Routing system

Previously, `index.php` directly included the module file to process the action, based on the `action` and `file` request parameters.

With the new architecture, the resolution logic has been moved to the **`IndexRouter`** class, which handles action path resolution and processing.

Action resolution

Searched paths (in order)

When an action is requested, `IndexRouter` searches for the file in the following paths:

1. `modules/{MODULE}/{ACTION}.php`
2. `modules/{MODULE}/{ACTION}/{ACTION}.php`
3. `modules/{MODULE}/controllers/{ACTION}.php` **(NEW)**
4. `modules/VteCore/{ACTION}.php`
5. `modules/VteCore/controllers/{ACTION}.php` **(NEW)**

Removal of `{MODULE}Ajax.php`

It is no longer necessary to create a `{MODULE}Ajax.php` file in the module folder. AJAX requests can be handled directly by BaseAction classes, which automatically detect the request type and desired output format.

Execution process

After resolving and including the action file:

1. `IndexRouter` checks if the file contains a class that extends `BaseAction`
2. If found, it calls the `fromRequest()` method to create an instance of the class
3. During `fromRequest()`, class properties are automatically populated from request parameters
4. `processFromIndex()` is called, which performs validation and processing
5. The result is formatted according to the requested content-type and sent to the client

If the file included in `index.php` does not contain a `BaseAction` class, then the included code is executed and entirely managed by the file (previous behavior).

Creating a new BaseAction

Basic structure

To create a new action, create a file in:

- `modules/{MODULE}/controllers/{ACTION}.php`
- or `modules/VteCore/controllers/{ACTION}.php` (for global actions)

The class must:

- Extend `BaseAction`
- Implement the `process()` method
- Declare properties with appropriate attributes

```
<?php

class MyAction extends BaseAction {

    protected function process() {
        // Your logic here
        return ['success' => true, 'data' => []];
    }
}
```

Declaring properties

Class properties can be automatically populated from the request using PHP 8 attributes:

Attribute	Source	Description
`RequestParam`	$_REQUEST	Parameter from request (GET or POST)
`RequestRawParam`	$_REQUEST	Raw parameter from request
`GetParam`	$_GET	Parameter from GET
`GetRawParam`	$_GET	Raw parameter from GET
`PostParam`	$_POST	Parameter from POST
`PostRawParam`	$_POST	Raw parameter from POST
`CookieParam`	$_COOKIE	Parameter from cookie

Attribute parameters

- **name**: Parameter name in the request (default: property name)
- **filter**: Filter to apply (e.g. `F::int`, `F::str`, `F::bool`). For more details, see Basic filters.

- **required**: Whether the parameter is required (default: based on nullable type)
- **default**: Default value if the parameter is not present
- **validation**: Regex or callable to validate the value
- **description**: Parameter description (for documentation)

```php
<?php
class MyAction extends BaseAction {

    // Required parameter (not nullable)
    #[RequestParam()]
    protected int $record;

    // Optional parameter (nullable)
    #[RequestParam()]
    protected ?string $searchText;

    // Parameter with different name
    #[RequestParam(name: 'return_module', filter: F::mod)]
    protected ?string $returnModule;

    // Parameter with regex validation
    #[RequestParam(validation: '/^(true|false)$/')]
    protected ?string $isDuplicate;

    // Parameter with callable validation
    #[RequestParam(validation: 'is_numeric')]
    protected ?string $amount;

    // Parameter with default value
    #[RequestParam(default: 10)]
    protected int $limit;

    // Parameter from POST
    #[PostParam()]
    protected ?string $comment;

    // Parameter from GET
    #[GetParam()]
    protected ?int $page;
```

```
    protected function process() {
        // Properties are already populated here
        return [
            'record' => $this->record,
            'search' => $this->searchText,
        ];
    }
}
```

Main methods

process()

Main method that contains the action logic. Must return data that will be formatted and sent to the client.

```
protected abstract function process();
```

validate(?string &$error): bool

Called before `process()` to validate the request. If it returns `false`, execution stops and an error is sent.

```
protected function validate(?string &$error): bool {
    if ($this->record <= 0) {
        $error = "Invalid record ID";
        return false;
    }
    return true;
}
```

beforeProcess()

Called before `process()`, after validation.

```
protected function beforeProcess(): void {
    // Initialization, logging, etc.
}
```

afterProcess(&$result)

Called after `process()`, allows modifying the result before sending.

```
protected function afterProcess(&$result): void {
    // Modify result, logging, etc.
    $result['timestamp'] = time();
}
```

Output

The output format is automatically determined based on:

1. The class's `$outputFormat` property (if set)
2. The `format` or `_format` parameter in the request
3. The `Accept` header of the request
4. Whether the request is AJAX (default: `json`) or not (default: `html`)

Supported formats

- **json** - JSON response (automatic for AJAX)
- **html** - HTML response (automatic for standard requests)

You can force a specific format by setting the `$outputFormat` property:

```
class MyAction extends BaseAction {

    protected ?string $outputFormat = 'json';

    protected function process() {
        return ['data' => 'Always JSON'];
    }
}
```

Or by using the `setOutputFormat()` method:

```
class MyAction extends BaseAction {

    protected function beforeProcess(): void {
        $this->setOutputFormat('html');
    }

    protected function process() {
        return '<h1>HTML Output</h1>';
    }
}
```

JSON response format

On success:

```
{
    "success": true,
    "result": { /* data returned by process() */ }
}
```

On error:

```
{
    "success": false,
    "error": {
        "message": "Error message",
        "code": 400
    }
}
```

Requesting JSON from client

To request a JSON response, the client can:

- Add the parameter `?format=json` or `?_format=json` to the URL
- Set the header `Accept: application/json`
- Make an AJAX request (automatically detected)

Example

```
<?php

use RequestParam;
use PostParam;

class SaveData extends BaseAction {

    // We don't specify outputFormat, it will be determined automatically

    #[RequestParam()]
    protected ?int $record;

    #[PostParam(required: true)]
```

```
    protected string $name;

    #[PostParam(filter: F::int, default: 1)]
    protected int $status;

    #[PostParam()]
    protected ?string $description;

    protected function validate(?string &$error): bool {
        if (strlen($this->name) < 3) {
            $error = "Name must be at least 3 characters long";
            return false;
        }
        return true;
    }

    protected function beforeProcess(): void {
        // Log the operation
        global $log;
        $log->info("Saving record: " . $this->name);
    }

    protected function process() {
        global $adb;

        // Save logic
        if ($this->record) {
            // Update logic
        } else {
            // Insert logic
        }

        // Return data
        // If the request is AJAX, it will be automatically formatted as JSON
        // Otherwise as HTML
        return [
            'record_id' => $id,
            'message' => 'Save completed'
        ];
    }
```

```
    protected function afterProcess(&$result): void {
        // Add timestamp to result
        $result['timestamp'] = time();
    }
}
```

Differences from the previous version

Aspect	Before	Now
Action resolution	In index.php	In IndexRouter.php
Ajax files	`{MODULE}Ajax.php` required	No longer necessary
Controllers paths	Did not exist	`modules/{MODULE}/controllers/` `modules/VteCore/controllers/`
Parameter population	Manual with $_REQUEST	Automatic with PHP attributes
Validation	Manual	validate() method + attribute validation
Output format	Manually handled	Automatic
Error handling	Manual	Automatic with try/catch in processFromIndex()

Best Practices

- Create new actions as BaseAction classes in `modules/{MODULE}/controllers/`
- Use attributes to declare parameters explicitly
- Always specify the appropriate filter (`F::int`, `F::str`, etc.)
- Implement the `validate()` method for business logic validations
- Use attribute validation for simple validations (regex, callable)
- Let the system automatically determine the output format (JSON/HTML)
- Force `$outputFormat` only when necessary
- Use `beforeProcess()` for initialization and `afterProcess()` for post-processing

Revision #4
Created 2026-01-27 09:20:42 UTC by manuel.tagliapietra
Updated 2026-01-27 10:21:44 UTC by manuel.tagliapietra