

Create your WEB Application with MVC paradigm and OPP – PAP support

Tutorial 16

Okay, now, you should have everything work well, if not, please, see again the last tutorial before you continuous.

OK, now you must prepare the functions to send data to the controller.

Let's made some updates to all functions that you construct at this moment.

Please update the function above the send_data() function:

```
// Function to get data from the form
function get_data() {
  let category_input = document.querySelector("#category-name");
  if (category_input.value.trim() === "" || isNaN(category_input.value.trim())) {
    Swal.fire({
      icon: 'error',
      title: 'Erro',
      text: 'Por favor insira um nome de categoria válido!'
    });
    return;
  }

  var data = category_input.value.trim();

  // Force to create an object
  // You can add more datafields if you want it
  send_data({
    data: data,
    data_type: 'add_category' // expressive with the expression
  });
}
```

Let's debug this code.

As you see before, the Input field are selected with specific selector and locates the input field in the form with the ID category-name.

Then it validates the input data. Checks if the field is empty or contains only numbers. If the data is invalid, it displays an alert and stops execution.

Next gets and cleans the input data. It captures the value of the input field, removing any leading or trailing whitespace.

Finally creates a data object and sends it. This object is then passed to the send_data function().

Please update also your send_data function to handle the responses:

```
// Function to send data to the server
function send_data(data = {}) {
    var ajax = new XMLHttpRequest();

    // Handler for AJAX response
    ajax.addEventListener('readystatechange', function() {
        if (ajax.readyState === 4 && ajax.status === 200) {
            handle_result(ajax.responseText);
        }
    });

    // Set request headers
    ajax.open("POST", "<?=ROOT?>admin/ajax", true);
    ajax.setRequestHeader("Content-Type", "application/json");

    // Send AJAX request
    ajax.send(JSON.stringify(data));
}
```

Finally, update the function handle_result:

```
// Function to handle the result
function handle_result(result) {
    // JSON data from the response
    if (result != "") {
        var obj = JSON.parse(result);

        if (obj.message_type === "info") {
            Swal.fire({
                icon: 'success',
                title: 'Sucesso',
                text: obj.message
            }).then(() => {
                // Close the modal after the alert is dismissed
                $('#categoryModal').modal('hide');
                // Optionally, clear the input field after successful submission
                document.querySelector("#category-name").value = "";
            });
        } else {
            Swal.fire({
                icon: 'error',
                title: 'Erro',
                text: obj.message
            });
        }
    }
}
```

Okay, this function will handle the responses from your controller. Imagine if something goes wrong; the user needs to know what the issue is. So, this function will do that job.

From the controller, you will receive a `message_type` with the information. If the type is `info`, it means that everything is okay, the data has been successfully added to the database, and the modal is closed. If not, you have an error message indicating what is wrong. The messages come from the model, as you will see next.



For the alert messages I recommend you use the sweet alert. If you don't know this technology, please make some research on web.

To use this feature, you need to add a new library to your header like the example:

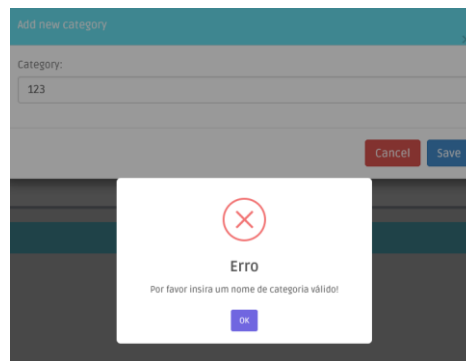
```
<!-- Sweet Alert 2 -->
<!-- Adicione isso no <head> do seu documento HTML -->
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/sweetalert2@11/dist/sweetalert2.min.css">
```

And the JavaScript lib to your footer inside the body, as the example:

```
<!-- Sweet Alert 2 -->
<!-- Adicione isso antes do fechamento do </body> -->
<script src="https://cdn.jsdelivr.net/npm/sweetalert2@11"></script>

</body>
</html>
```

Now the result when you have a error should be like this:



It's more beautiful, instead of the traditional console alert from your browser.

Ok, now you go to your controller and prepare to receive the data from your front side template.

Please update your index ajax code:

```
class Ajax extends Controller
{
    //Public default metodo index, mesmo que o utilizador coloque ou não qualquer URL, o Index va

    public function index()
    {
        //Obtenha os dados enviados pelo cliente
        $data = file_get_contents("php://input");
        //print_r($data);
        $data = json_decode($data);

        //show($data);

        if (is_object($data) && isset($data->data_type) && $data->data_type == 'add_category') {
            // Carrega o modelo de categoria
            $category = $this->load_model('Category');
            $check = $category->create($data);

            if (!empty($_SESSION['error'])) {
                $arr['message'] = $_SESSION['error'];
                $_SESSION['error'] = "";
                $arr['message_type'] = "error";
            } else {
                $arr['message'] = "Categoria adicionada com sucesso!";
                $arr['message_type'] = "info";
            }

            $arr['data'] = "";
            echo json_encode($arr);
        }
    }
}
```

The index function in your controller has been updated to handle various tasks related to adding a new category, including processing incoming JSON data, managing validation errors, and providing appropriate responses.

Upon receiving data from the client, the function decodes the JSON payload into a PHP object. This data is essential for determining the type of operation requested by the client.

The function verifies that the received data contains a specific `data_type` property set to `add_category`. This step ensures that the function only proceeds with adding a new category if the correct operation is requested.

If the data verification is successful, the function proceeds to load the category model. This model is responsible for interacting with the database and handling category-related operations.

With the category model loaded, the function calls the create method on the model, passing along the received data. The create method performs the necessary database operations to add the new category.

During the category creation process, if any errors occur, they are stored in the session variable `$_SESSION['error']`. The function checks for the presence of errors and prepares an appropriate error message if necessary. If no errors are encountered, a success message is prepared instead.

Once the message (whether it's an error or success) is determined, the function constructs a response in the form of a JSON-encoded object. This response contains the message, along with a message type indicator. This information is crucial for the client-side script to display an appropriate alert to the user, informing them of the outcome of the category addition process.

With the index function now handling all these tasks efficiently, your application is equipped to seamlessly add new categories with proper validation and error handling mechanisms in place.

Next, you can proceed to update the create method in your category model to ensure it aligns with the functionality implemented in the index function.

Now, you can update your create model.

```
public function create($data)
{
    $DB = Database::getInstance();

    // Verifica se os dados são válidos antes de inserir
    if (!empty($data->data) && $data->data_type == 'add_category') {
        // Prepare os dados para inserção
        $category = ucwords(trim($data->data));

        // Verifica se o nome da categoria é válido
        if (!preg_match("/^[a-zA-Z]+$/", $category)) {
            $_SESSION['error'] = "Por favor insira um nome de categoria correto!";
            return false;
        }

        // Construa a consulta SQL para inserção
        $query = "INSERT INTO categories (category) VALUES (:category)";
        $params = array(':category' => $category);
        // Executa a consulta SQL
        $check = $DB->write($query, $params);
        if ($check) {
            return true;
        } else {
            // Se houver um erro ao inserir, defina uma mensagem de erro na sessão
            $_SESSION['error'] = "Erro ao inserir categoria no banco de dados.";
        }
    } else {
        $_SESSION['error'] = "Dados inválidos para adicionar categoria.";
    }
    return false;
}
```

Let's debug:

The create method within your category model is responsible for inserting new category data into the database. Let's break down how it accomplishes this task:

Data Validation: Before attempting to insert data into the database, the method verifies that the provided data is valid and that the operation type is set to add a new category (add_category). This ensures that only legitimate data is processed further.

Data Preparation: Once the data is validated, the method prepares the category name for insertion. It trims any leading or trailing whitespace from the provided category name and converts it to uppercase using the ucwords function.

Category Name Validation: The method performs additional validation on the category name to ensure it consists solely of alphabetic characters. It utilizes a regular expression (preg_match) to check that the category name contains only letters. If the name is found to be invalid, an appropriate error message is set in the session, and the method returns false to indicate failure.

SQL Query Construction: If the category name passes validation, the method constructs an SQL query to insert the category into the database. The query is parameterized to prevent SQL injection vulnerabilities, with the category name provided as a parameter.

Executing the SQL Query: The method executes the SQL query using the database instance (\$DB->write) and passes the query parameters. If the insertion is successful, the method returns true to indicate success. Otherwise, if an error occurs during insertion, an error message is set in the session, indicating the failure to insert the category into the database.

Handling Invalid Data: If the provided data is determined to be invalid or if the data type is incorrect, appropriate error messages are set in the session to inform the user of the issue.

Returning Success or Failure: Finally, the method returns false to indicate failure if any validation checks fail or if the insertion operation encounters an error. Otherwise, it returns true to signify successful insertion.

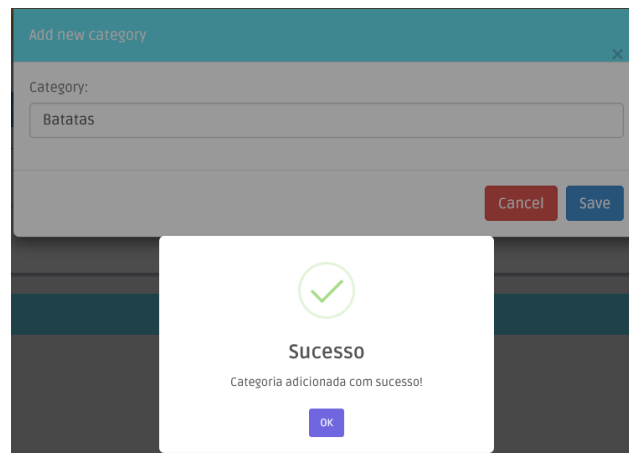
With these mechanisms in place, the create method ensures that only valid category data is inserted into the database, with appropriate error handling in case of any issues.

Ok, now, before your test, please also update the button as the picture:

```
<div class="modal-footer">
  <button type="button" class="btn btn-danger" data-dismiss="modal">Cancel</button>
  <button type="button" class="btn btn-primary" onclick="get_data()">Save</button>
</div>
```




Now go to your modal, and try to insert something in the modal box.

Result expected:



If you have any issue, the error will pop up.

Ok, at this moment, everything is working well, and you can check your database to validate the insert data:

					id	category	disabled
<input type="checkbox"/>		Editar		Copiar		Apagar	20 Batatas 0
<input type="checkbox"/>		Editar		Copiar		Apagar	21 Rebocar 0
<input type="checkbox"/>		Editar		Copiar		Apagar	22 Pneus 0
<input type="checkbox"/>		Editar		Copiar		Apagar	23 Batata 0
<input type="checkbox"/>		Editar		Copiar		Apagar	24 Batatas 0

Next tutorial you will configure the table to show the data.

Remember, always comment your code, to facilitate in the future to read them.

END Tutorial 16

Reference: PHP Ecommerce website development | Send data to database | MVC OOP - Quick programming

Template: NICEADMIN, design by BOOTSTRAP

Modal: <https://getbootstrap.com/docs/4.0/components/modal/>