`