


```

|   └─ [ ] ExternalAPIClient.php
└─ views/
|   └─ admin_area/
|       └─ [ ] product.blade.php
|           └─ [ ] service.blade.php
|               └─ client_area/
|                   └─ [ ] general.blade.php
└─ lang/
    └─ [ ] en.php
        └─ [ ] pl.php

```

Core Module Implementation

Main Module Class

- [] Class extends appropriate parent (Product/Plugin/Payment/Notification)
- [] Constructor calls `parent::__construct()`
- [] All required methods implemented
- [] Proper error handling in all methods
- [] Comprehensive logging implemented

Lifecycle Methods

- [] `activate()` - Creates database tables, initial setup
- [] `deactivate()` - Cleans up resources
- [] `update()` - Handles version migrations

Product Module Specific

- [] `getProductData()` - Processes product configuration
- [] `saveProductData()` - Validates and saves product config
- [] `getProductPage()` - Returns admin product configuration HTML
- [] `getServiceData()` - Processes service instance data
- [] `saveServiceData()` - Validates and saves service config
- [] `getServicePage()` - Returns admin service configuration HTML

Service Lifecycle (Product Modules)

- [] `create()` - Queues service creation job using correct `Task::add()` pattern
- [] `createJob()` - Actual service creation logic executed asynchronously
- [] `suspend()` - Queues service suspension job (optional - implement if needed)
- [] `suspendJob()` - Actual service suspension logic (optional - implement if needed)
- [] `unsuspend()` - Queues service reactivation job (optional - implement if needed)
- [] `unsuspendJob()` - Actual service reactivation logic (optional - implement if needed)

- [] `termination()` - Queues service termination job (optional - implement if needed)
- [] `terminationJob()` - Actual service termination logic (optional - implement if needed)

Admin Interface

Permissions

- [] `adminPermissions()` returns proper permission definitions
- [] All permissions have descriptive names and keys
- [] Permission keys follow naming convention

Navigation

- [] `adminSidebar()` returns navigation menu items
- [] All menu items have required permissions
- [] Menu links point to valid routes

Routes

- [] `adminWebRoutes()` defines all web routes
- [] `adminApiRoutes()` defines all API routes
- [] All routes have proper permissions
- [] Route names are unique and descriptive

Controllers

- [] Controllers extend Laravel Controller class
- [] All controller methods are public
- [] Web methods return View objects
- [] API methods return JsonResponse objects
- [] Proper input validation implemented
- [] Error handling implemented

Client Area (Product Modules)

Configuration

- [] `getClientAreaMenuConfig()` returns menu tabs
- [] Each tab has name and template path
- [] Template files exist for all tabs

Variables

- [] `variables_{tab_name}()` methods exist for all tabs
- [] Methods return appropriate data arrays

AJAX Controllers

- [] `controllerClient_{tab_name}Get()` methods implemented for AJAX requests
- [] `controllerClient_{tab_name}Post()` methods implemented if needed
- [] Methods handle requests appropriately and return JsonResponse
- [] Proper error handling and validation in AJAX methods

Database Design

Tables

- [] Proper table naming convention used
- [] Primary keys defined appropriately
- [] Foreign keys reference correct tables
- [] Indexes added for frequently queried columns
- [] JSON columns used for flexible data storage
- [] Enum fields for status tracking

Migrations

- [] `activate()` creates all necessary tables
- [] `deactivate()` properly cleans up tables
- [] `update()` handles schema changes between versions
- [] Foreign key constraints properly handled

API Integration

External API Client

- [] GuzzleHttp Client properly configured (timeouts, base_uri, headers)
- [] Authentication implemented correctly (Bearer token, API key, etc.)
- [] Retry logic implemented for failed requests with exponential backoff
- [] Rate limiting considerations implemented
- [] Error handling for different HTTP response codes
- [] API credentials stored in environment variables, not hardcoded

Security

- [] API credentials stored securely (environment variables)
- [] Sensitive data encrypted before storage
- [] Input sanitization implemented
- [] Output escaping for user-facing data

Validation & Security

Input Validation

- [] All user inputs validated using Laravel Validator
- [] Custom validation rules for business logic
- [] Proper error messages in multiple languages
- [] XSS prevention implemented
- [] SQL injection prevention through proper ORM usage

Security Measures

- [] Permission checks in all controller methods
- [] CSRF protection for forms
- [] Rate limiting for API endpoints
- [] Secure password generation and storage

Error Handling & Logging

Error Handling

- [] Try-catch blocks around all external API calls and database operations
- [] Graceful degradation for failed operations
- [] User-friendly error messages (no technical details exposed to users)
- [] Proper HTTP status codes returned (200, 422, 500, etc.)
- [] Service status properly updated on failures (setProvisionStatus('failed'))

Logging

- [] `$this->logInfo()` for successful operations with context
- [] `$this->logError()` for failed operations with error details
- [] `$this->logDebug()` for development debugging (not in production)
- [] Structured logging with service_uuid, action, and relevant data
- [] No sensitive data logged (passwords, API keys, personal data)

Performance Optimization

Caching

- [] Expensive API calls cached with `Cache::remember()`
- [] Cache keys use module-specific prefixes
- [] Cache invalidation strategy implemented
- [] Cache TTL set appropriately (typically 5-30 minutes for API data)

Database

- [] N+1 query problems avoided using eager loading (with())
- [] Appropriate indexes created on foreign keys and frequently queried columns
- [] Large datasets processed in chunks using chunk() method
- [] Database transactions used for related operations (DB::transaction())
- [] Bulk inserts used instead of multiple single inserts

Memory Management

- [] Memory usage monitored for large operations
- [] Large arrays/objects unset when no longer needed
- [] Streaming used for large file operations

Testing Phase

Unit Testing

- [] Tests for all core methods
- [] Validation logic tested
- [] Error handling tested
- [] Mock external dependencies

Integration Testing

- [] Full service lifecycle tested
- [] API integration tested with real/mock services
- [] Database operations tested
- [] Queue job processing tested

Manual Testing

- [] Admin interface navigation works
- [] Product/service configuration saves correctly
- [] Client area displays properly
- [] Service operations work end-to-end
- [] Error scenarios handled gracefully

Performance Testing

- [] Load testing for API endpoints
- [] Memory usage profiling
- [] Database query optimization
- [] Cache effectiveness measurement

Pre-Deployment Checklist

Configuration

- [] `config.php` contains all required metadata
- [] Version number updated
- [] Environment-specific settings configured
- [] API credentials configured in environment

Documentation

- [] README with installation instructions
- [] Configuration guide
- [] API documentation (if applicable)
- [] Troubleshooting guide

Security Review

- [] Code review completed
- [] Security vulnerabilities addressed
- [] Penetration testing performed (for complex modules)
- [] Compliance requirements met

Deployment Preparation

- [] Database backup procedures documented
- [] Rollback plan prepared
- [] Monitoring alerts configured
- [] Performance baselines established

Post-Deployment

Monitoring

- [] Module logs monitored for errors
- [] Performance metrics tracked

- User feedback collected
- Error rates within acceptable limits

Maintenance

- Regular security updates applied
- Performance optimizations implemented
- Bug fixes deployed promptly
- Documentation kept up-to-date

Code Quality Standards

PHP Standards

- PSR-12 coding standards followed
- Type hints used where applicable
- Proper namespacing implemented
- DocBlocks for all public methods

Laravel Best Practices

- Eloquent ORM used for database operations
- Service container used for dependency injection
- Facades used appropriately
- Events and listeners used for decoupling

Module-Specific Standards

- Consistent error response format
- Standardized logging format
- Common validation patterns used
- Consistent naming conventions

Version Control

Git Practices

- [] Meaningful commit messages
- [] Feature branches used for development
- [] Code reviewed before merging
- [] Version tags created for releases

Release Management

- [] CHANGELOG.md maintained
- [] Semantic versioning followed
- [] Breaking changes documented
- [] Migration guides provided

Common Pitfalls to Avoid

Don't Do This

- [] Store API keys in code or config files (use .env variables)
- [] Skip input validation "for internal use"
- [] Use direct SQL queries instead of Eloquent ORM or Query Builder
- [] Ignore error handling in background jobs (always try-catch)
- [] Log sensitive information (passwords, API keys, personal data)
- [] Skip database indexes for frequently queried columns
- [] Use synchronous operations for long-running tasks (use Task::add())
- [] Hardcode configuration values (use config() method)
- [] Skip permission checks in controllers
- [] Ignore rate limiting for external APIs
- [] Use non-existent methods like `queueTask()` (use Task::add() directly)
- [] Use incorrect Task::add() parameters (no 'uuid' parameter needed)
- [] Call non-existent helper methods like `getJobParameter()`

Best Practices

- [] Use environment variables for sensitive configuration (.env file)
- [] Validate all inputs with Laravel Validator rules
- [] Use Laravel's built-in security features (CSRF, XSS protection)
- [] Implement comprehensive error handling with try-catch blocks
- [] Use structured logging with context (service_uuid, action, data)
- [] Optimize database queries and add indexes on foreign keys
- [] Use Task::add('ModuleJob', 'Module', \$data, \$tags) with correct parameters
- [] Make configuration flexible using config.php and .env
- [] Implement proper authorization checks in all controller methods
- [] Handle external service failures gracefully with retries
- [] Follow the real API methods from actual module classes

- [] Use correct Task::add() parameters: \$data must contain 'module' and 'method' only, no 'uuid'
 - [] Always call setProvisionStatus('processing') before queueing tasks
 - [] Store service data using setProvisionData() method
-

Remember: A well-designed module is secure, performant, maintainable, and provides excellent user experience. Take time to plan, implement best practices, and thoroughly test before deployment.

Revision #5

Created 30 July 2025 02:35:13 by Dmytro Kravchenko

Updated 13 August 2025 05:11:33 by Dmytro Kravchenko