Specifics of multistore module development

Specifics of multistore module development

Usage of configuration variables

As indicated in the "Configuration object" section of the "Creating a PrestaShop module" chapter of this developer guide, some of the methods from the Configuration object have three optional parameters, which make it possible to target any other existing store on the current PrestaShop installation: `id_shop`, `id_shop_group` and `id_lang`.

While these parameters can prove useful when having to handle multiple specific and known stores from one of the presented Configuration method, they are not recommended for general usage. Your configuration code should only target the current store using the current Context, which every `Configuration` method does automatically.

In the few occasions where you need to apply a new value to the same `Configuration` variable for all existing stores, using the `Configuration::updateGlobalValue('myVariable', 'myValue')` method. You can then retrieve that value for the current store by using the `Configuration::get('myVariable')` method.

Handling images and other types of files

Images can be defined on a per-store or a per-store-group basis: an item can have a default image on most stores on the PrestaShop installation, and another image for a specific store.

In order to achieve that, the name of the image must be stored in a configuration variable. The default image is then replaced on the fly with the store-specific image.

Let's explore how the Blockadvertising module does it. Using this module, you can upload an image to be displayed on the front-office of the store, serving as an advertisement to any link you might want. The module was built to make use of the multistore feature: depending on the selected store or store group in the multistore menu, the user can assign the uploaded image to a given store context or store group context.

Here is how it saves a submitted file:

Saving the file on a per-store basis

```
{
  // Saving only the main portion of the uploaded file's name, without the file extension.
  Configuration::updateValue(
    'BLOCKADVERT_IMG_EXT',
    substr($_FILES['adv_img']['name'], strrpos($_FILES['adv_img']['name'], '.') + 1)
    );

  // Setting the image's name with a name contextual to the shop context.
  $this->adv_imgname = 'advertising';

  // Creating two versions of the image name, depending on the store context:
  // If the context is the current group, use the image named 'advertising-g'
  // If the context is the current store, use the image named 'advertising-s'
  if (Shop::getContext() == Shop::CONTEXT_GROUP)
    $this->adv_imgname = 'advertising'.'-g'.(int)$this->context->shop->getContextShopGroupID();
  elseif (Shop::getContext() == Shop::CONTEXT_SHOP)
    $this->adv_imgname = 'advertising'.'-s'.(int)$this->context->shop->getContextShopID();

  // Copying the image in the module directory with its new contextual name.
  if (!move_uploaded_file($_FILES['adv_img']['tmp_name'],
    _PS_MODULE_DIR_.$this->name.'/'.$this->adv_imgname.'.'.Configuration::get('BLOCKADVERT_IMG_EXT')))
    $errors .= $this->l('File upload error.');
}
```

And here is how it decides which image to display, depending on the context:

Loading the file on a per-store basis

```
protected function initialize()
{
  // Setting the main name of the image.
  $this->adv_imgname = 'advertising';

  // Setting the contextual name of the file, depending on the context.
  if ((Shop::getContext() == Shop::CONTEXT_GROUP  || Shop::getContext() == Shop::CONTEXT_SHOP)
    && file_exists(_PS_MODULE_DIR_.$this->name.'/'.$this->adv_imgname.'-g'
    .$this->context->shop->getContextShopGroupID().'.'.Configuration::get('BLOCKADVERT_IMG_EXT')))
    $this->adv_imgname .= '-g'.$this->context->shop->getContextShopGroupID();
  if (Shop::getContext() == Shop::CONTEXT_SHOP
    && file_exists(_PS_MODULE_DIR_.$this->name.'/'.$this->adv_imgname.'-s'
    .$this->context->shop->getContextShopID().'.'.Configuration::get('BLOCKADVERT_IMG_EXT')))
    $this->adv_imgname .= '-s'.$this->context->shop->getContextShopID();
  $this->adv_img = Tools::getMediaServer($this->name)._MODULE_DIR_.$this->name.'/'
    .$this->adv_imgname.'.'.Configuration::get('BLOCKADVERT_IMG_EXT');
}
```

Creating database tables

There can be several scenarios:

- The object/entity exists in some stores, but not in all of them: you should use an associative table, and handle it using `id_group_shop`.
- The object/entity exists in all stores: use the regular tables, with a custom field. Handle it using `id_entity`, `id_group_shop` and the attributes.
- The language must be different for every store: handle it using `id_shop` in the `_lang` table.

Object in some stores only

The method to use is `Shop::addSqlRestriction($share = false, $alias = null)`.

It makes it possible to add restrictions to a SQL query, in order to retrieve information on a shop or a shop group.
The parameters are optional:

- `$share`: is used to add a "WHERE" clause indicating that we are in the same group as the current store.
- `$alias`: is used to set the table alias to which the WHERE clause should be applied.

For instance, this query retrieves visitor connections from a given shop:

```
'SELECT date_add, COUNT(`date_add`) as total
 FROM `'._DB_PREFIX_.'connections`
 WHERE 1 '.Shop::addSqlRestriction();
```

Object in all stores

The method to use is `Shop::addSqlAssociation($table, $alias, $inner_join = true, $on = null, $force_not_default = false)`.

It makes it possible to add a JOIN clause to the SQL query, joining a table and its associated table in multistore mode.

For instance, this query automatically associates the `_DB_PREFIX_.'product_shop'` table to the `_DB_PREFIX_.'product'` table using the 'p' alias:

```
'SELECT wp.`id_product`
 FROM `'._DB_PREFIX_.'wishlist_product` wp
 LEFT JOIN `'._DB_PREFIX_.'product` p ON p.`id_product` = wp.`id_product`
 '.Shop::addSqlAssociation('product', 'p');
```

This way, the joint between the information from the product and the current store's context is applied.

Different language in every store

The method to use is `Shop::addSqlRestrictionOnLang($alias = null, $id_shop = null)`.

It makes it possible to add a restriction on `id_shop` for a multishop language table.

For instance, this query automatically associates the `_DB_PREFIX_.'product_lang'` table to the `_DB_PREFIX_.'wishlist_product'` table using the `'pl'` alias:

```
'SELECT wp.`id_product`
 FROM `'._DB_PREFIX_.'wishlist_product` wp
 LEFT JOIN `'._DB_PREFIX_.'product` p ON p.`id_product` = wp.`id_product`
 '.Shop::addSqlAssociation('product', 'p').'
 LEFT JOIN `'._DB_PREFIX_.'product_lang` pl ON pl.`id_product` = wp.`id_product`'.Shop::addSqlRestrictionOnLang
('pl').'
 WHERE pl.`id_lang` = '.(int)($id_lang);
```

This way, the joint between the language information of the product and the current store language is applied.