API Reference

Base Module Class Methods

Core Methods

`__construct()`

Initializes the module instance. Always call `parent::__construct()` first.

```
public function __construct()
{
    parent::__construct();
    // Custom initialization logic here
}
```

Lifecycle Methods

`activate(): string`

Called when module is activated. Should set up database tables, initial configuration.

Returns: `'success'` or error message

```
public function activate(): string
{
    try {
        Schema::create('module_table', function (Blueprint $table) {
            $table->id();
            $table->uuid('service_uuid');
            $table->string('external_id')->nullable();
            $table->enum('status', ['active', 'suspended', 'terminated']);
            $table->timestamps();

            $table->foreign('service_uuid')->references('uuid')->on('services');
            $table->index(['service_uuid', 'status']);
        });
```

```
        $this->logInfo('activate', 'Module activated successfully');
        return 'success';
    } catch (Exception $e) {
        $this->logError('activate', 'Activation failed: ' . $e->getMessage());
        return 'Error: ' . $e->getMessage();
    }
}
```

`deactivate(): string`

Called when module is deactivated. Should clean up resources.

Returns: `'success'` or error message

```
public function deactivate(): string
{
    try {
        Schema::dropIfExists('module_table');
        $this->logInfo('deactivate', 'Module deactivated successfully');
        return 'success';
    } catch (Exception $e) {
        $this->logError('deactivate', 'Deactivation failed: ' . $e->getMessage());
        return 'Error: ' . $e->getMessage();
    }
}
```

`update(): string`

Called when module version changes. Handle data migrations here.

Returns: `'success'` or error message

```
public function update(): string
{
    try {
        $currentVersion = $this->getCurrentVersion();

        if (version_compare($currentVersion, '1.1.0', '<')) {
            Schema::table('module_table', function (Blueprint $table) {
                $table->string('new_field')->nullable();
            });
```

```
        }

        return 'success';
    } catch (Exception $e) {
        return 'Error: ' . $e->getMessage();
    }
}
```

Product Module Methods

Configuration Methods

`getProductData(array $data = []): array`

Processes and stores product configuration data.

Parameters:

- `$data` - Array of configuration values

Returns: Processed configuration array

```
public function getProductData(array $data = []): array
{
    $this->product_data = [
        'server_location' => $data['server_location'] ?? '',
        'plan_type' => $data['plan_type'] ?? '',
        'disk_space' => $data['disk_space'] ?? '',
        'bandwidth' => $data['bandwidth'] ?? '',
    ];
    return $this->product_data;
}
```

`saveProductData(array $data = []): array`

Validates and saves product configuration.

Parameters:

- `$data` - Configuration data to validate

Returns:

```
[
    'status' => 'success|error',
    'message' => 'Error messages if validation fails',
    'code' => 200|422,
    'data' => $validatedData
]
```

Example:

```
public function saveProductData(array $data = []): array
{
    $validator = Validator::make($data, [
        'server_location' => 'required|string',
        'plan_type' => 'required|in:basic,premium,enterprise',
        'disk_space' => 'required|integer|min:1',
        'bandwidth' => 'required|integer|min:1',
    ]);

    if ($validator->fails()) {
        return [
            'status' => 'error',
            'message' => $validator->errors(),
            'code' => 422,
        ];
    }

    return [
        'status' => 'success',
        'data' => $data,
        'code' => 200,
    ];
}
```

`getProductPage(): string`

Returns HTML for product configuration page in admin area.

Returns: Rendered HTML string

Notes: Should use `view()` to render Blade templates and pass validated configuration.

```
public function getProductPage(): string
{
    return $this->view('admin_area.product', [
        'config' => $this->product_data,
        'validation' => [/* rules, hints */],
    ]);
}
```

Service Methods

`getServiceData(array $data = []): array`

Processes service instance data.

Parameters:

- `$data` - Service configuration array

Returns: Processed service data

Example:

```
public function getServiceData(array $data = []): array
{
    $this->service_data = [
        'domain' => strtolower(trim($data['domain'] ?? '')),
        'username' => trim($data['username'] ?? ''),
        'notes' => $data['notes'] ?? null,
    ];
    return $this->service_data;
}
```

`saveServiceData(array $data = []): array`

Validates and saves service configuration.

Returns: Same format as `saveProductData()`

Example:

```
public function saveServiceData(array $data = []): array
{
```

```
    $validator = Validator::make($data, [
        'domain' => 'required|regex:/^[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]*\.[a-zA-Z]{2,}$/',
        'username' => 'required|alpha_dash|min:3|max:20',
    ]);

    if ($validator->fails()) {
        return [
            'status' => 'error',
            'message' => $validator->errors(),
            'code' => 422,
        ];
    }

    return [
        'status' => 'success',
        'data' => $data,
        'code' => 200,
    ];
}
```

`getServicePage(): string`

Returns HTML for service configuration page.

Returns: Rendered HTML string

Service Lifecycle Methods

`create(): array`

Initiates service creation process (usually queues a job).

Returns:

```
['status' => 'success|error', 'message' => 'Optional message']
```

Example:

```
public function create(): array
{
    $data = [
```

```
        'module' => $this,              // Module instance reference
        'method' => 'createJob',        // Method to execute
        'tries' => 1,                   // Retry attempts
        'backoff' => 60,                // Delay between retries (seconds)
        'timeout' => 600,               // Max execution time
        'maxExceptions' => 1,           // Max exceptions before failure
    ];

    $service = Service::find($this->service_uuid);
    $service->setProvisionStatus('processing');
    Task::add('ModuleJob', 'Module', $data, ['create']);
    return ['status' => 'success'];
}
```

`createJob(): array`

Actual service creation logic executed asynchronously.

Returns: Status array

Example:

```
public function createJob(): array
{
    try {
        $service = Service::find($this->service_uuid);
        $service->setProvisionStatus('processing');

        // Your service creation logic here
        // ...

        $service->setProvisionStatus('completed');
        $this->logInfo('createJob', 'Service created successfully');
        return ['status' => 'success'];

    } catch (Exception $e) {
        $service->setProvisionStatus('failed');
        $this->logError('createJob', 'Service creation failed: ' . $e->getMessage());
        return ['status' => 'error', 'message' => 'Service creation failed'];
    }
}
```

`suspend(): array`

Initiates service suspension.

```
public function suspend(): array
{
    $service = Service::find($this->service_uuid);
    $service->setProvisionStatus('processing');

    Task::add('ModuleJob', 'Module', [
        'module' => $this,
        'method' => 'suspendJob',
        'tries' => 1,
        'backoff' => 60,
        'timeout' => 600,
        'maxExceptions' => 1,
    ], ['suspend', 'service:' . $this->service_uuid]);

    return ['status' => 'success'];
}
```

`suspendJob(): array`

Actual suspension logic.

```
public function suspendJob(): array
{
    try {
        $api = new ExternalAPI($this->getServerConfig());
        $result = $api->suspend($this->service_data);

        if ($result['status'] === 'success') {
            Service::find($this->service_uuid)->setProvisionStatus('suspended');
            $this->logInfo('suspendJob', 'Service suspended');
            return ['status' => 'success'];
        }

        Service::find($this->service_uuid)->setProvisionStatus('failed');
        return ['status' => 'error', 'message' => $result['error'] ?? 'Unknown error'];
    } catch (Exception $e) {
        Service::find($this->service_uuid)->setProvisionStatus('failed');
        $this->logError('suspendJob', 'Suspension failed', $e->getMessage());
        return ['status' => 'error', 'message' => 'Suspension failed'];
    }
```

```
}
```

unsuspend(): array

Initiates service reactivation.

```
public function unsuspend(): array
{
    Task::add('ModuleJob', 'Module', [
        'module' => $this,
        'method' => 'unsuspendJob',
        'tries' => 1,
        'backoff' => 60,
        'timeout' => 600,
        'maxExceptions' => 1,
    ], ['unsuspend', 'service:' . $this->service_uuid]);

    return ['status' => 'success'];
}
```

unsuspendJob(): array

Actual reactivation logic.

```
public function unsuspendJob(): array
{
    try {
        $api = new ExternalAPI($this->getServerConfig());
        $result = $api->unsuspend($this->service_data);
        if ($result['status'] === 'success') {
            Service::find($this->service_uuid)->setProvisionStatus('completed');
            return ['status' => 'success'];
        }
        return ['status' => 'error', 'message' => $result['error'] ?? 'Unknown error'];
    } catch (Exception $e) {
        $this->logError('unsuspendJob', 'Reactivation failed', $e->getMessage());
        return ['status' => 'error'];
    }
}
```

termination(): array

Initiates service termination.

```
public function termination(): array
{
    Task::add('ModuleJob', 'Module', [
        'module' => $this,
        'method' => 'terminationJob',
        'tries' => 1,
        'backoff' => 60,
        'timeout' => 600,
        'maxExceptions' => 1,
    ], ['terminate', 'service:' . $this->service_uuid]);
    return ['status' => 'success'];
}
```

`terminationJob(): array`

Actual termination logic.

```
public function terminationJob(): array
{
    try {
        $api = new ExternalAPI($this->getServerConfig());
        $result = $api->terminate($this->service_data);
        if ($result['status'] === 'success') {
            Service::find($this->service_uuid)->setProvisionStatus('terminated');
            return ['status' => 'success'];
        }
        return ['status' => 'error', 'message' => $result['error'] ?? 'Unknown error'];
    } catch (Exception $e) {
        $this->logError('terminationJob', 'Termination failed', $e->getMessage());
        return ['status' => 'error'];
    }
}
```

`change_package(): array`

Initiates package/plan change.

```
public function change_package(): array
{
```

```
    Task::add('ModuleJob', 'Module', [
        'module' => $this,
        'method' => 'change_packageJob',
        'tries' => 1,
        'backoff' => 60,
        'timeout' => 600,
        'maxExceptions' => 1,
    ], ['change_package', 'service:' . $this->service_uuid]);
    return ['status' => 'success'];
}
```

`change_packageJob(): array`

Actual package change logic.

```
public function change_packageJob(): array
{
    try {
        $api = new ExternalAPI($this->getServerConfig());
        $result = $api->changePackage([
            'plan' => $this->service_data['plan'] ?? 'basic',
            'bandwidth' => $this->product_data['bandwidth'] ?? null,
        ]);
        if ($result['status'] === 'success') {
            $this->logInfo('change_packageJob', 'Plan changed');
            return ['status' => 'success'];
        }
        return ['status' => 'error', 'message' => $result['error'] ?? 'Unknown error'];
    } catch (Exception $e) {
        $this->logError('change_packageJob', 'Change failed', $e->getMessage());
        return ['status' => 'error'];
    }
}
```

Plugin Module Methods

Lifecycle

`activate(): string` — create required tables, seed defaults. Must be idempotent.

`deactivate(): string` — drop or archive resources safely.

`update(): string` — migrate schema/config between versions.

Admin Interface

`adminSidebar(): array` — sidebar entries. See format in Admin Interface Methods.

`adminWebRoutes(): array` — web routes with permissions.

`adminApiRoutes(): array` — API routes for AJAX.

```
public function adminWebRoutes(): array
{
    return [
        [
            'method' => 'get',
            'uri' => 'dashboard',
            'permission' => 'monitoring-dashboard',
            'name' => 'dashboard',
            'controller' => 'MonitoringController@dashboard',
        ],
    ];
}
```

Background Work

Schedule jobs via `Task::add()` and listen to events in `hooks.php`.

```
Task::add('ModuleJob', 'Module', [
    'module' => $this,
    'method' => 'collectMetrics',
], ['metrics']);
```

Payment Module Methods

Configuration

`getModuleData(array $data = []): array` — normalize gateway settings.

Parameters: gateway keys, secrets, webhook secrets, sandbox flag.

Returns: sanitized config array.

Client UI

`getClientAreaHtml(array $data = []): string` — render payment form/session.

Parameters: `$data['invoice']`, optional customer/context.

Returns: rendered HTML.

```
public function getClientAreaHtml(array $data = []): string
{
    $invoice = $data['invoice'];
    $session = (new StripeClient($this->module_data))->createSession(
        referenceId: $invoice->uuid,
        invoiceId: $invoice->number,
        description: 'Invoice #' . $invoice->number,
        amount: $invoice->getDueAmountAttribute(),
        currency: $invoice->client->currency->code,
        return_url: $this->getReturnUrl(),
        cancel_url: $this->getCancelUrl(),
    );
    return $this->view('client_area', ['session' => $session]);
}
```

Settings Page

`getSettingsPage(array $data = []): string` — render admin configuration; should provide generated `webhook_url`.

Webhooks

`apiWebhookPost(Request $request): Response` — process gateway callbacks (POST).

`apiWebhookGet(Request $request): Response` — optional GET verification.

```
public function apiWebhookPost(Request $request): Response
{
    $payload = $request->all();
    // Verify signature and handle events
    switch ($payload['type'] ?? '') {
```

```
        case 'payment_intent.succeeded':
            return $this->onPaymentSuccess($payload);
        case 'payment_intent.payment_failed':
            return $this->onPaymentFailed($payload);
        default:
            return response('ok', 200);
    }
}
```

Payment Actions

`onPaymentSuccess(array $payload): Response|array` — mark invoice paid, log transaction.

`onPaymentFailed(array $payload): Response|array` — record failure.

`refund(string $transactionId, int|float $amount): array` — optional refund handler.

Notification Module Methods

Configuration

`getModuleData(array $data = []): array` — server, port, encryption, credentials, sender.

Delivery

`send(array $data = []): array` — send message through the channel.

Parameters:

- `to` (string) — recipient address
- `subject` (string) — subject/title
- `message` (string) — body (HTML or text)
- `attachments` (array) — optional attachments

Returns: `['status' => 'success']` or error with message/code.

```
public function send(array $data = []): array
{
    $validator = Validator::make($data, [
        'to' => 'required|email',
        'subject' => 'required|string|max:255',
        'message' => 'required|string',
```

```
    ]);
    if ($validator->fails()) {
        return ['status' => 'error', 'message' => $validator->errors(), 'code' => 422];
    }
    $mailer = $this->buildMailerConfiguration();
    Mail::mailer($mailer)->send(new NotificationMail(
        $data['to'], $data['subject'], $data['message'], $data['attachments'] ?? []
    ));
    return ['status' => 'success'];
}
```

Common Module Properties

The following properties are available to all module instances:

- `$module_name` (string) — module identifier
- `$module_type` (string) — one of: `Product`, `Plugin`, `Payment`, `Notification`
- `$config` (array) — values from `config.php`
- `$product_data` (array) — normalized product configuration (Product only)
- `$service_data` (array) — current service instance data (Product only)
- `$module_data` (array) — module-specific settings (Payment/Notification)
- `$service_uuid` (string) — current service UUID (Product)
- `$payment_gateway_uuid` (string) — payment gateway UUID (Payment)
- `$logger_context` (array) — default context for module logs

Webhook Endpoints

Modules can expose webhook endpoints via static routes. Typical signature:

```
// POST endpoint
public function apiWebhookPost(Request $request): Response
{
    // Validate, authenticate, process, respond
}

// GET endpoint
public function apiWebhookGet(Request $request): Response
{
    return response('ok', 200);
}
```

Admin Interface Methods

`adminPermissions(): array`

Defines module permissions for admin users.

Returns:

```
[
    [
        'name' => 'Permission Display Name',
        'key' => 'permission-key',
        'description' => 'Permission description'
    ]
]
```

Example:

```
public function adminPermissions(): array
{
    return [
        [
            'name' => 'View Servers',
            'key' => 'view-servers',
            'description' => 'View hosting servers',
        ],
        [
            'name' => 'Manage Servers',
            'key' => 'manage-servers',
```

```
            'description' => 'Create and manage hosting servers',
        ],
    ];
}
```

`adminSidebar(): array`

Defines sidebar menu items in admin area.

Returns:

```
[
    [
        'title' => 'Menu Title',
        'link' => 'route-name',
        'active_links' => ['route1', 'route2'],
        'permission' => 'required-permission-key'
    ]
]
```

Link resolution:

- **link** must reference a valid route from `adminWebRoutes()`. It can be either the route `name` or the `uri` defined there.
- The system resolves the item in this order: by route **name** → by route **uri** → fallback to a raw relative href.
- **Raw/fallback behavior**: if neither name nor uri matches a defined route, the menu item will render with a raw relative link (e.g., `/admin/modules/{Type}/{ModuleName}/{link}`). Such a link is considered broken and will lead to a non-working page (404). Keep links in sync with `adminWebRoutes()`.

Active state handling:

- `active_links` controls when the item (and its parent group) stays expanded and highlighted.
- The current admin route is matched against this array by route **name** or **uri**. A match sets the menu state to active/open.
- Use short keys or prefixes that you also use in `adminWebRoutes()` for consistent matching.

Working example:

```
public function adminWebRoutes(): array
{
    return [
```

```
        [
            'method' => 'get',
            'uri' => 'servers',
            'permission' => 'view-servers',
            'name' => 'servers',
            'controller' => 'ServerController@index',
        ],
        [
            'method' => 'get',
            'uri' => 'server-groups',
            'permission' => 'view-server-groups',
            'name' => 'server-groups',
            'controller' => 'ServerGroupController@index',
        ],
    ];
}

public function adminSidebar(): array
{
    return [
        [
            'title' => 'Server Management',
            'link' => 'servers', // resolves by route name
            'active_links' => ['servers', 'server-groups'],
            'permission' => 'manage-servers',
        ],
    ];
}
```

Broken link example (demonstration):

```
public function adminSidebar(): array
{
    return [
        [
            'title' => 'Diagnostics',
            'link' => 'diag', // NOT present in adminWebRoutes()
            'active_links' => ['diag'],
            'permission' => 'view-diagnostics',
        ],
```

```
    ];
}
// Result: menu renders an href like /admin/modules/Product/YourModule/diag
// Since no route exists, navigation will fail (404). Keep link and routes consistent.
```

`adminWebRoutes(): array`

Defines web routes for admin area.

Returns:

```
[
    [
        'method' => 'get|post|put|delete',
        'uri' => 'route/path',
        'permission' => 'required-permission',
        'name' => 'route.name',
        'controller' => 'ControllerName@methodName'
    ]
]
```

`adminApiRoutes(): array`

Defines API routes for admin area.

Returns: Same format as `adminWebRoutes()`

Client Area Methods

`getClientAreaMenuConfig(): array`

Defines client area menu tabs.

Returns:

```
[
    'tab_key' => [
        'name' => 'Tab Display Name',
        'template' => 'client_area.template_name'
    ]
]
```

Example:

```
```

```
public function getClientAreaMenuConfig(): array
{
    return [
        'general' => [
            'name' => 'General',
            'template' => 'client_area.general',
        ],
        'files' => [
            'name' => 'File Manager',
            'template' => 'client_area.files',
        ],
    ];
}
```

> **Note**: Each tab requires a corresponding `variables_{tab_name}()` method to provide data to the template.

`variables_{tab_name}(): array`

Provides variables for specific client area tab.

Returns: Array of variables for the template

Example:

```
public function variables_general(): array
{
    return [
        'service_data' => $this->service_data,
        'config' => $this->config,
        'status' => $this->getServiceStatus(),
    ];
}
```

`controllerClient_{tab_name}{Method}(Request $request): JsonResponse`

Handles AJAX requests from client area.

Parameters:

- `$request` - Laravel Request object

Returns: JsonResponse

Example:

```
public function controllerClient_generalGet(Request $request): JsonResponse
{
    try {
        $data = $this->getServiceDetails();
        return response()->json([
            'success' => true,
            'data' => $data,
        ]);
    } catch (Exception $e) {
        return response()->json([
            'success' => false,
            'message' => $e->getMessage(),
        ], 500);
    }
}
```

Utility Methods

`view(string $template, array $data = []): string`

Renders module template.

Parameters:

- `$template` - Template path relative to module views directory
- `$data` - Variables to pass to template

Returns: Rendered HTML

Example:

```
public function getServicePage(): string
{
    return $this->view('admin_area.service', [
        'service_data' => $this->service_data,
        'config' => $this->config,
    ]);
}
```

`config(string $key): mixed`

Gets configuration value from `config.php`.

Parameters:

- `$key` - Configuration key

Returns: Configuration value

Example:

```
$apiUrl = $this->config('api_url');
$apiKey = $this->config('api_key');
```

Logging Methods

`logInfo(string $action, array|string $request = [], array|string $response = []): void`

Logs informational message.

Example:

```
$this->logInfo('service_created', [
    'service_uuid' => $this->service_uuid,
    'product_data' => $this->product_data,
], ['status' => 'success']);
```

`logError(string $action, array|string $request = [], array|string $response = []): void`

Logs error message.

Example:

```

```

```
$this->logError('api_call_failed', [
    'endpoint' => '/api/create',
    'data' => $requestData,
], ['error' => $e->getMessage()]);
```

`logDebug(string $action, mixed $request = [], mixed $response = []): void`

Logs debug message.

Example:

```
$this->logDebug('processing_step', [
    'step' => 'validation',
    'data' => $inputData,
]);
```

Task System

Task::add()

Queues a background job.

```
Task::add($jobName, $queue, $inputData, $tags);
```

Parameters:

- `$jobName` - Job class name (e.g., 'ModuleJob')
- `$queue` - Queue name (e.g., 'Module')
- `$inputData` - Array of data to pass to job
- `$tags` - Array of tags for job identification

Job Data Structure for ModuleJob:

```
$data = [
    'module' => $this,                    // Module instance (required)
    'method' => 'methodToCall',           // Method name to execute (required)
    'tries' => 1,                         // Number of retry attempts (default: 1)
    'backoff' => 60,                      // Delay between retries in seconds (default: 10)
```

```
'timeout' => 600,                    // Maximum execution time in seconds (default: 600)
    'maxExceptions' => 1,                // Max exceptions before failure (default: 2, recommended: 1)
];

// Tags for job identification and filtering
$tags = ['create', 'service:' . $this->service_uuid];

Task::add('ModuleJob', 'Module', $data, $tags);
```

Complete Example:

```
public function create(): array
{
    $data = [
        'module' => $this,
        'method' => 'createJob',
        'tries' => 1,
        'backoff' => 60,
        'timeout' => 600,
        'maxExceptions' => 1,
    ];

    $tags = [
        'create',
        'hosting',
        'service:' . $this->service_uuid,
    ];

    $service = Service::find($this->service_uuid);
    $service->setProvisionStatus('processing');
    Task::add('ModuleJob', 'Module', $data, $tags);
    return ['status' => 'success'];
}
```

Helper Functions

view_admin_module()

Renders admin module view.

```
view_admin_module($type, $name, $view, $data = [], $mergeData = [])
```

Parameters:

- `$type` - Module type (e.g., 'Product')
- `$name` - Module name
- `$view` - View path
- `$data` - View data
- `$mergeData` - Additional data to merge

Example:

```
public function dashboard(Request $request): View
{
    $title = 'Server Dashboard';
    return view_admin_module('Product', 'MyHostingService', 'admin_area.dashboard',
compact('title'));
}
```

logModule()

Direct logging function.

```
logModule($type, $name, $action, $level, $request = [], $response = [])
```

Parameters:

- `$type` - Module type
- `$name` - Module name
- `$action` - Action being performed
- `$level` - Log level ('info', 'error', 'debug')
- `$request` - Request data
- `$response` - Response data

Service Model Methods

---

setProvisionStatus()

Updates service provisioning status.

```
$service->setProvisionStatus($status);
```

Status Values:

- `'pending'` - Waiting to be processed
- `'processing'` - Currently being processed
- `'completed'` - Successfully completed (active service)
- `'failed'` - Failed processing
- `'error'` - Error occurred during processing
- `'suspended'` - Service is suspended
- `'pause'` - Service is paused/idle
- `'terminated'` - Service is terminated

Example:

```
public function createJob(): array
{
    try {
        $service = Service::find($this->service_uuid);
        $service->setProvisionStatus('processing');

        // Generate service credentials
        $this->service_data['username'] = $this->product_data['username_prefix']
.
                                          random_int(100000, 999999)
.
                                          $this->product_data['username_suffix'];
        $this->service_data['password'] = generateStrongPassword(10);

        // Create API client and provision account
        $apiClient = new HostingAPIClient($this->config('api_url'), $this-
>config('api_key'));
        $result = $apiClient->createAccount([
            'domain' => $this->service_data['domain'],
            'username' => $this->service_data['username'],
            'password' => $this->service_data['password'],
        ]);

        if ($result['status'] === 'success') {
            $service->setProvisionData($this->service_data);
            $service->setProvisionStatus('completed');
```

```
            $this->logInfo('createJob', 'Service created successfully');
            return ['status' => 'success'];
        } else {
            $service->setProvisionStatus('failed');
            $this->logError('createJob', 'Service creation failed', $result);
            return ['status' => 'error', 'message' => $result['error']];
        }

    } catch (Exception $e) {
        $service->setProvisionStatus('failed');
        $this->logError('createJob', 'Exception occurred', $e->getMessage());
        return ['status' => 'error', 'message' => 'Service creation failed'];
    }
}
```

Validation Rules

Common Validation Patterns

```
// Domain validation
'domain' => 'required|regex:/^[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]*\.[a-zA-Z]{2,}$/',

// Username validation
'username' => 'required|alpha_dash|min:3|max:20',

// Strong password
'password' => 'required|min:8|regex:/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/',

// IP address
'ip' => 'nullable|ip',

// Plan/package validation
'plan' => 'required|in:basic,premium,enterprise',

// Disk space (in GB)
'disk_space' => 'required|integer|min:1|max:1000',
```

Error Handling Patterns

Standard Error Response

```
return [
    'status' => 'error',
    'message' => 'User-friendly error message',
    'errors' => $validator->errors(), // For validation errors
    'code' => 422, // HTTP status code
];
```

Exception Handling in Jobs

```
public function createJob(): array
{
    try {
        // Job logic here
        return ['status' => 'success'];

    } catch (ExternalServiceException $e) {
        $this->logError('createJob', 'External service error', $e->getMessage());
        return ['status' => 'error', 'message' => 'Service temporarily
unavailable'];

    } catch (ValidationException $e) {
        $this->logError('createJob', 'Validation error', $e->errors());
        return ['status' => 'error', 'message' => 'Invalid data provided'];

    } catch (Exception $e) {
        $this->logError('createJob', 'Unexpected error', $e->getMessage());
        return ['status' => 'error', 'message' => 'An unexpected error occurred'];
    }
}
```

Performance Best Practices

Caching

```
use Illuminate\Support\Facades\Cache;

// Cache expensive operations
$servers = Cache::remember("module_{$this->module_name}_servers", 300, function() {
    return $this->fetchServersFromAPI();
});

// Cache with tags for easier invalidation
Cache::tags(['module', $this->module_name])->put('key', $value, 300);

// Invalidate cache
Cache::tags(['module', $this->module_name])->flush();
```

Database Optimization

```
// Use database transactions for multiple operations
DB::transaction(function() use ($data) {
    $this->createUser($data);
    $this->assignPermissions($data);
    $this->sendWelcomeEmail($data);
});

// Eager loading relationships
$services = Service::with(['product', 'customer'])->get();
```

Memory Management

```
// Process large datasets in chunks
Service::chunk(100, function($services) {
    foreach ($services as $service) {
        $this->processService($service);
    }
});
```

Security Guidelines

Input Sanitization

```
// Always sanitize input
$data = array_map('trim', $data);
$data = array_map('strip_tags', $data);

// For HTML content, use proper escaping
$safeHtml = htmlspecialchars($userInput, ENT_QUOTES, 'UTF-8');
```

Sensitive Data Handling

```
// Encrypt sensitive data before storage
$encryptedPassword = encrypt($password);

// Don't log sensitive information
$this->logInfo('user_created', [
    'username' => $username,
    'email' => $email,
    // Don't log password or API keys
]);
```

Permission Checking

```
// Always check permissions in controllers
if (!$admin->hasPermission('Product-ModuleName-action')) {
    abort(403, 'Insufficient permissions');
}
```

Revision #8
Created 30 July 2025 02:26:01 by Dmytro Kravchenko
Updated 13 August 2025 05:11:33 by Dmytro Kravchenko