Adding a configuration page -

Adding a configuration page

Your module can get a "Configure" link in the back office module list, and therefore let the user change some settings. This "Configure" link appears with addition of the `getContent()` method to your main class. This is a standard PrestaShop method: its sole existence sends a message to the back office, saying "there's a configuration page in this module, display the configuration link".

But having a `getContent()` public method in the `MyModule` object does only make the "Configure" link appear; it does not create the configuration page out of nowhere. We are going to explain how to create one, where we will be able to edit the content of the `MYMODULE_NAME` variable that we stored in the `ps_configuration` data table.

The getContent() method

First, here is the complete code for the `getContent()` method:

```
public function getContent()
{
    $output = null;

    if (Tools::isSubmit('submit'.$this->name))
    {
        $my_module_name = strval(Tools::getValue('MYMODULE_NAME'));
        if (!$my_module_name
          || empty($my_module_name)
          || !Validate::isGenericName($my_module_name))
            $output .= $this->displayError($this->l('Invalid Configuration value'));
        else
        {
            Configuration::updateValue('MYMODULE_NAME', $my_module_name);
            $output .= $this->displayConfirmation($this->l('Settings updated'));
        }
    }
    return $output.$this->displayForm();
}
```

The `getContent()`method is the first one to be called when the configuration page is loaded. Therefore, we use it to first update any value that might have been submitted by the form that the configuration page contains.

Here is a line by line explanation:

1. `Tools::isSubmit()` is a PrestaShop-specific method, which checks if the indicated form has been validated.
   In this case, if the configuration form has not yet been validated, the whole `if()` block is skipped and PrestaShop will only use the last line, which displays the configuration with the current values, as generated by the `displayForm()` method.
2. `Tools:getValue()` is a PrestaShop-specific method, which retrieve the content of the `POST` or `GET` array in order to get the value of the specified variable.
   In this case, we retrieve the value of the `MYMODULE_NAME` form variable, turn its value into a text string using the `strval()` method, and stores it in the `$my_module_name` PHP variable.
3. We then check for the existence of actual content in `$my_module_name`, including the use of `Validate::isGenericName()`.
   The `Validate` object contains many data validation methods, among which is `isGenericName()`, a method that helps you keep only strings that are valid PrestaShop names – meaning, a string that does not contain special characters, for short.
4. If any of these checks fail, the configuration will open with an error message, indicating that the form validation failed.
   The `$output` variable, which contains the final rendition of the HTML code that makes the configuration page, thus begins with an error message, created using PrestaShop's `displayError()` method. This method returns the correct HTML code for our need, and since that code is first in `$output`, this means the configuration will open with that message.
5. If all these checks are successful, this means we can store the value in our database.
   As we saw earlier in this tutorial, the `Configuration` object has just the method we need: `updateValue()` will store the new value for `MYMODU`

LE_NAME in the configuration data table.
To that, we add a friendly message to the user, indicating that the value has indeed been saved: we use PrestaShop's displayConfirmation() method to add that message as the first data in the $output variable – and therefore, at the top of the page.

6. Finally, we use the custom displayForm() method (which we are going to create and explain in the next section) in order to add content to $output (whether the form was submitted or not), and return that content to the page.
Note that we could have included the code for displayForm() right within getContent(), but chose to separate the two for readability and separation of concerns.

This form-validation code is nothing new for PHP developers, but uses some of the PrestaShop methods that you will very regularly use.

Displaying the form

The configuration form itself is displayed with the displayForm() method. Here is its code, which we are going to explain after the jump:

```
public function displayForm()
{
    // Get default language
    $default_lang = (int)Configuration::get('PS_LANG_DEFAULT');

    // Init Fields form array
    $fields_form[0]['form'] = array(
        'legend' => array(
            'title' => $this->l('Settings'),
        ),
        'input' => array(
            array(
                'type' => 'text',
                'label' => $this->l('Configuration value'),
                'name' => 'MYMODULE_NAME',
                'size' => 20,
                'required' => true
            )
        ),
        'submit' => array(
            'title' => $this->l('Save'),
            'class' => 'btn btn-default pull-right'
        )
    );

    $helper = new HelperForm();

    // Module, token and currentIndex
    $helper->module = $this;
    $helper->name_controller = $this->name;
    $helper->token = Tools::getAdminTokenLite('AdminModules');
    $helper->currentIndex = AdminController::$currentIndex.'&configure='.$this->name;

    // Language
    $helper->default_form_language = $default_lang;
    $helper->allow_employee_form_lang = $default_lang;

    // Title and toolbar
    $helper->title = $this->displayName;
    $helper->show_toolbar = true;        // false -> remove toolbar
    $helper->toolbar_scroll = true;      // yes - > Toolbar is always visible on the top of the screen.
    $helper->submit_action = 'submit'.$this->name;
    $helper->toolbar_btn = array(
        'save' =>
        array(
            'desc' => $this->l('Save'),
            'href' => AdminController::$currentIndex.'&configure='.$this->name.'&save'.$this->name.
            '&token='.Tools::getAdminTokenLite('AdminModules'),
        ),
        'back' => array(
            'href' => AdminController::$currentIndex.'&token='.Tools::getAdminTokenLite('AdminModules'),
            'desc' => $this->l('Back to list')
        )
    );

    // Load current value
    $helper->fields_value['MYMODULE_NAME'] = Configuration::get('MYMODULE_NAME');

    return $helper->generateForm($fields_form);
}
```

While this might look like a huge block of code for a single value to change, this block actually uses some of PrestaShop's method to make it easier to build forms, most notably the `HelperForm` object.

Diving in displayForm()

Let's run down that method:

1. Using the `Configuration::get()` method, we retrieve the value of the currently chosen language ("`PS_LANG_DEFAULT`"). For security reasons, we cast the variable into an integer using `(int)`.
2. In preparation for the generation of the form, we must build an array of the various titles, textfields and other form specifics.
   To that end, we create the `$fields_form` variable, which will contain a multidimensional array. Each of the arrays it features contains the detailed description of the tags the form must contain. From this variable, PrestaShop will render the HTML form as it is described.
   In this example, we define three tags (`<legend>`, `<input>` and `<submit>`) and their attributes using arrays. The format is quite easy to get: the legend and submit arrays simply contain the attributes to each tag, while the input contains as many `<input>` tags are needed, each being in turn an array which contains the necessary attributes. For instance:

   ```
   'input' => array(
       array(
           'type' => 'text',
           'label' => $this->l('Configuration value'),
           'name' => 'MYMODULE_NAME',
           'size' => 20,
           'required' => true
       ))
   ```

   ...generates the following HTML tags:

   ```
   <label>Configuration value </label>
   <div class="margin-form">
     <input id="MYMODULE_NAME" class="" type="text" size="20" value="my friend" name="MYMODULE_NAME">
     <sup>*</sup>
   <div class="clear"></div>
   ```

   As you can see, PrestaShop is quite clever, and generates all the code that is needed to obtain a useful form.
   Note that the value of the main array is actually retrieved later in the form generation code.
3. We then create an instance of the `HelperForm` class. This section of the code is explained in the next section of this chapter.
4. Once the `HelperForm` settings are all in place, we generate the form based on the content of the `$fields_form` variable.

Using HelperForm

`HelperForm` is one of the helper methods that were added with PrestaShop 1.5, along with `HelperOptions`, `HelperList`, `HelperView` and `HelperHelpAccess`. They enable you to generate standard HTML elements for the back office as well as for module configuration pages.
You can get more information about Helper classes in the "Helpers" chapter of this developer guide, with a page dedicated to `HelperForm`.

Here is our sample code, as a reminder:

```
$helper = new HelperForm();

// Module, Token and currentIndex
$helper->module = $this;
$helper->name_controller = $this->name;
$helper->token = Tools::getAdminTokenLite('AdminModules');
$helper->currentIndex = AdminController::$currentIndex.'&configure='.$this->name;

// Language
$helper->default_form_language = $default_lang;
$helper->allow_employee_form_lang = $default_lang;

// title and Toolbar
$helper->title = $this->displayName;
$helper->show_toolbar = true;        // false -> remove toolbar
$helper->toolbar_scroll = true;      // yes - > Toolbar is always visible on the top of the screen.
$helper->submit_action = 'submit'.$this->name;
$helper->toolbar_btn = array(
    'save' =>
    array(
        'desc' => $this->l('Save'),
        'href' => AdminController::$currentIndex.'&configure='.$this->name.'&save'.$this->name.
        '&token='.Tools::getAdminTokenLite('AdminModules'),
    ),
    'back' => array(
        'href' => AdminController::$currentIndex.'&token='.Tools::getAdminTokenLite('AdminModules'),
        'desc' => $this->l('Back to list')
   )
);

// Load current value
$helper->fields_value['MYMODULE_NAME'] = Configuration::get('MYMODULE_NAME');

return $helper->generateForm($fields_form);
```

Our example uses several of `HelperForm`'s attributes: they need to be set before we generate the form itself from the `$fields_form` variable:

- `$helper->module`: requires the instance of the module that will use the form.
- `$helper->name_controller`: requires the name of the module.
- `$helper->token`: requires a unique token for the module. `getAdminTokenLite()` helps us generate one.
- `$helper->currentIndex`:
- `$helper->default_form_language`: requires the default language for the shop.
- `$helper->allow_employee_form_lang`: requires the default language for the shop.
- `$helper->title`: requires the title for the form.
- `$helper->show_toolbar`: requires a boolean value – whether the toolbar is displayed or not.
- `$helper->toolbar_scroll`: requires a boolean value – whether the toolbar is always visible when scrolling or not.
- `$helper->submit_action`: requires the action attribute for the form's `<submit>` tag.
- `$helper->toolbar_btn`: requires the buttons that are displayed in the toolbar. In our example, the "Save" button and the "Back" button.
- `$helper->fields_value[]`: this is where we can define the value of the named tag.

Finally, after all is set and done, we can call the `generateForm()` method, which will take care of putting it all together and, as its name says, generate the form that the user will use to configure the module's settings.

Here is the rendition of the form as it is presently written – which you can see by yourself by clicking on the "Configure" link for the module in the back office:

Change the value to whichever you like, click on the "Save" button, then go reload the homepage: your module is indeed updated with the new string!