FAQ & Troubleshooting

Frequently Asked Questions

General Module Development

Q: What's the difference between Product, Plugin, Payment and Notification modules?

A: Each module type has its own purpose:

- **Product**: Manages services with full lifecycle (creation, suspension, deletion)
- **Plugin**: Extends system functionality without binding to specific services
- **Payment**: Processes payments through various payment gateways
- **Notification**: Sends notifications through various channels

Q: How to properly name modules?

A: Follow the conventions used in this codebase:

- Class name: `lowerCamelCase` with vendor prefix (e.g., `puqNextcloud`)
- Directory name: exactly the same as class name (e.g., `modules/Product/puqNextcloud`)
- Module file: `{ClassName}.php` (e.g., `puqNextcloud.php` inside directory `puqNextcloud`)
- Controller namespace: `Modules\{Type}\{ModuleName}\Controllers` (e.g., `Modules\Product\puqNextcloud\Controllers`)

Example:

```
// File: modules/Product/puqNextcloud/puqNextcloud.php
class puqNextcloud extends Product
{
    // ...
}
```

Q: What capabilities differ by module type in this codebase?

- **Product**

    - Implements service lifecycle methods (e.g., `create`, `suspend`, `unsuspend`, `terminate`, `change_package`) and often schedules jobs via `Task::add`
    - Typically defines `adminPermissions()`, `adminSidebar()`, `adminWebRoutes()`, `adminApiRoutes()`
    - Client Area: defines `getClientAreaMenuConfig()`, `variables_{tab}()`, `controllerClient_*`
- **Plugin**
    - May define `adminPermissions()`, `adminSidebar()`, `adminWebRoutes()`, `adminApiRoutes()`
    - Typically has no service lifecycle and does not use `Task::add`
- **Payment**
    - Usually defines `adminPermissions()` and `adminApiRoutes()` (admin web routes may be absent)
    - May have client templates (`views/client_area/*.blade.php`), but often uses static controllers `controllerClientStatic_*` in the module class (not bound to a specific service)
    - Generally does not use queues (`Task::add`) nor service lifecycle methods
- **Notification**
    - Usually defines `adminPermissions()` and a configuration interface
    - Web/API routes may be absent or minimal (e.g., test connection)

Q: Can I use external libraries in modules?

A: Yes, but follow the recommendations:

```
// In your project's composer.json add dependencies
{
    "require": {
        "guzzlehttp/guzzle": "^7.0",
        "league/oauth2-client": "^2.0"
    }
}

// In module use through autoloader
use GuzzleHttp\Client;
use League\OAuth2\Client\Provider\GenericProvider;
```

Q: How to handle multilingualism in modules?

A: Use language files:

```
// lang/en.php
return [
    'service_created' => 'Service created successfully',
```

```
    'invalid_credentials' => 'Invalid API credentials',
];

// In module code
__('Product.ModuleName.service_created')
```

Database & Migrations

Q: How to properly create tables in modules?

A: In the `activate()` method:

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

            // Always add foreign keys to link with main tables
            $table->foreign('service_uuid')->references('uuid')->on('services');

            // Add indexes for frequently used fields
            $table->index(['service_uuid', 'status']);
        });

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

Q: How to update table structure when updating module?

A: Use the `update()` method:

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

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

        if (version_compare($currentVersion, '1.2.0', '<')) {
            Schema::table('module_custom_table', function (Blueprint $table) {
                $table->index('new_field');
            });
        }

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

Jobs & Queue System

Q: When to use synchronous vs asynchronous operations?

A: Guidelines:

Synchronous operations (direct method calls):

- Data validation
- Simple database operations
- Retrieving cached data
- Operations < 2 seconds

Asynchronous operations (via Task::add) — primarily applicable to Product modules:

- External API calls
- Resource creation/modification

- Long-running operations (> 2 seconds)

> **Note**: Payment and Notification modules in this codebase generally do not use queues

Q: How to ensure job execution reliability?

A: Use proper configuration:

```
$data = [
    'module' => $this,                // Module instance (required)
    'method' => 'createJob',          // Method to execute (required)
    'tries' => 1,                     // Number of attempts (default: 3, recommended: 1)
    'backoff' => 60,                  // Delay between attempts in seconds (default: 10)
    'timeout' => 600,                 // Execution timeout (default: 600)
    'maxExceptions' => 1,             // Maximum exceptions (default: 2, recommended: 1)
];

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

$service = Service::find($this->service_uuid);
$service->setProvisionStatus('processing');
Task::add('ModuleJob', 'Module', $data, $tags); // used in Product modules
```

Q: How to monitor job execution?

A: Use logging and monitoring:

```
public function createJob(): array
{
    $startTime = microtime(true);

    try {
        $this->logInfo('createJob.started', ['service_uuid' => $this->service_uuid]);

        // Main logic - example of real provisioning
        $apiClient = new HostingAPIClient($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'],
        ]);

        $executionTime = round((microtime(true) - $startTime) * 1000, 2);
        $this->logInfo('createJob.completed', [
            'service_uuid' => $this->service_uuid,
            'execution_time_ms' => $executionTime,
            'result' => $result
        ]);

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

    } catch (Exception $e) {
        $this->logError('createJob.failed', [
            'service_uuid' => $this->service_uuid,
            'error' => $e->getMessage(),
            'trace' => $e->getTraceAsString()
        ]);

        return ['status' => 'error', 'message' => 'Service creation failed'];
    }
}
```

API Integration

Q: How to properly integrate with external APIs?

A: Follow best practices:

```
use GuzzleHttp\Client;
use GuzzleHttp\Exception\GuzzleException;

class ExternalAPIClient
{
    private Client $client;
    private string $apiKey;

    public function __construct(string $baseUrl, string $apiKey)
```

```
    {
        $this->apiKey = $apiKey;
        $this->client = new Client([
            'base_uri' => $baseUrl,
            'timeout' => 30,
            'headers' => [
                'Authorization' => 'Bearer ' . $apiKey,
                'Content-Type' => 'application/json',
                'User-Agent' => 'PUQcloud-Module/1.0',
            ],
        ]);
    }

    public function makeRequest(string $method, string $endpoint, array $data = []): array
    {
        $attempts = 0;
        $maxAttempts = 3;

        while ($attempts < $maxAttempts) {
            try {
                $response = $this->client->request($method, $endpoint, [
                    'json' => $data,
                ]);

                return [
                    'success' => true,
                    'data' => json_decode($response->getBody(), true),
                    'status_code' => $response->getStatusCode()
                ];

            } catch (GuzzleException $e) {
                $attempts++;

                if ($attempts >= $maxAttempts) {
                    return [
                        'success' => false,
                        'error' => $e->getMessage(),
                        'status_code' => $e->getCode()
                    ];
                }
```

```
                // Exponential backoff
                sleep(pow(2, $attempts));
            }
        }
    }
}
```

Security

Q: How to securely store API keys and passwords?

A: Use encryption and environment variables:

```
// In .env file
EXTERNAL_API_KEY=your_secret_key_here

// In module config.php
return [
    'api_key' => env('EXTERNAL_API_KEY'),
    // DON'T store secrets directly in config!
];

// For passwords use Laravel encryption
$encryptedPassword = encrypt($password);
$decryptedPassword = decrypt($encryptedData);
```

Q: How to validate input data?

A: Use strict validation:

```
public function saveServiceData(array $data = []): array
{
    // Clean input data
    $data = array_map('trim', $data);

    $validator = Validator::make($data, [
        'domain' => [
            'required',
            'string',
```

```
            'max:255',
            'regex:/^[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]*\.[a-zA-Z]{2,}$/'
        ],
        'username' => [
            'required',
            'alpha_dash',
            'min:3',
            'max:20',
            'not_in:admin,root,administrator' // Forbidden names
        ],
        'password' => [
            'required',
            'min:8',
            'regex:/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]/'
        ],
    ], [
        'domain.regex' => 'Please enter a valid domain name',
        'password.regex' => 'Password must contain uppercase, lowercase, number and special
character',
    ]);

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

    // Additional business logic validation
    if ($this->isDomainBlacklisted($data['domain'])) {
        return [
            'status' => 'error',
            'message' => 'Domain is not allowed',
            'code' => 403,
        ];
    }

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

```
}

private function isDomainBlacklisted(string $domain): bool
{
    // Example implementation - configure according to your requirements
    $blacklistedDomains = ['spam.com', 'test.invalid'];
    return in_array($domain, $blacklistedDomains);
}
```

Troubleshooting Guide

Module Not Loading

Symptoms:

- Module doesn't appear in the list
- "Class not found" error
- Module in "error" status

Solutions:

1. Check file permissions:

```
chmod 755 modules/Product/YourModule/
chmod 644 modules/Product/YourModule/*.php
chmod 644 modules/Product/YourModule/config.php
```

2. Check PHP syntax:

```
php -l modules/Product/YourModule/YourModule.php
```

3. Check class name correctness:

```
// File: modules/Product/MyModule/MyModule.php
<?php

class MyModule extends Product  // Name must exactly match file name
{
    // ...
}
```

4. Check configuration file:

```
// config.php must return an array
return [
    'name' => 'Module Name',
    'version' => '1.0.0',
    // ... other required fields
];
```

Routes Not Working

Symptoms:

- 404 error when accessing module routes
- Routes don't appear in `php artisan route:list`

Solutions:

1. Clear route cache:

```
php artisan route:clear
php artisan config:clear
```

2. Check module status:

```
# Module should be in 'active' status
```

3. Check route definition correctness:

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

```
            'method' => 'get',                              // Correct HTTP method
            'uri' => 'dashboard',                           // URI without leading
slashes
            'permission' => 'view-dashboard',               // Existing permission (from
adminPermissions)
            'name' => 'dashboard',                          // Unique route name
            'controller' => 'ModuleController@dashboard', // Existing controller and
method
        ]
    ];
}
```

4. You can also define admin API routes (used for AJAX in admin area):

```
public function adminApiRoutes(): array
{
    return [
        [
            'method' => 'get',
            'uri' => 'server/{uuid}',                   // Route parameters are
supported
            'permission' => 'configuration',
            'name' => 'server.get',
            'controller' => 'puqNextcloudController@getServer',
        ],
    ];
}
```

5. Check controller existence:

```
// File must exist: modules/Product/YourModule/Controllers/YourController.php
namespace Modules\Product\YourModule\Controllers;

use App\Http\Controllers\Controller;

class YourController extends Controller
{
    public function dashboard() {
        // Method must exist and be public
    }
```

```
}
```

Jobs Not Processing

Symptoms:

- Jobs remain in "queued" status
- Jobs disappear without execution
- Errors during job execution

Solutions:

1. Check queue worker operation:

```
# Start worker
php artisan queue:work

# Check queue configuration
php artisan queue:monitor
```

2. Check failed jobs:

```
# View failed jobs
php artisan queue:failed

# Retry specific job by ID (replace 1 with actual ID)
php artisan queue:retry 1

# Retry all failed jobs
php artisan queue:retry all

# Clear all failed jobs
php artisan queue:flush
```

3. Check module logs:

```
// View module logs in admin panel or in log file
tail -f storage/logs/laravel.log
```

4. Ensure job data correctness:

```
// Check that all required data is passed
```

```
$data = [
    'module' => $this,                // Required: module instance
    'method' => 'createJob',          // Required: method must exist in module class
    'tries' => 1,                     // Optional: number of attempts
    'backoff' => 60,                  // Optional: delay between attempts
    'timeout' => 600,                 // Optional: maximum execution time
    'maxExceptions' => 1,             // Optional: maximum exceptions
];

$tags = ['create', 'service:' . $this->service_uuid];
Task::add('ModuleJob', 'Module', $data, $tags);
```

API Integration Issues

Symptoms:

- Timeouts when accessing external APIs
- Authentication errors
- Incorrect API responses

Solutions:

1. Check API configuration:

```
// Ensure all settings are correct
$apiKey = $this->config('api_key');
$apiUrl = $this->config('api_url');

if (empty($apiKey) || empty($apiUrl)) {
    $this->logError('api_config', 'API credentials not configured');
    return ['status' => 'error', 'message' => 'API not configured'];
}
```

2. Add debug information:

```
public function callAPI($endpoint, $data = [])
{
    $this->logDebug('api_call', [
        'endpoint' => $endpoint,
        'data' => $data,
        'headers' => $this->getHeaders()
```

```
    ]);

    try {
        $response = $this->client->post($endpoint, ['json' => $data]);

        $this->logDebug('api_response', [
            'status_code' => $response->getStatusCode(),
            'body' => $response->getBody()->getContents()
        ]);

        return $response;

    } catch (Exception $e) {
        $this->logError('api_error', [
            'endpoint' => $endpoint,
            'error' => $e->getMessage()
        ]);
        throw $e;
    }
}
```

3. Use retry mechanism:

```
private function apiCallWithRetry($endpoint, $data, $maxAttempts = 3)
{
    $attempt = 0;

    while ($attempt < $maxAttempts) {
        try {
            return $this->callAPI($endpoint, $data);
        } catch (Exception $e) {
            $attempt++;

            if ($attempt >= $maxAttempts) {
                throw $e;
            }

            $delay = pow(2, $attempt); // Exponential backoff
            sleep($delay);
        }
```

```
    }
}
```

Performance Issues

Symptoms:

- Slow module page loading
- High memory consumption
- Timeouts during operations

Solutions:

1. Optimize database queries:

```
// Bad: N+1 queries
$services = Service::all();
foreach ($services as $service) {
    echo $service->product->name; // Additional query for each service
}

// Good: Eager loading
$services = Service::with('product')->get();
foreach ($services as $service) {
    echo $service->product->name; // Data already loaded
}
```

2. Use caching:

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

public function getExpensiveData($serviceId)
{
    return Cache::remember("service_data_{$serviceId}", 300, function() use ($serviceId)
{
        return $this->fetchDataFromAPI($serviceId);
    });
}

// Invalidate cache when data changes
public function updateServiceData($serviceId, $data)
```

```
{
    $result = $this->updateData($serviceId, $data);
    Cache::forget("service_data_{$serviceId}");
    return $result;
}
```

3. Process large datasets in chunks:

```
// Instead of loading all records at once
Service::chunk(100, function ($services) {
    foreach ($services as $service) {
        $this->processService($service);
    }
});
```

4. Monitor performance:

```
private function measureOperation($operationName, callable $operation)
{
    $startTime = microtime(true);
    $startMemory = memory_get_usage();

    $result = $operation();

    $executionTime = (microtime(true) - $startTime) * 1000;
    $memoryUsed = memory_get_usage() - $startMemory;

    $this->logInfo('performance', [
        'operation' => $operationName,
        'execution_time_ms' => round($executionTime, 2),
        'memory_used_kb' => round($memoryUsed / 1024, 2),
        'peak_memory_mb' => round(memory_get_peak_usage() / 1024 / 1024, 2)
    ]);

    return $result;
}
```

Client Area Issues

Symptoms:

- Tabs don't display
- AJAX requests don't work
- Templates don't render

Solutions:

1. Check client area configuration:

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

2. Ensure template existence:

```
# File must exist
modules/Product/YourModule/views/client_area/general.blade.php
```

3. Check variable methods:

```
// Method must exist for each tab
public function variables_general(): array
{
    return [
        'service_data' => $this->service_data,
        'config' => $this->config,
    ];
}
```

4. Check AJAX controllers:

```
// Pattern A: method per tab and HTTP verb suffix
public function controllerClient_generalGet(Request $request): JsonResponse
{
    return response()->json([
        'status' => 'success',
```

```
        'data' => $this->getGeneralData()
    ]);
}

// Pattern B: custom action name without HTTP verb suffix
public function controllerClient_user_quota(Request $request): JsonResponse
{
    // ...
}

// Static client controllers (no bound service, e.g., Payment module forms)
public function controllerClientStatic_fetch_public_key(array $data = []): JsonResponse
{
    $request = $data['request'] ?? request();
    // ...
}
```

5. Optionally, modules may expose dedicated client API routes:

```
public function clientApiRoutes(): array
{
    return [
        // Define client API endpoints if needed
    ];
}
```

Debugging Tools

Enable Detailed Logging

```
// In module methods for debugging
public function createJob(): array
{
    $this->logDebug('createJob.start', [
        'service_uuid' => $this->service_uuid,
        'product_data' => $this->product_data,
        'service_data' => $this->service_data
    ]);
```

```
    // ... main logic

    $this->logDebug('createJob.end', ['result' => $result]);
    return $result;
}
```

Using Laravel Telescope

```
# Install Telescope for debugging
composer require laravel/telescope --dev
php artisan telescope:install
php artisan migrate
```

Performance Profiling

```
// Add to module for performance monitoring
use Illuminate\Support\Facades\DB;

public function enableQueryLogging()
{
    DB::enableQueryLog();
}

public function logQueries($operation)
{
    $queries = DB::getQueryLog();
    $this->logDebug('database_queries', [
        'operation' => $operation,
        'query_count' => count($queries),
        'queries' => $queries
    ]);
}
```

This FAQ and troubleshooting guide will help developers quickly resolve the most common issues when developing modules for PUQcloud.

Revision #4
Created 30 July 2025 02:36:44 by Dmytro Kravchenko
Updated 13 August 2025 05:11:33 by Dmytro Kravchenko