Project Plan

The WP Frame Studio Ecosystem

1. Project Mission & Core Philosophy

Mission: To create an enterprise-grade development framework for WordPress that provides a modern, expressive, and robust toolset inspired by Laravel, optimized for **performance and modularity** within the WordPress ecosystem.

Core Principles:

1. **WordPress-Native:** The framework is *not* a replacement for WordPress. It hooks into and enhances the existing WordPress request lifecycle, functions, and APIs.
2. **Developer Experience (DX):** Prioritize a clean, modern, and intuitive API that reduces boilerplate. Includes strong **CLI scaffolding** for quick starts.
3. **Modularity & Opt-In:** Features (Services) must be highly independent, loaded via Service Providers, and **only initialized if configured for use** (Opt-in Architecture).
4. **Security First:** All data access and manipulation must enforce WordPress best practices: **input sanitization, capability checks, and nonce verification built into the core layer.**
5. **Contract-Driven:** Core services are defined by interfaces to enable easy replacement and testing.

2. Architectural Commitments & Risk Mitigation

This section directly addresses the complexity and performance risks identified in the critique:

Risk/Area	Mitigation Strategy	Implementation Detail
Complexity / Overload	**Extreme Modularity & Opt-in:** Major features (Queue, ORM, Events) are self-contained modules loaded only when their Service Providers are	If `config/queue.php` is missing, the `QueueServiceProvider` is skipped entirely, reducing

	registered and their config files exist.	container bindings.
Performance / Load	**Lazy Loading & Benchmarking:** All Service Providers are designed for deferred loading, only initializing services when they are first called via the Facade or `app()` helper.	Initial deployment will include benchmarking to establish a low, acceptable memory footprint, especially on shared hosting environments.
WP Compatibility	**Strict Contract/Interface Adherence:** Define clear Service Contracts for core features (e.g., `OptionRepositoryInterface`, `QueueInterface`). This makes the framework testable against WP core changes.	All custom features use WordPress hooks exclusively, ensuring non-collision and proper lifecycle integration.
Maintenance/Testing	**Comprehensive Test Suite:** Implement Unit and Integration tests for every core feature, especially around security and data layers.	Use versioning (Semantic Versioning) and a clear deprecation policy documented in the README.
Developer Onboarding	**Scaffolding & Documentation:** Add `make:` commands to the CLI for rapid file generation and provide extensive examples for the most common tasks (CPT, Blocks, REST).	Theme and Plugin skeletons will include a fully functional **sample application**.

3. Comprehensive Feature List

This comprehensive list details all capabilities integrated into the framework, categorized by function, ensuring full clarity on the project's scope.

A. Core Foundation & Service Management

1. **Service Container (The True Heart):** A central Dependency Injection (DI) container (Application.php) managing service binding and resolution. (Feature 31)

2. **Core Application Class:** Manages the entire framework lifecycle and acts as the service container. (Feature 32)
3. **Central Lifecycle Controller:** A dedicated boot manager that orchestrates the loading of all core services and providers. (Feature 01)
4. **Configuration System:** Loads configuration from the config/ directory into a centralized repository, accessible via the config() helper. (Feature 02)
5. **Facades System:** Provides static, expressive access to dynamic services bound in the container (e.g., View::make()). (Feature 05)
6. **Exception & Error Handling:** A robust error handler utilizing Whoops for detailed reporting in development environments. (Feature 33)
7. **Environment Variable Loading:** Automatic loading of .env file variables using vlucas/phpdotenv.

B. WordPress API Abstractions (DX Focus)

8. **Hook Manager:** Class-based abstraction for registering WordPress Actions and Filters (add_action/add_filter) via the **Hook facade**. (Feature 03)
9. **Admin Menu Pages (Settings API):** Class-based management for registering top-level and sub-menu admin pages, routing them to Controllers. (Feature 13)
10. **Custom Post Type Manager:** Dedicated class (PostType.php) for defining CPTs, handling arguments, metaboxes, custom columns, and saving data. (Feature 12)
11. **Taxonomy Manager:** Dedicated class (Taxonomy.php) for registering and managing custom taxonomies. (Feature 14)
12. **Files Enqueue Manager:** Manages the registration and enqueueing of all scripts and styles using the **Enqueue facade**.
13. **Admin Bar Integration:** Dedicated classes within the boilerplate skeletons (AdminBar.php) to hook into and manage the admin menu bar via the Hook facade. (Feature 03 reference expanded)
14. **Class-based Shortcodes:** Simple abstraction (Shortcode.php) to define shortcode logic within a dedicated class method. (Feature 35)
15. **Class-based Widgets:** Simple abstraction (Widget.php) to define sidebar widgets using a dedicated class structure. (Feature 35)
16. **Multisite Utilities:** Provides wrappers and helpers for common multisite functions.
17. **Gutenberg Block Registration Abstraction:** Dedicated manager (BlockManager.php) for defining and registering dynamic Gutenberg Blocks using PHP for setup and Twig for server-side rendering. (Feature 38)
18. **Template Resolution Manager (Theme Only):** Service that intercepts template_include filters to map WP's template hierarchy (e.g., single.php) to the framework's Twig views (e.g., single.twig) via controller logic. (**Feature 42**)
19. **Gutenberg Editor Enhancement Manager:** Manages the registration of Block Patterns, Block Styles, and Block Editor-specific assets (using enqueue_block_editor_assets). (**Feature 43**)
20. **Option Repository:** Provides a type-safe and default-value aware layer for accessing and updating global WordPress Options, decoupled from complex admin pages.

(**Feature 44**)

C. Data, Persistence & Storage

21. **Eloquent ORM/Connection:** Uses illuminate/database to boot **Eloquent**, allowing object-relational mapping for custom and native WP tables. (Feature 08)
22. **Query Builder:** Provides the powerful DB facade for constructing complex, portable, and secure database queries. (Feature 09)
23. **Database Migrations:** A CLI-driven system for programmatic creation, modification, and destruction of custom database tables. (Feature 34)
24. **Caching Layer:** Integrates illuminate/cache, including a custom driver for seamless mapping to **WordPress Transients**. (Feature 22)
25. **File Storage Abstraction (Flysystem):** Uses the Storage facade to abstract file operations across different disks (local, S3, WP Uploads directory). (Feature 30)
26. **User Management:** Provides wrappers and Eloquent models for secure interaction with WP_User objects and authentication. (Feature 15)
27. **Session & Flash Data Management:** Dedicated service (SessionManager.php) using Database or Transients for cross-request session and **Flash Data** (e.g., "Settings saved") persistence. (Feature 39)

D. Web, API & Security

28. **Router / REST API System:** A clean, expressive router for defining custom WP_REST_API endpoints, complete with middleware support. (Feature 11)
29. **Templating / View System:** View resolution system (ViewManager.php) primarily utilizing the **Twig** templating engine for clean separation of PHP and presentation logic. (Feature 18)
30. **Sanitized Request Object:** A wrapper (WpRequest.php) that provides sanitized and validated access to user input ($_POST, $_GET, etc.). (Feature 23)
31. **Security Utilities:** Dedicated classes for handling Nonce checks, sanitization, and capability authorization. (Feature 23)
32. **Localization/Translation:** Management of translations loading from the resources/lang directory. (Feature 19)
33. **Response Handling:** Standardized response generation for API endpoints (JSON, custom headers).
34. **UI Component Abstraction:** Base classes for common UI elements like Forms, Tables, and Modals. (Feature 29)
35. **Form Request Validation:** Abstraction (FormRequest.php) allowing developers to define decoupled, authorized, and validated requests that fail before hitting the Controller. (Feature 40)
36. **Asset Manifest/Mix Helper:** Extends the Enqueue Manager to read a mix-manifest.json file, ensuring cache-busting and versioning for modern compiled assets. (Feature 41)

E. Background Tasks & Messaging

37. **Event Dispatcher:** illuminate/events integration for decoupled communication between services (e.g., triggering a background job after an event). (Feature 06)
38. **Scheduler (WP-Cron Mapping):** Maps internal Jobs/Tasks to the native WP-Cron system, providing a robust scheduling layer. (Feature 07)
39. **Jobs/Queue System:** Base classes for dispatchable jobs, designed to handle long-running processes asynchronously. (Feature 07)
40. **Mailer (SMTP):** Replaces wp_mail() with a custom implementation using Symfony Mailer, supporting SMTP configuration via config/. (Feature 24)
41. **Notification System:** A unified system for sending user notifications via various channels (e.g., Email, Database). (Feature 25)
42. **CLI Commands:** Integration of Symfony Console, allowing developers to run custom maintenance or generation commands via the **frame** executable. (Feature 10)

4. Component Architecture & Package Dependencies

- **Core (Headless) Package: ractstudio/wp-frame-studio**
  - **Role:** The engine. Contains all logic.
  - **Namespace:** RactStudio\FrameStudio
  - **Key Dependencies:**
    - illuminate/container: For the Service Container.
    - illuminate/database: For Eloquent ORM, Query Builder, and Migrations.
    - illuminate/config: For the config() helper and directory.
    - illuminate/events: For the Event dispatcher.
    - illuminate/cache: For the Caching layer.
    - illuminate/filesystem: For the Storage facade (used by Flysystem).
    - illuminate/translation: For the lang/ directory.
    - illuminate/queue: For the Jobs/Queue system.
    - illuminate/support: For Collections, Facades, and other helpers. (Used for Session/Validation too)
    - vlucas/phpdotenv: For .env file loading.
    - symfony/console: For the frame CLI command runner.
    - league/flysystem: For file storage abstraction (e.g., S3).
    - twig/twig: For the Twig templating engine.
    - filp/whoops: For developer error handling.
    - maximebf/php-debugbar: For the Debug Toolbar.
    - symfony/mailer: For the SMTP Mailer.
- **Skeletons (Plugin/Theme): WP Plugin Frame & WP Theme Frame**
  - **Role:** Boilerplates that require the core package and provide the app/, config/, resources/ directories for developers.

5. WP Frame Studio (Core Package) File Structure

This is the fully documented structure for the core Composer package (ractstudio/wp-frame-studio), which contains all the framework logic.

wp-frame-studio/

├── src/ # Source directory containing all framework classes (Namespace: RactStudio\FrameStudio)

│ ├── Foundation/ # Core classes for application boot and structure

│ │ ├── Application.php # The central Dependency Injection (DI) Container and Service Locator.

│ │ ├── BootManager.php # Orchestrates the framework initialization lifecycle (loads Config, Env, Providers).

│ │ ├── ServiceProvider.php # Abstract base class for all providers (used for binding and booting services).

│ │ └── Kernel.php # The entry point for running the application after bootstrap.

│ │

│ ├── Contracts/ # Interfaces defining core service behavior for swapability (e.g., IQueue, ICache).

│ │ ├── CacheInterface.php # Contract for all cache drivers/stores.

│ │ ├── QueueInterface.php # Contract for dispatching and processing queued jobs.

│ │ ├── SettingsRepositoryInterface.php# Contract for accessing global WP options safely.

│ │ └── StorageInterface.php # Contract for file system operations (Flysystem abstraction).

│ │

│ ├── Bootstrap/ # Classes run early in the application lifecycle to configure essential services

│ │ ├── LoadEnvironmentVariables.php # Loads variables from the `.env` file into the

environment.

│   │   ├── LoadConfiguration.php   # Reads and merges configuration files from the user's `config/` directory.

│   │   ├── RegisterCoreFacades.php   # Binds Facade accessors to their concrete Service Container implementations.

│   │   └── StartSession.php   # Initializes the SessionManager and loads existing session/flash data.

│   │

│   ├── Api/   # Handling for REST API, Routing, and Middleware

│   │   ├── Router.php   # Registers custom WP REST API endpoints with clean, expressive syntax.

│   │   ├── Request.php   # Framework representation of an incoming WP REST API request.

│   │   ├── Response.php   # Helper class for standardized API response generation (JSON, headers).

│   │   └── Middleware/

│   │   └── Middleware.php   # Base interface/class for API middleware logic (e.g., authentication checks).

│   │

│   ├── Cache/   # Implements CacheInterface contract

│   │   ├── CacheManager.php   # Resolves and manages different cache stores.

│   │   └── Drivers/

│   │   └── TransientDriver.php   # Concrete cache driver that maps to WordPress Transients.

│   │

│   ├── Cli/   # Console/Command Line Interface (CLI) tools

│   │   ├── Console.php   # Integrates Symfony Console to run developer commands via the `frame` executable.

│ │ └── Commands/ # Enhanced with Scaffolding

│ │ ├── MakeCommand.php # Base command for generating boilerplate files (Controller, Model, Provider, etc.).

│ │ └── MigrateCommand.php # Runs custom database migrations (up/down).

│ │

│ ├── Http/ # HTTP Input, Validation, and Security

│ │ ├── WpRequest.php # Wraps raw PHP request data, provides sanitation/accessors.

│ │ ├── FormRequest/ # Validation layer before hitting controllers

│ │ │ └── FormRequest.php # Base class for defining authorization and validation rules for requests.

│ │ └── Security/ # ENHANCED SECURITY LAYER

│ │ ├── Nonce.php # Helper for creating and verifying WordPress nonces.

│ │ ├── Sanitizer.php # Centralized Input Sanitization utility for common data types.

│ │ └── Authorizer.php # Utility for checking user capabilities/permissions (based on `current_user_can`).

│ │

│ ├── Queue/ # Implements QueueInterface contract

│ │ ├── QueueManager.php # Manages queue connections and job dispatching.

│ │ ├── Job.php # Base class for asynchronous, dispatchable background tasks.

│ │ └── Scheduler.php # Maps internal scheduled tasks to the WordPress WP-Cron system.

│ │

│ ├── Support/ # Helper classes, collections, and facades

│ │ ├── Facades/ # Static proxies for accessing bound services.

```
|   |   |   └── ... (All Facades remain, ensuring easy access to contracted services)
|   |   └── helpers.php                # Contains global helper functions (e.g., `app()`, `config()`).
|   |
|   ├── Wordpress/                     # Abstractions and managers for core WP features
|   |   ├── Post/
|   |   |   ├── Meta/                  # Wrapper for Post, Term, and User Metadata registration.
|   |   |   |   └── MetaFieldRegistrar.php # Handles registering custom meta fields for Gutenberg and classic editors.
|   |   |   └── PostModel.php          # Eloquent Model specifically mapped to the `wp_posts` table.
|   |   ├── Option/
|   |   |   └── OptionRepository.php   # Implements SettingsRepositoryInterface for safe, type-safe access to WP Options.
|   |   └── Template/
|   |       └── TemplateResolver.php   # Intercepts WP template loading to map to framework Controllers/Views.
|   |
|   └── Debug/                         # Development and error logging
|       ├── ErrorHandler.php           # Catches exceptions and uses Whoops for detailed reporting in dev.
|       └── Profiler.php               # Tools for benchmarking memory usage and execution time of framework services.
|
├── composer.json                      # Defines the package metadata and external dependencies.
└── README.md                          # Documentation for the core package.
```

6. WP Plugin Frame (Plugin Skeleton) File Structure

This is the fully commented boilerplate structure for a new plugin built on the framework.

my-awesome-plugin/

├── my-awesome-plugin.php # Primary Plugin Entry File. Executes the Composer autoloader and boots the framework Application.

├── composer.json # Defines plugin metadata, auto-loads the `app/` namespace, and requires `ractstudio/wp-frame-studio`.

├── frame # The executable file to run CLI commands (e.g., `php frame migrate`, `php frame make:...`).

├── .env.example # Example file for local environment variables (e.g., custom database connection details).

├── .gitignore # Specifies files and folders to ignore in version control (e.g., `/vendor`, `/storage`).

│

├── bootstrap/ # Contains the single core file to initialize the framework.

│ └── app.php # Framework Bootstrap. Creates the Application container instance and returns it.

│

├── app/ # Main Application Logic. All developer-written, PSR-4 auto-loaded classes reside here.

│ ├── Providers/ # Service Providers that register and boot application services (Opt-in).

│ │ ├── PluginServiceProvider.php# Main provider; registers fundamental plugin features (e.g., theme support, global bindings).

│ │ ├── RouteServiceProvider.php # Responsible for loading and registering all API route definitions.

│ │ ├── HookServiceProvider.php # Responsible for registering all WordPress Actions and

Filters.

│ │ ├── EnqueueServiceProvider.php# Responsible for registering all scripts and styles defined in the manifest.

│ │ └── CommandServiceProvider.php# Registers custom `frame` CLI commands implemented by the developer.

│ │

│ ├── Http/ # Classes handling incoming HTTP requests.

│ │ ├── Controllers/ # Handles the business logic for REST API and Admin Page routes.

│ │ │ ├── MyApiController.php# Example controller for a custom REST API endpoint.

│ │ │ └── Admin/

│ │ │ └── DashboardController.php # Handles view rendering and logic for an admin menu page.

│ │ └── FormRequests/ # Classes for request validation and authorization.

│ │ └── UpdateSettingsRequest.php # Example FormRequest class to validate form submission data.

│ │ └── Admin/

│ │ └── AdminBar.php # Class containing methods to add custom nodes to the WordPress Admin Menu Bar.

│ │

│ ├── Blocks/ # Gutenberg Block implementations and enhancements.

│ │ ├── CtaBlock.php # Extends the framework's `Block.php` for dynamic block rendering.

│ │ └── BlockPatterns.php # Class for registering custom Gutenberg Block Patterns and Block Styles.

│ │

│ ├── Cli/

│ │ └── MyCustomCommand.php # Example custom CLI command implementation (e.g.,

data import).

│   │

│   ├── Models/   # Eloquent ORM Models for database interaction.

│   │   ├── Book.php   # Example PostModel to interact with a custom Post Type via Eloquent.

│   │   ├── Order.php   # Example Eloquent Model for a custom database table.

│   │   └── UserProfile.php   # Example Eloquent Model for a custom user meta/profile table.

│   │

│   ├── Options/   # Option Repository definitions.

│   │   └── GlobalSettings.php # Defines the keys, casts, and default values for global WP options used by the plugin.

│   │

│   ├── PostTypes/   # Custom Post Type definitions.

│   │   └── BookPostType.php   # Extends framework `PostType.php` to define the 'book' CPT and its meta boxes.

│   │

│   ├── Taxonomies/   # Custom Taxonomy definitions.

│   │   └── GenreTaxonomy.php   # Extends framework `Taxonomy.php` to define a custom taxonomy for CPTs.

│   │

│   ├── Settings/   # Admin Page and Settings API integration.

│   │   └── PluginSettings.php # Extends framework `SettingsApi.php` to define admin menu structure and pages.

│   │

│   └── Enqueue/   # Asset definition class.

│   └── AssetManifest.php   # Defines all scripts and styles, their handles, dependencies,

and versioning.

```
|
├── config/                 # Framework Configuration files (controls Modularity/Opt-in).
|   ├── app.php               # Primary application settings, Service Provider list, and Facade aliases.
|   ├── database.php          # Configuration for custom database connections (optional - enables ORM).
|   ├── api.php               # Configuration for REST API settings (prefix, global middleware).
|   ├── cache.php             # Defines cache stores (e.g., 'transient' is the default WP store).
|   ├── filesystems.php       # Configures Flysystem disks (local, S3, 'wp_uploads').
|   ├── enqueue.php           # Global settings for asset loading behavior.
|   └── session.php           # Configuration for session drivers and expiration settings.
|
├── routes/                 # Route definition files (loaded by RouteServiceProvider).
|   └── api.php               # Defines all custom WP REST API endpoints using the `Route` facade.
|
├── resources/              # Uncompiled frontend assets and localization files.
|   ├── views/                # Twig templates for rendering admin pages and frontend views.
|   |   ├── admin/
|   |   |   └── dashboard.twig    # Template for a top-level admin menu page's content.
|   |   ├── blocks/           # Templates for dynamic block server-side rendering.
|   |   |   └── cta.twig
|   |   └── frontend/
|   |       └── shortcode-view.twig # Example template rendered by a class-based shortcode.
```

```
│   │
│   ├── lang/                  # Localization files for translations.
│   │   └── en/
│   │       └── messages.php   # PHP array returning key/value pairs for localization strings.
│   │
│   └── assets/                # Source files (e.g., Tailwind CSS source, raw JS source before compilation).
│
├── storage/                   # Writable directory for runtime data that persists between requests.
│   ├── logs/
│   │   └── frame.log          # Application log file for errors and debug output.
│   ├── cache/                 # Runtime cache files (if not using Transients).
│   └── framework/             # Internal framework runtime files.
│       ├── views/             # Compiled views cache (Twig compilation files).
│       └── sessions/          # Session storage files (if using a file-based session driver).
│
└── vendor/                    # Composer dependencies (managed automatically).
```

7. WP Theme Frame (Theme Skeleton) File Structure

This is the fully commented boilerplate structure for a new theme built on the framework, focusing on template control.

```
my-awesome-theme/
├── functions.php              # Primary Theme Entry File. Executes the Composer autoloader
```

and boots the framework Application.

├── style.css # Required WP Theme metadata file.

├── index.php # Required fallback file for the WP template hierarchy.

├── screenshot.png # Required theme screenshot.

├── composer.json # Defines theme auto-load namespace and requires `ractstudio/wp-frame-studio`.

├── frame # The executable file to run CLI commands (e.g., `php frame make:controller`).

├── .env.example

├── .gitignore

│

├── bootstrap/

│ └── app.php # Framework Bootstrap. Creates and returns the Application container instance.

│

├── app/ # Main Theme Logic. All developer-written, PSR-4 auto-loaded classes reside here.

│ ├── Providers/

│ │ ├── ThemeServiceProvider.php # Main provider: Registers theme-specific features (e.g., theme support, menus, sidebars).

│ │ ├── HookServiceProvider.php # Responsible for registering all WordPress Actions and Filters.

│ │ └── EnqueueServiceProvider.php# Responsible for registering all theme scripts and styles.

│ │

│ ├── Http/

│ │ ├── Controllers/

│ │ │ └── PageController.php # Handles logic for custom theme templates, resolving data before rendering the view.

│ │ └── FormRequests/

│ │ └── ContactFormRequest.php # Example FormRequest for validating a custom contact form submission.

│ │ └── Admin/

│ │ └── AdminBar.php # Class containing methods to add custom nodes to the WordPress Admin Menu Bar.

│ │

│ ├── Blocks/ # Gutenberg Block implementations and enhancements.

│ │ ├── PageHeaderBlock.php # Extends the framework's `Block.php` for a theme-specific dynamic block.

│ │ └── ThemePatterns.php # Class for registering custom Theme Block Patterns and Styles.

│ │

│ ├── Models/ # Eloquent ORM Models.

│ │ └── Post.php # Example PostModel for interacting with standard WP Posts.

│ │

│ ├── Options/ # Option Repository definitions.

│ │ └── ThemeSettings.php # Defines the keys and default values for global WP options used by the theme.

│ │

│ ├── PostTypes/ # Custom Post Types (if the theme registers its own CPTs).

│ │ └── ProjectPostType.php # Example CPT definition specific to the theme.

│ │

│ ├── Templates/ # Custom Template Resolution Logic.

│ │ └── SingleResolver.php # Class that hooks into WP template filters to map the

'single' hierarchy to a specific Controller/View.

│   │

│   └── Enqueue/

│   └── AssetManifest.php   # Defines all theme scripts and styles, ensuring cache-busting for compiled assets.

│

├── config/   # Framework Configuration files.

│   ├── app.php

│   ├── database.php

│   ├── api.php

│   ├── cache.php

│   ├── filesystems.php

│   ├── enqueue.php

│   └── session.php

│

├── routes/   # Route definition files.

│   └── web.php   # Defines custom frontend URL routes (if needed, e.g., non-WP-Query URLs).

│

├── resources/   # Uncompiled frontend assets and localization files.

│   ├── views/   # Twig templates for rendering the entire theme structure.

│   │   ├── layouts/

│   │   │   └── base.twig   # Main theme layout file, defining the HTML boilerplate structure.

│   │   ├── partials/

```
│   │   ├── header.twig       # Reusable header partial (nav, site branding).
│   │   └── footer.twig       # Reusable footer partial (copyright, scripts).
│   ├── blocks/
│   │   └── page-header.twig  # Template for dynamic block server-side rendering.
│   ├── index.twig            # Template for the front page or general fallback.
│   ├── page.twig             # Template for standard WP pages.
│   ├── single.twig           # Template for individual posts/CPTs.
│   └── archive.twig          # Template for archive listings.
│
├── lang/                     # Localization files.
│   └── en/
│       └── messages.php
│
└── assets/                   # Source files (e.g., CSS/JS pre-compilation).

├── storage/                  # Writable directory.
│   ├── logs/
│   │   └── frame.log
│   ├── cache/
│   └── framework/
│       ├── views/
│       └── sessions/
│
└── vendor/                   # Composer dependencies.
```

8. The Boot Process (How It All Connects)

This sequence incorporates the new features, focusing on the early initialization of the Session Manager and the immediate availability of the Request object.

1. **WP Request:** A user visits the site. WordPress boots.
2. **WP Loads Plugin/Theme:** WordPress loads my-awesome-plugin.php (or my-awesome-theme/functions.php).
3. **Framework Entry Point:** require_once __DIR__ . '/vendor/autoload.php';.
4. **Framework Bootstrap:** $app = require_once __DIR__ . '/bootstrap/app.php';.
5. **bootstrap/app.php Executes:**
    - Creates $app = new RactStudio\FrameStudio\Foundation\Application(__DIR__);
    - $app->singleton(...) Binds all core services (like CacheManager, Connection, Router) to the container.
    - $app->boot() is called.
6. **Application->boot() Executes:**
    - Runs all Bootstrap classes (loads .env, loads user config/ files).
    - **Session Initialization:** The **StartSession.php** bootstrapper runs, initializing the SessionManager and loading any existing session/flash data from the database/transients, making the Session facade available immediately.
    - Loops through config('app.providers') and registers each **User Service Provider** (e.g., PluginServiceProvider).
    - Calls the register() method on all providers.
    - Calls the boot() method on all providers.
7. **User Code Runs:**
    - Inside PluginServiceProvider::boot(), the developer's code runs, e.g.:
        - Hook::action('admin_bar_menu', [AdminBar::class, 'addItems']);
        - **Gutenberg Block Registration:** Block::register(CtaBlock::class);
        - **Asset Manifest/Enqueue:** Enqueue::register(AssetManifest::class); The Enqueue Manager now uses the new **AssetManifest helper** to resolve production-ready, versioned asset URLs.
8. **Framework Connects to WP:**
    - The **HookManager** takes all registered classes and calls the native add_action() and add_filter().
    - The **BlockManager** hooks into init and calls register_block_type() for all defined classes, pointing the render callback to the framework's View Manager (Twig).
    - The **EnqueueManager** hooks into admin_enqueue_scripts and wp_enqueue_scripts and uses the Asset Manifest to ensure correct asset loading.
9. **WP Lifecycle Continues:** WordPress continues its request cycle. If a request is handled by a controller, the framework automatically uses the new **Form Request validation** before executing the controller logic. If the request results in a redirect, the **Session Manager** handles saving the Flash Data (e.g., success message) for the next request.