Developer Guide

WP Frame Studio Developer Guide

Introduction: The Enterprise-Grade WP Framework

Welcome to the WP Frame Studio Developer Guide. This framework is designed to bring modern, large-scale application patterns (like those found in Laravel) to the WordPress ecosystem, focusing on **performance, modularity, and exceptional Developer Experience (DX)**.

The core philosophy is **Opt-in Modularity**: you only pay the performance cost for the features you explicitly choose to activate via configuration. Our goal is to shift development from managing WordPress hooks to writing clean, testable business logic inside classes.

1. Core Architecture and The Boot Process

The framework replaces the traditional "spaghetti" approach of WordPress development with a structured **Service Container** model.

1.1. The Application Container (Application.php)

The **Application Container** is the heart of the framework—a central registry for all classes, settings, and services. It lives in src/Foundation/Application.php.

- **Dependency Injection (DI):** This is the crucial feature. Instead of manually creating objects ($cache = new CacheManager($config)), the container automatically builds and injects dependencies into class constructors.
- **Service Locator:** It allows you to retrieve any registered service instance using the global **app() helper** or via **Facades**.
- **Singletons:** Services that should only exist once (like the Database Connection, Router, or Cache Manager) are registered as singletons to save memory and ensure consistency across the request.

1.2. Service Providers (The Modular Gateway)

Service Providers are the core mechanism for **modularity and opt-in control**. Every major feature (Queue, Cache, Database, Routing) is defined by a Provider.

Method	Purpose	When to Use

register()	Binds services to the Application Container (the "setup" phase).	Use to define class bindings, singletons, and link **Contracts (Interfaces)** to their concrete implementations. **DO NOT** register WordPress hooks here.
boot()	Connects the registered services to the WordPress lifecycle (the "go" phase).	**All WordPress hook registration MUST happen here.** This ensures your hooks are only registered if the service provider is enabled. Use Hook::action() and Hook::filter().

Opt-in Control: If a Service Provider (e.g., QueueServiceProvider) is *not* listed in the providers array in config/app.php, the entire module is ignored, guaranteeing zero performance overhead.

1.3. The Request Lifecycle: Hooking into WordPress

The framework boots extremely early in the WordPress lifecycle, ensuring services are available before the main WordPress actions fire.

1. **WP Loads:** functions.php or my-awesome-plugin.php runs.
2. **Bootstrap:** The application loads the Composer autoloader and executes bootstrap/app.php, which creates and initializes the **Application Container**.
3. **Core Bootstrap:** The framework's BootManager immediately runs foundational services:
    - Loads **.env** variables.
    - Loads and merges **config/** files.
    - Initializes the **WpRequest** and **SessionManager** (using Transients/DB).
4. **Providers Boot:** The framework iterates through the configured Service Providers, calling their **register()** methods first, then their **boot()** methods.
5. **Hooks Registered:** The boot() methods register all custom actions, filters, routes, and CPTs via the framework's internal **Hook Manager**.
6. **WP Continues:** WordPress continues its native loop. Any subsequent action or filter request is now handled by your clean, namespaced framework classes instead of inline hook callbacks.

2. Core Features: Service Implementation & Best

Practices

2.1. Contract-Driven Development and Facades

The framework is defined by interfaces (**Contracts** in src/Contracts). This is essential for testability and extensibility.

Element	Description	Usage and Location
Contract (Interface)	The blueprint for a service (e.g., QueueInterface.php).	Type-hint the **Interface** in class constructors to ensure your code accepts any compatible implementation.
Facade	Provides a static access point to a service bound in the container (e.g., Cache::get('key')).	Used for convenience when you need quick access to a service instance outside of DI (e.g., in a helper function or route definition).

Best Practice: Always type-hint against Contracts in your constructors.

```
// Bad: Tight Coupling
public function __construct(TransientDriver $cache) { ... }

// Good: Loose Coupling via Contract
public function __construct(CacheInterface $cache) { ... }
```

2.2. Security-First Input Handling: The Form Request

The **Form Request** is the most significant security enhancement. It ensures **sanitization, validation, and authorization** are completed before your business logic executes.

1. **Generate:** Use the CLI: php frame make:request MyFormRequest.
2. **Authorize:** Define the authorize() method to check user capabilities. **This is mandatory for security.**
   ```
   // Example: Only allow users who can 'manage_options' to run this request
   public function authorize(): bool {
       return Authorizer::can('manage_options');
   }
   ```
3. **Validate/Sanitize:** Define the rules() method. The framework will automatically use the

Sanitizer utility to clean input *before* validation runs.

4. **Use in Controller:**
   // In MyController.php
   // The framework handles the validation and authorization automatically before calling this method.
   public function handleSubmission(MyFormRequest $request) {
   // If we reach here, the data is guaranteed to be authorized, validated, and sanitized.
   $title = $request->input('title'); // Safe to use
   // ... business logic
   }

Principle: NEVER trust input directly. The **Form Request** acts as a security gate, throwing exceptions or redirecting if the input is compromised or invalid.

2.3. Asynchronous Tasks with Queues

The framework abstracts background processing using the **Job** concept and integrates with **WP-Cron** for scheduling.

- **Job Class:** Any task that takes more than 1 second (sending emails, image processing, API calls) should be defined as a **Job** (extends Job.php in src/Queue/).
- **Dispatching:** Dispatch the job asynchronously using the **Queue Facade**.
  // Dispatching an email job
  Queue::dispatch(new SendWelcomeEmail($user_id));

Best Practice: Keep your HTTP response fast. If a function is slow, turn it into a Job and dispatch it immediately.

3. Data and Presentation Implementation

3.1. Custom Post Types and Eloquent ORM

WP Frame Studio replaces raw WordPress functions for Post Type and Model interaction with the expressive **Eloquent ORM**.

1. **Define the CPT:** Create a class (e.g., ProjectPostType.php in app/PostTypes) that extends the framework's PostType class.
2. **Define the Model:** Create a corresponding Eloquent model (e.g., Project.php in app/Models) that extends the framework's PostModel.

// In a Controller or Service
use App\Models\Project;

// Example: Fetch all published 'project' posts, ordered by title
$projects = Project::query()

```
    ->where('post_status', 'publish')
    ->orderBy('post_title', 'asc')
    ->limit(10)
    ->get(); // Returns a Collection of Project models
```

Benefits: Models handle metadata casting, relationships, and provide a database-agnostic interface, making your code portable and testable.

3.2. Template Resolution (Themes Only)

In WP Theme Frame, the **Template Resolver** (in src/Wordpress/Template/) decouples presentation logic from the WordPress query loop.

1. **The Flow:** The TemplateResolver intercepts the request via the template_include filter.
2. **Mapping:** You define logic in app/Templates/ that determines which **Controller** method should handle the request (e.g., if is_single() is true, pass to SingleController@show).
3. **Rendering:** The Controller fetches the data and returns a rendered **Twig** view.

```
// Inside SingleController.php (Fetches data, returns view)
use RactStudio\FrameStudio\Support\Facades\View;

public function show($wp_query) {
    // Post::find will use the current queried object intelligently
    $post = Post::find($wp_query->post->ID);

    // Render the Twig template, passing only necessary data
    return View::render('templates.single', [
        'post' => $post,
        'author' => $post->author,
        'related_projects' => $post->relatedProjects()->get(),
    ]);
}
```

3.3. Asset Management and Build Pipeline

Assets are managed through a central AssetManifest.php and the EnqueueServiceProvider to ensure that compiled frontend files are always loaded with correct versioning (cache-busting).

Best Practice: Use a modern build tool (like Webpack/Mix/Vite) to compile your Tailwind CSS and Alpine.js. The framework will then read the generated **mix-manifest.json** to automatically

load the correct, cache-busted path from the build/ directory.

---

4. Developer Tools and Security Hardening

4.1. The frame CLI and Scaffolding

The frame executable is your main productivity tool. Use it to maintain your code structure and rapidly generate boilerplate files.

Command Example	Scaffolding/Maintenance Task	Enforced Best Practice
php frame make:controller Auth	Creates a new class for request handling.	Enforces Controllers to live in app/Http/Controllers.
php frame make:request Login	Creates a new security gate.	Enforces **Security-First** validation/authorization.
php frame make:post-type Event	Creates a new CPT definition.	Enforces clean CPT definition in a dedicated class.
php frame make:command Optimize	Creates a custom utility command.	Allows for easy maintenance and debugging outside of HTTP requests.
php frame migrate	Runs the database migration files.	Enforces robust database schema version control.

4.2. Security and Hardening Best Practices

1. **Sanitization:** Always use the framework's **Sanitizer** utility or the built-in sanitization from the **Form Request**. Never use raw user input in database queries or output without explicit cleaning.
2. **Authorization:** Always check user capabilities using the **Authorizer** utility (e.g., Authorizer::can('edit_posts')) before performing any state-changing actions.
3. **Nonce Verification:** Use the **Nonce** Facade for all forms and API requests to protect against Cross-Site Request Forgery (CSRF).
4. **Performance Profiling:** In development, use the framework's **Profiler**

(Debug/Profiler.php) to identify memory leaks and slow components. Never allow heavy logic inside a Service Provider's register() method.

4.3. Naming Conventions

Maintain strict naming and prefixing to prevent namespace collisions with other plugins:

- **Plugin/Theme Prefix:** All custom functions, global constants, database tables, and WP options **MUST** be prefixed (e.g., wpfs_ for framework features, and your plugin's name, like myplugin_).
- **PSR-4/Namespaces:** Rely on namespaces (App\Http\Controllers) for class-level collision prevention.

This guide provides the framework's architecture, key feature implementation, and critical security/performance directives.

5. Potential Challenges & Considerations

5.1. Performance Overhead

// Even with lazy loading, the container initialization has cost

$app = new Application(); // This runs on every request

Mitigation: Your lazy loading strategy is good, but consider:

- **Container compilation** for production
- **Selective provider loading** based on context (admin/frontend/API)

5.2. Complexity Barrier

- **Learning curve** for WP developers unfamiliar with Laravel patterns
- **Multiple abstraction layers** might confuse beginners

5.3. WordPress Compatibility

- **Multisite considerations** - Ensure all features work across networks
- **Plugin conflicts** - Other plugins might interfere with container/namespaces

5.4. Memory Usage

// Each Service Provider adds to memory footprint

class SomeProvider extends ServiceProvider {

```
    public function register() {
        // Even empty providers have overhead
    }
}
```

Recommendation: Add provider prioritization and conditional registration.

6. Technical Recommendations

6.1. Enhanced Boot Optimization

```
class BootManager {
    public function shouldBoot(): bool {
        // Don't boot for WP-CLI if not needed
        // Don't boot for specific WP actions
        return !defined('DOING_AJAX') || $this->isRelevantAjax();
    }
}
```

6.2. Add Configuration Caching

```
// Cache merged config for production
if (WP_DEBUG) {
    $config = $this->loadConfigFiles();
} else {
    $config = $this->getCachedConfig();
}
```

6.3. Database Connection Management

```
class DatabaseServiceProvider {
    public function register() {
        // Only connect when actually querying custom tables
        $this->app->singleton('db', function() {
            return new Connection($this->getConfig());
        });
    }
}
```