GiveWP 开发者文档

GiveWP 开发者架构文档，介绍捐赠插件扩展方式。

- Testing
- Admin UI
- Bootstrapping: service providers, the container, and hooks
- Donation forms: v2 and v3
- List tables, caching, and large-data behavior
- Models, repositories, and the event model
- Payment gateways
- Donor PII: what may cross to the client

Testing

GiveWP has two test suites. They cover different failure modes and neither replaces the other.

Suite	Runs	Command	CI
PHPUnit	PHP, in-process, against the WordPress test library	`composer test`	`.github/workflows/wordpress.yml`
Playwright	A real browser against a real WordPress install	`npm run test:e2e`	`.github/workflows/tests-e2e.yml`

PHPUnit

The bulk of the coverage. Models, repositories, actions, REST routes, migrations — anything whose behavior can be asserted without a browser belongs here, because it runs in seconds and fails with a stack trace pointing at the line.

Setup and conventions are in tests/README.md.

Add-ons run this same suite against core: an add-on's `tests/bootstrap.php` requires GiveWP's autoloader from the sibling directory and hands its main plugin file to `Give\Tests\Framework\Addons\Bootstrap`, which loads the add-on on `muplugins_loaded` and then defers to core's `tests/bootstrap.php`. In CI the shared `impress-org/givewp-github-actions/.github/workflows/addon-tests.yml` workflow checks out the GiveWP branch named by its `givewp_branch` input, so an add-on can be tested against `develop` before core ships.

Playwright

E2E tests live in `tests/e2e/` and run against a WordPress install started by `wp-env`.

This suite is written for CI. It exists to catch what PHPUnit structurally cannot see: assets that fail to enqueue, React apps that fail to mount, and v3 REST requests that error only once a real browser makes them with a real nonce. Running it locally is supported and useful while writing a spec, but the environment it is designed against is the clean install the workflow builds.

Keep the specs thin. A browser test that asserts something a PHPUnit test could have asserted is a slow test with a worse failure message.

Layout

One spec file per screen, named for the screen, holding everything that screen is covered for: `donor-dashboard.spec.ts`, `reports.spec.ts`, `tools-logs.spec.ts`, `tools-migrations.spec.ts`.

New coverage goes in the file for the screen it exercises. Grouping by the kind of assertion instead — every screen's mount check in one file, every screen's REST check in another — puts one screen's coverage in as many files as there are kinds of assertion, so nothing tells you what a screen is actually covered for, and each new kind adds a file that re-lists every screen. `test.describe` already groups within a file, which is where that grouping belongs.

`admin-pages.spec.ts` is the deliberate exception. It is a breadth-first smoke suite — each admin screen that mounts a React app, one assertion each — and its job is to be pointed at environments core does not control, which is what the add-on plan below builds on. Screens with a spec file of their own are not repeated in it.

Shared helpers live in `tests/e2e/utils/`. `utils/rest.ts` records the REST calls a page makes and asserts it reached the route it owns with nothing failing. Assert on that traffic rather than on rendered records: a wrong route, a missing nonce, or a response shape the client no longer unwraps all produce a page that builds and mounts cleanly and then shows nothing.

CI

`.github/workflows/tests-e2e.yml` installs Composer and npm dependencies, runs `npm run build`, starts wp-env on the port in `.wp-env.json`, and runs the suite. Failure artifacts (screenshot, video, trace) upload on failure. Open a trace with `npx playwright show-trace <path to trace.zip>`.

Running locally

```
# First time only
npm install
composer install
npx playwright install chromium

npm run build       # the admin screens under test are React apps served from build/
npm run env:start
npm run test:e2e
```

`npm run test:e2e:headed` opens a visible Chrome window with each action slowed to 800ms. `npm run test:e2e:ui` opens Playwright's time-travel viewer, which lets you step through a run and see a screenshot at every action.

If your wp-env is not on the port in `.wp-env.json` — a `.wp-env.override.json`, a TLS proxy in front of it, `--auto-port` picking something else — point the run at it:

```
WP_BASE_URL=https://give.wpenv.net:9443 npm run test:e2e
```

Two things the setup handles for you

- `@wordpress/e2e-test-utils-playwright` reads `WP_BASE_URL` from the environment once at module load and otherwise defaults to `http://localhost:8889`, wp-env's tests environment. The `baseURL` passed to `RequestUtils.setup()` never reaches it, so REST API discovery would go to 8889 regardless of what the rest of the suite targets. `tests/e2e/environment.ts` writes the resolved URL back into `process.env` to keep the two in agreement.
- `RequestUtils` builds its own request context with no way to pass `ignoreHTTPSErrors`, so a self-signed local certificate fails the login before any spec runs — while the browser, which does get `ignoreHTTPSErrors`, would have been fine. `tests/e2e/global-setup.ts` relaxes Node's certificate check for local HTTPS runs only; CI talks to wp-env over plain HTTP with verification on.
- A WordPress install answering on the expected port is not necessarily the one under test. If another wp-env environment holds the port, every spec would run green against an unrelated site. `global-setup.ts` checks that the REST API exposes the `givewp/v3` namespace and fails with that explanation if it does not. Seeing that error means `npm run env:start` picked a different port — `env:start` passes `--auto-port` so it can step aside from a busy one — and the run needs `WP_BASE_URL`.

Writing tests

Specs use `@wordpress/e2e-test-utils-playwright`, which supplies the `admin`, `page`, and `requestUtils` fixtures. `tests/e2e/global-setup.ts` logs in once and saves the session to `artifacts/storage-states/admin.json`, so no spec pays for a login.

Assertions should not depend on the site's data. CI runs against a fresh install; a developer runs against whatever their wp-env site accumulated. Assert on structure — a root element, a page heading, a form field — not on a row count or an empty state, unless the test seeds that state itself through `requestUtils.rest()`.

Add-on smoke tests in CI

Not wired up yet. This section is the design, not a description of something that runs.

Add-ons come from their release zips, not from source

Every add-on publishes a single `<slug>.zip` asset on its GitHub release — verified on `give-recurring` 2.19.1, `give-fee-recovery` 2.3.7, `give-form-field-manager` 3.2.1, and `give-data-generator` 1.0.0. The zip is what pup built and what customers install: `vendor/` is already there, assets are already compiled, and everything is wrapped in a `<slug>/` directory, so it unzips straight into a plugins directory.

That removes the per-add-on `composer install` and `npm ci && npm run build` a source checkout would need, and it tests the artifact people actually run rather than a working tree.

```
- name: Download the latest add-on releases
  run: |
    mkdir -p addons
    for slug in give-recurring give-fee-recovery; do
      gh release download --repo "impress-org/$slug" --pattern "$slug.zip" --dir addons
      unzip -q "addons/$slug.zip" -d addons
    done
  env:
    GH_TOKEN: ${{ secrets.ADDON_READ_TOKEN }}
```

Getting them into wp-env

`.wp-env.override.json` is gitignored and outranks `.wp-env.json`, so the job writes one naming what it just unpacked. Paths are relative to the config file.

```
- name: Compose a wp-env config that includes the add-ons
  run: |
    cat > .wp-env.override.json <<'JSON'
    {
        "plugins": [".", "./addons/give-recurring", "./addons/give-fee-recovery"],
        "port": 8888,
        "testsEnvironment": false
    }
    JSON
```

`addons/` is gitignored, so the same two steps work locally when you want to reproduce a CI failure.

The blocker is a token, not the tooling

`give-recurring`, `give-fee-recovery`, and `give-form-field-manager` are private. Core's `GITHUB_TOKEN` is scoped to `impress-org/give` alone, so it cannot read their releases. `.github/workflows/zip.yml` notes that no bot token is wired into this repository, and `gh secret list --repo impress-org/give` is still empty. A fine-grained PAT or GitHub App installation token with `contents: read` on the add-on repositories is the first step.

`give-data-generator` is public, so the whole mechanism can be proven end to end with the default `GITHUB_TOKEN` before any secret exists.

What the specs should assert

A smoke test that only checks core still works with an add-on active catches most of what breaks: a fatal on activation, a JavaScript error that stops a list table mounting, an asset that 404s. The existing specs in `tests/e2e/admin-pages.spec.ts` already do all three, which is why that file stays breadth-first — pointing it at an environment with add-ons loaded is most of the value, before a single add-on-specific spec is written.

The other direction

Running this from each add-on's own repository needs no new secrets at all: the add-on's `GITHUB_TOKEN` checks out the add-on, and GiveWP core is public so the same token checks that out too. That mirrors the PHPUnit setup already in place — a reusable `addon-e2e-tests.yml` beside `addon-tests.yml` in `impress-org/givewp-github-actions`, taking the same `givewp_branch` input.

It catches an add-on change breaking against core. It does not catch a core change breaking an add-on, which is the direction that needs the token above.

Admin UI

The WordPress admin surface has three generations living side by side. Knowing which one you're in determines everything else — which framework, which data layer, and whether a fix needs applying twice.

Three generations

	Where	Status
Legacy screens	`includes/admin/`	Maintained, still selectable by users
Modern React pages	`src/<Domain>/` + `src/Views/Components/`	Where new work goes
Settings	`includes/admin/settings/`, `src/Settings/`	PHP only — no React equivalent exists

Only the four archive screens have two versions. Donations, Donors, Subscriptions, and Donation Forms each ship a modern React page *and* the legacy screen it replaced, selected per user — see list-tables.md for the switching mechanism.

Settings are a different story. There is no React settings page anywhere in the plugin. Every settings tab is PHP, built on `includes/admin/settings/class-settings-*.php` and the `abstract-admin-settings-page.php` base. `src/Settings/Security/SecuritySettingsPage.php` is modern *code* following the legacy settings *pattern* — don't read its location under `src/` as meaning it's React. Adding a settings field means working in the PHP settings API.

Registering an admin page

Modern pages are a PHP class per domain — `src/<Domain>/<Domain>AdminPage.php` — with a consistent shape:

```
public function registerMenuItem()          // add_submenu_page, capability check
public function loadScripts()               // enqueue the built bundle, localize config
public function render()                    // print the mount point for React
public static function isShowing(): bool    // are we on this page right now?
public static function isShowingDetailsPage(): bool
```

The domain's `ServiceProvider` wires `registerMenuItem()` to `admin_menu`. For the four archive screens that registration is conditional on the user's legacy preference.

`isShowing()` / `isShowingDetailsPage()` matter more than they look — they gate asset loading, and enqueueing a heavy admin bundle unconditionally is a real performance cost on every admin page.

`loadScripts()` is where localization happens, and it is subject to the rules in pii.md: pass configuration and an API root, never records.

The details page framework

`src/Views/Components/AdminDetailsPage/` is a **generic, reusable details page**, used by Campaign, Donation, Donor, Subscription, and Event Tickets. If you're building a sixth, use it — don't start from a copy of one of the five.

```
<AdminDetailsPage
    objectId={donation.id}
    objectType="donation"
    objectTypePlural="donations"
    useObjectEntityRecord={useDonationEntityRecord}
    tabDefinitions={tabDefinitions}
    breadcrumbUrl={…} pageTitle={…}
    StatusBadge={…} PrimaryActionButton={…} ContextMenuItems={…}
/>
```

What it provides, and what that implies:

- **Schema-driven validation.** It fetches the object from `/givewp/v3/{objectTypePlural}/{id}`, derives an `ajv` resolver from the REST schema, and feeds react-hook-form. Validation rules come from the API, not from the component — so a field's validation is changed on the PHP route, not in the React form.
- **Tabs** via `Tabs/Router`, `TabList`, `TabPanels`, declared through `tabDefinitions`.
- **Slot-based extension** — it renders inside a `SlotFillProvider` with a `PluginArea`, which is how add-ons inject sections without patching core.
- Error boundary, notifications store, confirmation dialog, and `AdminSection` / `AdminSectionField` layout primitives.

Data: core-data entities, and the PII split

Admin React pages don't hand-roll fetching. GiveWP registers WordPress **core-data entities** so `useEntityRecord` / `useDispatch(coreStore)` work against the v3 REST API:

```
dispatch(coreStore).addEntities([
    {name: 'donation', kind: 'givewp', baseURL: '/givewp/v3/donations', plural: 'donations',
…},
    // campaign, donor, subscription, form
]);
```

Each domain exposes a thin hook over this — `useDonationEntityRecord`, and siblings — which is what `useObjectEntityRecord` receives.

There are two entity bundles, and the difference is PII. `RegisterAdminEntities` loads `entities-admin.ts`; `RegisterPublicEntities` loads `entities-public.ts`. They register the same entities with different `baseURLParams`:

	admin bundle	public bundle
`includeSensitiveData`	`true`	`false`
`anonymousDonations` / `anonymousDonors`	`include`	`redact`

This is the admin/public boundary from pii.md enforced at the data layer, with safe defaults on the public side. Two consequences:

- **Don't load the admin entity bundle on a public page.** It requests full PII by default. The route's own permission checks are the real gate, but the request shape is wrong and the intent is wrong.
- **Don't "fix" the public bundle's params** to make a public feature show more data. Those values are the guard, not an oversight.

Shared components

`src/Admin/` is a component library, not a domain — 38 TSX files, no PHP. Look here before writing anything from scratch:

- `components/` — `Charts`, `StatWidget`, `SummaryTable`, `OverviewPanel`, `PrivateNotes`, `Notices`, `Header`, `Grid`, `Spinner`
- `fields/` — `Status`, `AssociatedDonor`, `DateTimeLocalField`, `LockedTextInput`
- `hooks/`, `providers/` (including the Emotion styles provider)

Styling uses the design system foundation tokens rather than hardcoded values — see the Styles section of AGENTS.md.

Extension surface

`AdminDetailsPage` publishes a small API onto `window.givewp.admin` for add-ons:

```
window.givewp.admin.components  // AdminSection, AdminSectionField
window.givewp.admin.hooks       // useFormContext, useFormState
```

Combined with the `PluginArea` slot, that's how an add-on adds a section to a details page. Treat both as public API — renaming or removing an export there breaks add-ons silently.

Checklist

- Which generation are you in? Settings means PHP; an archive screen means checking whether the legacy version needs the same fix.
- Building a details page? Use `src/Views/Components/AdminDetailsPage/`, don't copy an existing one.
- Need a component? Check `src/Admin/components` and `src/Admin/fields` first.
- Fetching data? Use the registered core-data entity and the domain's entity-record hook, not a bare `apiFetch`.
- Enqueuing assets? Gate on `isShowing()` so the bundle doesn't load across the whole admin.
- Localizing? Configuration only — see pii.md.
- Changing a details-page field's validation? That lives in the REST schema, not the React form.

Related

- list-tables.md — the archive screens and their legacy/modern switch
- pii.md — what may cross to the client, and the entity bundle split

- `src/API/REST/V3/Entities/` — entity registration and the two bundles
- `src/Views/Components/AdminDetailsPage/` — the details page framework
- `src/Admin/` — shared components, fields, hooks

Bootstrapping: service providers, the container, and hooks

Every feature in `src/` reaches the rest of WordPress through the same three mechanisms: a service provider registers it, the container resolves it, and `Hooks` binds it to a WordPress hook. Getting any of the three subtly wrong produces code that loads without error and simply never runs.

The boot sequence

1. `give.php` ends with `give()->boot()`, executed when WordPress includes the plugin file.
2. `Give::boot()` defines constants and registers `add_action('plugins_loaded', [$this, 'init'], 0)` — **priority 0**, so GiveWP initializes before most other plugins' `plugins_loaded` handlers.
3. `Give::init()` fires `before_give_init`, binds a couple of core classes, installs the uncaught exception handler, then calls `loadServiceProviders()`, loads form templates and routes, and fires `give_init`.
4. `loadServiceProviders()` walks `$serviceProviders` and runs the two-phase cycle below.

`loadServiceProviders()` is also called from `Give::install()` on plugin activation, and is guarded by a `$providersLoaded` flag so it runs once per request.

Service providers

A provider implements `Give\ServiceProviders\ServiceProvider` — two methods, and the split between them is load-bearing:

```
public function register()  // bind things into the container. Nothing else exists yet.
public function boot()      // wire up hooks. Every provider has registered by now.
```

`loadServiceProviders()` runs **every** `register()` first, then **every** `boot()`:

```
foreach ($this->serviceProviders as $serviceProvider) {
    $serviceProvider = new $serviceProvider();
    $serviceProvider->register();
    $providers[] = $serviceProvider;
}

foreach ($providers as $serviceProvider) {
    $serviceProvider->boot();
}
```

That's the whole reason for two phases. In `register()` you cannot rely on another domain's bindings existing yet. By `boot()`, all of them do. Resolving a cross-domain dependency in `register()` works only by accident of array order, and breaks when someone reorders the list.

The trap: the array in give.php

`$serviceProviders` is a **hardcoded, hand-maintained array** at `give.php:203`. A provider not in that array never loads. There is no autodiscovery, no directory scan, and no warning — the feature just doesn't exist, with no error to explain why.

Creating `src/MyFeature/ServiceProvider.php` is half the job. Adding it to that array is the other half, and it's the half that gets forgotten.

Two supporting details:

- The loop validates `is_subclass_of($serviceProvider, ServiceProvider::class)` and throws `InvalidArgumentException` if not, so a provider that forgets the interface fails loudly. That is the *only* failure mode that announces itself.
- The array is hand-maintained enough that `LegacySubscriptionsServiceProvider::class` currently appears in it twice.

Order matters only for `register()`. If you find yourself needing a specific position in the array, that's a sign the work belongs in `boot()` instead.

Add-ons

Add-ons don't edit the array. They call `give()->registerServiceProvider(MyProvider::class)`, which appends to it — necessarily before `plugins_loaded` priority 0 has run.

The container

`give()` returns the `Give` instance; `give(SomeClass::class)` resolves out of the container (`Give\Container\Container`, a Laravel-style container).

```
give(DonationRepository::class);            // resolve, autowiring constructor dependencies
give()->singleton('forms', DonationFormRepository::class);
give()->bind(Foo::class, fn() => new Foo());
give()->singleton(TemplateHandler::class, function () { /* … */ });
```

`Give::__call` forwards unknown methods to the container, which is why `give()->singleton()` and `give()->bind()` work even though `Give` defines neither. `Give::__get` forwards property access to `$this->container->get()`, which is how the legacy accessors resolve:

```
give()->form_meta->get_meta($formId, 'formBuilderSettings', true);
give()->payment_meta->update_meta(/* … */);
```

Constructor dependencies are autowired, so type-hinting a class is usually enough — explicit binding is for interfaces, string aliases, and anything needing construction logic.

Hooks

Use `Give\Helpers\Hooks`, not raw `add_action`/`add_filter`, when the callback is a class method:

```
Hooks::addAction('wp_head', PrintFormMetaTags::class);                 // defaults to
__invoke
Hooks::addAction('init', DonationFormBlock::class, 'register');
Hooks::addAction('give_insert_payment', CacheCampaignData::class, '__invoke', 11, 1);
Hooks::addFilter('give_donation_total', AdjustTotal::class, 'filter');
```

Three things this buys you over `add_action`:

- **Lazy instantiation.** The class is resolved from the container when the hook fires, not when it is registered. A provider's `boot()` can wire up a hundred handlers without constructing any of them.
- **Fail-fast on typos.** `method_exists($class, $method)` is checked at registration time and throws `InvalidArgumentException`, rather than producing a silent no-op at runtime.
- **A disable filter, per hook and per handler:**

  ```
  add_filter('give_disable_hook-wp_head', '__return_true');
  add_filter('give_disable_hook-init:Give\…\DonationFormBlock@register',
  '__return_true');
  ```

  Useful for third-party compatibility work and for tests that need one handler out of the

way.

Raw `add_action` is still correct for closures and plain functions.

Hooks are how modern code reaches legacy code

`includes/` is legacy and shouldn't be extended — new behavior belongs in `src/`. When that new behavior has to participate in something legacy does, the intended route is to hook a modern class onto an action or filter the legacy code already fires, rather than editing `includes/`.

`Give\Campaigns\Actions\CacheCampaignData` is a representative example: a modern action class, bound to the legacy `give_insert_payment` and `give_update_payment_status` hooks, with no change to the legacy payment code at all.

Editing `includes/` directly is still right when the bug is genuinely there. The rule is about not putting new behavior in legacy files, and not reaching for an edit when a hook is available.

Checklist for a new feature

- Provider created **and** added to `$serviceProviders` in `give.php`? (Add-ons: `registerServiceProvider()`.)
- Bindings in `register()`, hooks in `boot()` — nothing resolving another domain in `register()`?
- Class-method callbacks going through `Hooks::` so they stay lazy?
- Anything needed before `plugins_loaded` priority 0? If so, it can't live in a provider.

Related

- `give.php` — `$serviceProviders` (line ~203), `loadServiceProviders()`, `init()`, `give()`
- `src/ServiceProviders/ServiceProvider.php` — the interface, with docblocks on the two phases
- `src/Container/Container.php` — `bind`, `singleton`, `alias`, `tag`, `extend`, `make`
- `src/Helpers/Hooks.php` — `addAction`, `addFilter`, `doAction`

Donation forms: v2 and v3

Two generations of donation form live side by side in this codebase, on the same post type, with overlapping class names. Nearly every mistake in this subsystem comes from working on the wrong one. Read this before changing anything under `src/DonationForms/`.

How to tell them apart

Both are `give_forms` posts. The only discriminator is a meta key:

```
Give\Helpers\Form\Utils::isV3Form($formId);
// => (bool) give()->form_meta->get_meta($formId, 'formBuilderSettings', true)
```

A form is **v3** if it has `formBuilderSettings` meta. Otherwise it's **v2**. There is no column, post status, or taxonomy that tells you — always go through `isV3Form()`.

	v2 (legacy)	**v3 (form builder)**
Namespace	`Give\DonationForms\V2\`	`Give\DonationForms\` (root)
Model	`V2\Models\DonationForm`	`Models\DonationForm`
Model can write?	**No** — `ModelReadOnly`	Yes — `ModelCrud`
Editor	Classic WP metaboxes	React form builder (`src/FormBuilder/`)
Field storage	Individual `give_formmeta` rows	Blocks, in `formBuilderFields` meta
Settings storage	Individual meta rows	One serialized blob in `formBuilderSettings`
Front-end render	PHP templates (`src/Views/Form/Templates/`)	React app (`src/DonationForms/resources/app/`)

v2 is behind a feature flag

The v2 editor — "option-based forms" in current terminology — is gated by `Give\FeatureFlags\OptionBasedFormEditor\OptionBasedFormEditor::isEnabled()`, reading the `option_based_form_editor` setting.

The flag **self-initializes on first read**. If the option is unset, it queries whether any option-based forms exist in the database, persists the answer, and returns it:

```
$option = give_get_option('option_based_form_editor', '');

if (empty($option)) {
    $option = self::existOptionBasedFormsOnDb();
    give_update_option('option_based_form_editor', $option ? 'enabled' : 'disabled');
    return $option;
}
```

So a fresh install has it **off**, and a site upgraded from a version with v2 forms has it **on** — without anyone touching a setting. That's why a v2 code path can look dead on your local install and be live on a customer's.

What the flag actually gates, per `FeatureFlags/OptionBasedFormEditor/ServiceProvider::maybeDisableOptionBasedFormEditorSettings()`, is **admin settings visibility**: it filters v2 options out of the General, Default Options, and Advanced settings tabs, and removes the `v2` group from the payment gateways tab. `V2\DonationFormsAdminPage` also passes `isOptionBasedFormEditorEnabled` through to its UI.

It is a settings-and-affordances flag, not a kill switch — don't assume "flag off" means no v2 forms are rendering on the site.

The traps

Two classes named `DonationForm`. `Give\DonationForms\Models\DonationForm` is v3 and supports create/update/delete. `Give\DonationForms\V2\Models\DonationForm` is v2 and is **read-only** — it implements `ModelReadOnly`, so there is no `save()`. If you're reaching for a v2 write, you're on the wrong path; v2 forms are written through legacy meta functions, not the model.

Check the `use` statement before assuming which model a file is holding. Some files import both, aliased — `src/DonationForms/V2/Models/DonationForm.php` itself imports the v3 model as `ModelsDonationForm`.

v3 queries silently exclude v2 forms. `DonationFormRepository::prepareQuery()` adds `whereIsNotNull` on both the `formBuilderSettings` and `formBuilderFields` meta joins. So `DonationForm::query()` returns v3 forms only. The docblock records this as deliberate — "*@since 3.12.1 Prevent returning forms without the `formBuilderSettings` and `formBuilderFields` meta keys*" — but it means "the form exists but my query returns nothing" is almost always a v2 form hitting a v3 query.

"Legacy" means two different things. `Utils::isLegacyForm()` does *not* mean "is a v2 form" — it checks whether the form's *template* is the `legacy` template, and only v2 forms have templates at all. Templates are registered in `src/Form/Templates.php` via the `give_register_form_template` filter — `sequoia` (multi-step), `classic`, `legacy` — with the classes under `src/Views/Form/Templates/`. So

there are effectively three generations in the wild:

1. v2 form on the `legacy` template — the pre-2.7 form markup
2. v2 form on a form template — `sequoia` or `classic`
3. v3 form built in the form builder

`isLegacyForm()` distinguishes 1 from 2. `isV3Form()` distinguishes 3 from 1-and-2. They answer different questions and are not inverses of each other.

v3 writes v2-shaped meta on every save. `StoreBackwardsCompatibleFormMeta` runs on `givewp_donation_form_created` and `givewp_donation_form_updated`, projecting v3 blocks back into legacy meta keys — donation levels, goal, recurring settings. Add-ons and legacy reporting read those keys.

Consequence: if you add a v3 field that legacy code needs to see, the field alone isn't enough — the projection in `StoreBackwardsCompatibleFormMeta` has to learn about it too. And if legacy meta looks stale for a v3 form, suspect this action rather than the reader.

Where a request enters

v3, embedded in a page — the `[give_form]` shortcode (`Shortcodes/GiveFormShortcode.php`) and the donation form block both route to `BlockRenderController`. Note the shortcode returns `$output` unchanged when the form isn't v3, handing off to the legacy shortcode handler. If `BlockRenderController` renders nothing, it falls back to an iframe pointed at the view route.

v3, standalone view — `Controllers/DonationFormViewController::show()` for the real form and `::preview()` for the builder preview. Both build a `DonationFormViewModel` and render the React app. `preview()` takes blocks and settings from the request rather than the database, which is how unsaved builder state renders.

Donation submission — `Controllers/DonateController.php`, reached via `Routes/DonateRoute.php`. Routes are signed; see `Routes/DonateRouteSignature.php` and the `Generate*RouteUrl` actions in `Actions/`.

Legacy v2 submission goes through `includes/process-donation.php`, which branches on `FormUtils::isV3Form()` near the top. Same for `includes/gateways/actions.php`.

Migration (v2 → v3)

`src/FormMigration/` converts a v2 form into a v3 one. It's a pipeline (`Pipeline.php`) of steps in `Steps/`, one per concern — `DonationGoal`, `PaymentGateways`, `FormFields`, plus a step per supported add-on (`Mailchimp`, `GiftAid`, `FeeRecovery`, …). `FormMetaDecorator.php` wraps raw v2 meta access for

the steps.

If you add a v2 setting that has a v3 equivalent, it needs a migration step, or migrated forms lose it silently.

`V2/Actions/ConvertV2FormToV3Form.php` is a separate, narrower conversion used for display purposes — don't confuse it with the migration pipeline.

Relationship to campaigns

Since 4.0 every form belongs to a campaign, joined through the `give_campaign_forms` table (`src/Campaigns/Migrations/Tables/CreateCampaignFormsTable.php`). Campaigns own the goal, and `Campaigns/Actions/FormInheritsCampaignGoal.php` pushes it down to the form.

So a form's goal may not be the form's own setting. Creating a form outside a campaign context produces an orphan — see `src/DonationForms/OrphanedForms/` and `Campaigns/Actions/CreateDefaultCampaignForm.php`.

Which one do I touch?

- **Bug affecting existing sites' forms** → check whether the report is about a v2 or v3 form first. The fix location differs completely, and "works on my form" usually means the two of you tested different generations.

[Judgement] The rest of this section is inferred from the codebase's direction — v2 is read-only at the model layer, has a migration pipeline pointing away from it, and receives no new capability — not from a stated policy. On that reading: new donor-facing form features go in v3, and v2 changes are bug fixes or migration steps rather than features. Confirm with a maintainer before acting on it for anything substantial.

Related

- `src/FormBuilder/` — the React form builder that produces v3 forms
- `src/Framework/FieldsAPI/` — the field schema v3 forms compile down to (README)
- `src/Framework/Blocks/` — block collection stored in `formBuilderFields`
- `src/DonationForms/resources/app/` — the front-end React form

List tables, caching, and large-data behavior

Admin list tables are where GiveWP meets sites with hundreds of thousands of donations. Almost every performance regression in the admin has originated in a list table column. This documents the rendering pipeline, the two different caching strategies bolted onto it, and the failure modes that only appear at scale.

The archive screens have two implementations, and the user picks

Donations, Donors, Subscriptions, and Donation Forms each have **two admin implementations**: the modern React pages described below, and the legacy WordPress admin screens they replaced. Both are shipped, both are maintained, and which one a given user sees is **their own per-user setting**.

This applies to those four archive screens only — it is not a general statement about the admin. Settings, for instance, have no modern counterpart at all. See admin-ui.md for the full map.

The choice lives in user meta — `_give_{slug}_archive_show_legacy` — and is written by a REST endpoint per domain (`Donations/Endpoints/SwitchDonationView` and its siblings for donors, subscriptions, and forms, at `give-api/v2/admin/{slug}/view` with an `isLegacy` param).

Each domain's `ServiceProvider` reads it on `admin_menu` and registers the modern page **only if the meta is empty**:

```
$showLegacy = get_user_meta($userId, '_give_donations_archive_show_legacy', true);


// only register new admin page if user hasn't chosen to use the old one
if (empty($showLegacy)) {
    give(DonationsAdminPage::class)->registerMenuItem();
}
```

Two consequences worth internalising:

- **A bug report about "the donations page" is ambiguous** until you know which UI the reporter is on. The fix location is completely different, and neither implementation shares code with the other.
- **A fix applied to one is not applied to the other.** If a customer-reported issue exists in both, it needs two fixes — and if you only reproduce on the modern page, confirm the reporter wasn't on legacy before closing.

The legacy screens live under `includes/admin/`; the modern ones are the `src/<Domain>/` React pages and REST endpoints this document describes.

The pipeline

A modern list table has three parts:

1. **A `ListTable` subclass** (`src/<Domain>/ListTable/`) extending `Give\Framework\ListTable\ListTable`, declaring `ModelColumn` objects.
2. **A REST endpoint** (`src/<Domain>/Endpoints/List*.php`) that runs the query, calls `$listTable->items($models)`, and returns rows.
3. **A React page** (`src/Views/Components/ListTable/`) that fetches from the endpoint with SWR. See its README for the component props.

Donations, Donors, Subscriptions, Campaigns, Event Tickets and v2 Donation Forms each have their own set. The v2 forms list is a different, older system — see v2 forms list: async columns, not caching below.

The N+1 trap

`ListTable::items()` loops rows, and inside each row loops columns, calling `$column->getCellValue($model)`. **Any query inside `getCellValue()` runs once per row.** A column that looks up a donor name, a donation count, or a goal figure turns a 30-row page into 30 extra queries — and at 100 rows per page with several such columns, into hundreds.

This is the single most common performance mistake in this subsystem, and it doesn't show up in local testing because a dev database has twenty donations in it.

The sanctioned fix is the **bulk-data path** added in 4.0: the endpoint fetches everything for the whole page up front, hands it to `$listTable->setData($data)`, and columns read from it via the `ListTableData` trait instead of querying:

```
$column->isUsingListTableData(); // $useData, defaults to true
$column->getListTableData();     // whatever the endpoint passed to setData()
```

`CampaignsDataRepository` is the reference implementation, and its docblock says so outright: "*Used to optimize the campaigns list table performance and to avoid n+1 problems. Instead of doing expensive queries in multiple columns in each row, this class loads everything upfront for multiple campaigns.*"

If you add a column that needs data the model doesn't already carry, extend the bulk fetch. Don't query in the column.

Failures are swallowed, per row

`ListTable::safelyGetCellValue()` wraps every cell in a try/catch so one broken column can't fatal the whole table. On exception the cell renders "Something went wrong…" and writes a `Log::error()` entry.

Consequence: a column that throws produces **one log row per rendered table row**. A broken column on a 100-row page writes 100 log entries per page view, into the database. Sites have filled their log tables this way. If you're debugging slow admin pages and the log table is huge, suspect a throwing column before anything else.

Two caches, two designs

Campaigns: an options-table cache

`CampaignsDataRepository::campaigns($ids)` (4.8.0) stores aggregated per-campaign stats in two WordPress options:

- `give_campaigns_data` — `amounts`, `donationsCount`, `donorsCount`
- `give_campaigns_subscriptions_data` — the same, for recurring

Writes come from `Campaigns/Actions/CacheCampaignData`, hooked to `give_insert_payment`, `give_update_payment_status`, `give_recurring_add_subscription_payment`, and `givewp_campaigns_merged`. It dispatches through Action Scheduler (`as_enqueue_async_action('givewp_cache_campaign_data', …)`), so the refresh is asynchronous — the list table can show stale figures until the queue drains. On a site with a stuck or slow Action Scheduler queue, campaign stats simply stop updating, and nothing surfaces that as an error.

Things to know before touching this:

- **The cache is authoritative once warm.** `campaigns()` computes which ids are *not* cached, but then, if the cache holds any data at all, returns early with only what's cached — the uncached ids are never fetched on that path. Stats for a campaign missing from the cache come from `CacheCampaignData` running later, not from a read-time fallback.
- **The options are written with a plain `update_option()`**, no explicit `$autoload`. The plugin requires WP 6.6+, where WordPress applies its own autoload heuristic and keeps large values out, but a modest-sized value on a many-campaign site can still be autoloaded on every request. Check `wp_options.autoload` for these two keys before assuming.
- **Full rebuild** is the `Campaigns/Migrations/CacheCampaignsData` migration, which deletes both options and repopulates. That's the recovery path when the cache is wrong.

v2 forms list: async columns, not caching

The v2 donation forms list (`src/DonationForms/AsyncData/`, 3.16.0) took the opposite approach. Rather than pre-fetching, it renders the expensive columns — goal, donation count, revenue — as skeleton placeholders and fills them in over AJAX after page load:

```
AsyncDataHelpers::getSkeletonPlaceholder(); // <span class="give-skeleton js-give-async-data">
```

`Actions/GetAsyncFormDataForListView` serves those requests, caching each form's result in a transient `give_async_data_for_list_view_form_{formId}` for **5 minutes**.

That transient has **no invalidation** — nothing deletes it on donation insert or form update. It is TTL-only, so a new donation can take up to five minutes to appear in these columns. Either way, it reliably generates "the numbers are wrong" reports that aren't bugs.

[Judgement] Whether the missing invalidation is a deliberate trade-off or an oversight isn't recorded anywhere — nothing in the code says. Don't add invalidation on the assumption it was forgotten; ask first, since the whole point of the transient is to keep expensive queries off the page load.

Behavior is controlled by constants and a filter, all in `AdminFormListViewOptions`:

Toggle	Type	Effect
`GIVE_IS_ALL_STATS_COLUMNS_ASYNC_ON_ADMIN_FORM_LIST_VIEWS`	constant	master switch, defaults to `true`
`GIVE_IS_GOAL_COLUMN_ASYNC_ON_ADMIN_FORM_LIST_VIEWS`	constant	per-column override
`GIVE_IS_DONATIONS_COLUMN_ASYNC_ON_ADMIN_FORM_LIST_VIEWS`	constant	per-column override
`GIVE_IS_REVENUE_COLUMN_ASYNC_ON_ADMIN_FORM_LIST_VIEWS`	constant	per-column override

Toggle	Type	Effect
`givewp_use_cached_form_stats_meta_keys_on_admin_form_list_views`	filter	read stale cached meta instead of querying live

These exist because large sites need an escape hatch. A support escalation about an unusably slow forms list usually ends with one of these being set.

`FormGrid/` applies the same treatment to the front-end form grid shortcode.

Pagination cost

`List*` endpoints run two queries per page: the page of models (`limit`/`offset`) and a separate `count()` for the total (`ListDonations::getTotalDonationsCount()`). On a large donations table the COUNT with filters applied is frequently the slower of the two, and it runs on every page view, including page 400.

If you add a filter, add it to *both* `getDonations()` and `getTotalDonationsCount()` — they build their conditions through the shared `getWhereConditions()`, and bypassing it makes the total disagree with the rows.

Checklist for a new column

- Does it need data beyond what the model already loaded? → extend the endpoint's bulk fetch and read it via `getListTableData()`. Never query in `getCellValue()`.
- Can it throw? → it will be caught and logged once per row. Guard instead of relying on the catch.
- Is it expensive and non-essential? → consider the async placeholder pattern rather than blocking the page.
- Sortable or filterable? → the filter must reach `getWhereConditions()` so the count query sees it too.

Related

- `src/Framework/ListTable/` — `ListTable`, `ModelColumn`, and the `Columns`, `IsFilterable`, `ListTableData` concerns
- `src/Views/Components/ListTable/` — the React side (README)
- `src/Framework/QueryBuilder/` — what the endpoints build queries with (README)

Models, repositories, and the event model

Models are how modern GiveWP reads and writes its data, and — less obviously — they are the source of the plugin's event stream. Understanding which hooks fire from where, and which ones see everything, is the difference between a listener that works and one that silently misses half its events.

The layer

A model is a typed object over a domain record; a repository holds the SQL. Models delegate to repositories rather than querying themselves.

```
src/Framework/Models/Model.php              base class
src/Framework/Models/Contracts/ModelCrud        find / create / save / delete
src/Framework/Models/Contracts/ModelReadOnly    find / query only — no save()
src/Framework/Models/Contracts/ModelHasFactory  test factories
src/Framework/Models/ModelQueryBuilder.php  query builder returning hydrated models
```

Core models are `Donation`, `Donor`, `Subscription`, `Campaign`, and `DonationForm`, each at `src/<Domain>/Models/` with a matching `src/<Domain>/Repositories/`.

Check the contract before assuming you can write. `ModelReadOnly` has no `save()` — the v2 `DonationForm` model is the notable case, see donation-forms.md.

Prefer models over the legacy `Give_*` classes and over raw `$wpdb`. The query builder is documented in its own README.

The event model

Repositories fire the hooks, not models. Every core model has a full six-event lifecycle, dispatched from its repository's `insert()`, `update()`, and `delete()`:

```
givewp_{model}_creating   →  givewp_{model}_created
givewp_{model}_updating   →  givewp_{model}_updated
givewp_{model}_deleting   →  givewp_{model}_deleted
```

`{model}` is one of `donation`, `donor`, `subscription`, `campaign`, `donation_form`. The `-ing` events fire before the write, the `-ed` events after, and the `-ed` listeners receive the persisted model with its `id` populated.

These are the hooks new code should listen on. They carry a typed model rather than a loose array, and they're the layer the codebase is moving toward.

Dispatch goes through `Hooks::doAction()`, which calls `do_action()` and then writes a `Hook Dispatched: {name}` entry at debug level. Useful when tracing why a listener didn't run — and worth remembering that it means hook dispatch is visible in the log table.

Backwards compatibility: the legacy bridge

The legacy hooks — `give_insert_payment`, `give_update_payment_status`, and friends — predate the model layer and are relied on by add-ons, customer snippets, and code in `includes/`. They are still fired, but **not by the modern write path directly**. A set of bridge listeners re-dispatches them from the model events:

```
src/Donations/LegacyListeners/
    DispatchGivePreInsertPayment             ← givewp_donation_creating
    DispatchGiveInsertPayment                ← givewp_donation_created
    DispatchGiveUpdatePaymentStatus          ← givewp_donation_updated
    DispatchGiveRecurringAddSubscriptionPaymentAndRecordPayment
src/Subscriptions/LegacyListeners/
src/Revenue/LegacyListeners/
```

Each rebuilds the legacy payload shape — `DispatchGiveInsertPayment` assembles a `GiveInsertPaymentData` array with `price`, `formTitle`, `userInfo`, and the rest — and dispatches the old hook. The re-dispatch is marked `@deprecated` in its docblock.

The bridge is **conditional on someone listening**:

```
if (!has_action('give_insert_payment')) {
    return;
}
```

That guard is a performance measure — building the legacy payload loads the donor and derives price IDs. Don't remove it on the assumption it's redundant.

The trap: the bridge is one-way

Model → legacy only.

- A donation written through `Donation` / `DonationRepository` fires `givewp_donation_created`, and the bridge then fires `give_insert_payment`. **Both** run.
- A donation written through the legacy `give_insert_payment()` function — which uses `Give_Payment::save()` and never touches the repository — fires `give_insert_payment` only. `givewp_donation_created` does **not** fire.

The consequence is the opposite of what you'd expect from a deprecation:

> A listener on the **deprecated** `give_insert_payment` sees donations from both write paths. A listener on the **modern** `givewp_donation_created` sees only those written through the model.

This is why `Campaigns/Actions/CacheCampaignData` is bound to `give_insert_payment` and `give_update_payment_status` rather than the model hooks — it needs complete coverage of donation events. That's a deliberate choice for correctness, not an un-migrated leftover. Don't "modernize" a legacy listener to a model hook without first confirming nothing writes that record through the legacy path.

Direction of travel

[Judgement — confirmed with maintainers] Model hooks are the target. The intent is to converge on them completely **when coverage allows** — that is, as legacy write paths are retired or routed through repositories, so that a model hook genuinely sees every event. Until then the legacy hooks stay, and stay fired.

Practically, for new work:

- **New listeners go on model hooks** — unless you specifically need coverage of a legacy write path, in which case use the legacy hook and note why.
- **Don't add new listeners to legacy hooks casually**, and don't fire new legacy-shaped hooks.

- **Don't remove bridge listeners.** They're what keeps add-ons and customer snippets working.
- **Write new data through models and repositories**, never through `Give_Payment` and friends. Every write that goes through the repository is one more event the model hooks can see, which is how coverage improves.

Checklist

- Listening for a donation/donor/subscription event? Model hook by default; legacy hook only if you need coverage of legacy writes, with a comment saying so.
- Writing data? Through the model/repository, so the model events fire.
- Touching a `LegacyListeners/` class? Understand which model hook feeds it before changing it.
- Model needs a new field? The repository's `insert()`/`update()` and the meta mapping both need it — and check whether a ViewModel exposes it (pii.md).
- Assuming you can `save()`? Check whether the model implements `ModelCrud` or `ModelReadOnly`.

Related

- `src/Framework/Models/` — base class, contracts, query builder integration
- `src/Framework/QueryBuilder/` — README
- `src/<Domain>/Repositories/` — where the events are dispatched
- `src/Donations/LegacyListeners/` — the bridge
- donation-forms.md — the two `DonationForm` models and their contracts

Payment gateways

The Gateway API connects donation forms to payment processors. It has a PHP half (`src/Framework/PaymentGateways/`) and a JavaScript half (a registered gateway object on the v3 form), and the boundary between them is a security boundary. Most of this document is about that boundary.

The rule

Funds are never collected on the client. The capture happens server-side, or it doesn't happen.

This is the anchor. The client is allowed to set up a payment — build an order, tokenize a card, collect an authorization from the donor. What it must never do is complete the transaction. The step that actually moves money runs on the server, inside the gateway's `createPayment()`, after the form has been validated and the payment object has been checked against what the server expects.

Two supporting rules follow from it:

- **Nothing reaches the processor until the server has validated the complete form.** Client-side validation is UX. A browser can skip it, and several field types have rules that exist only server-side.
- **Data arriving from the processor is a claim, not a fact.** Whether it's an order the client handed you or a webhook the processor sent you, verify it against server-held state before acting.

All three have been the subject of fixes. None is theoretical.

The lifecycle

A v3 form donation runs through these steps:

1. `beforeCreatePayment(values)` — JS, optional. Prepares client-side state and returns extra data to merge into the request. **This is not a validation gate.**
2. The form POSTs to the donate route (`Routes/DonateRoute.php`, signed — see `DonateRouteSignature`), which runs full server-side validation and then calls the gateway's PHP `createPayment()` / `createSubscription()`.

3. `afterCreatePayment(response)` — JS, optional. Completes whatever the server started, using values the server returned.

Step 2 is where money moves. Whatever the client did in step 1, the transaction is completed by the PHP gateway during the server-side donate request.

Stripe: `beforeCreatePayment` calls `elements.submit()` to collect card data and returns settings. The server validates the form and creates the PaymentIntent, returning a `clientSecret`. `afterCreatePayment` calls `stripe.confirmPayment()` with that secret — confirming an intent the server created and owns.

PayPal Commerce: the order is created and approved through PayPal's client checkout flow, and the donor authorizes it in PayPal's own UI. **None of that captures funds.** The capture happens in `PayPalCommerce::createPayment()` on the server — see below.

When the SDK owns the button

PayPal Commerce can't use the flow above, because PayPal's own React buttons intercept the click and open PayPal's popup directly. Left alone, the donor would be inside PayPal's checkout before GiveWP's form validation ever ran — and an order would exist on PayPal's side for a donation the server would have rejected.

So the gateway rebuilds the gate manually, in the buttons' `onClick`, before allowing PayPal to proceed (`src/PaymentGateways/Gateways/PayPalCommerce/payPalCommerceGateway.tsx`):

1. Check gateway/subscription compatibility → `actions.reject()`
2. Client-side validation via react-hook-form `trigger()` → `actions.reject()`
3. **Server-side validation** — `handleValidationRequest()` POSTs the whole form to `DonationForms/Routes/ValidationRoute.php` → `actions.reject()`
4. Only then `actions.resolve()`

The in-code comment explains why step 3 is not redundant with step 2, and it's worth repeating because it's the part that looks removable:

> *Ideally, the client-side validations should be enough. However, in some cases, these validations are reached later when the donation is already created on the PayPal side... #1 - Billing Address Block: depending on the selected country, the city, state, and zip fields can be required or not and there are custom validation rules on the server side. #2 - Gift Aid Block: when users opt-in... required fields... the validation rules for it live only on the server.*

Only then does PayPal's SDK invoke its `createOrder` callback. That callback doesn't call PayPal directly either — `createOrder.tsx` POSTs the complete form values plus the gateway's own keys to the `give_paypal_commerce_create_order` AJAX action, and `AjaxRequestHandler::createOrder()` makes the API call and returns an order id. The card-fields flow (`beforeCreatePayment` → `cardFieldsForm.submit()`) reaches the same handler without passing through the `onClick` gate, and the action is registered on `wp_ajax_nopriv` behind a page-rendered nonce, so the handler cannot rely on anything upstream having validated the request. It does not trust it:

- `validateFrontendRequest()` — form id and donation form nonce
- `ValidateDonationFormRequest` (`src/DonationForms/Actions/`) — the posted values are held to the form's own rules through the same validator the validate route uses (v3: the fields-API rules on every posted field — amount minimum/maximum when custom amounts are on, required fields, honeypot; v2: `give_donation_form_validate_fields()` followed by `give_checkout_error_checks`, the same sequence `give_process_donation_form()` runs, which is where the level check and add-on rules hang). The gateway never learns form configuration; it asks the form layer and acts on the verdict. This is the seam to reuse if another gateway ever has to call its processor before the donate route runs.
- for v3 forms, the total sent to PayPal (`give-amount`, fee recovery included) must be at least the `amount` the form validated; `createPayment()` reconciles the order to the donation before capture, so that is the only bound the handler needs
- for v2 forms, the final amount after the `give_donation_total` filter is checked again: positive and within the form's configured maximum

The donor then approves the order inside PayPal's UI. **At this point no money has moved.** An approved PayPal order is an authorization, not a payment.

Where the funds are actually collected

`onApprove` stores the order id and clicks the form's real submit button, which runs the normal signed donate route → full server-side validation → `PayPalCommerce::createPayment()`.

That method is where money moves, and it verifies before it does:

1. Fetch the order from PayPal by id — the server asks PayPal for the order's state rather than trusting what the client reported.
2. Branch on status. `COMPLETED` means it was already captured; `APPROVED`/`CREATED` means it still needs capturing. Anything else throws.
3. `validate3dSecure()`, and `updateOrderFromDonation()` if `shouldUpdateOrder()` finds the order no longer matches the donation.
4. `approveOrder($payPalOrderId)` — despite the name, this issues PayPal's `OrdersCaptureRequest`. **This is the capture.**
5. `validatePayPalOrder()` — the capture exists, isn't `DECLINED`, and carries no error details.
6. Return `PaymentComplete::make($transactionId)` using the real capture id.

So the client's entire contribution is an order id. Everything that binds that id to money happens server-side, after validation, with the order re-fetched from PayPal rather than taken on the client's word.

The v2/legacy path still exposes capture through the `give_paypal_commerce_approve_order` AJAX action (`AjaxRequestHandler::approveOrder()`) — a client-triggerable capture endpoint, registered on both `wp_ajax` and `wp_ajax_nopriv`, that the v2 scripts (`SmartButtons.js`, `AdvancedCardFields.js`) call right after PayPal's `onApprove` and before the form is submitted. So on v2, capture happens before the donation record exists and before `give_process_donation` has run. `approveOrder()` and `updateOrderAmount()` refuse v3 forms outright, which is what makes the rule above hold without exception on v3. For v2 they run the legacy validator (`give_donation_form_validate_fields()`, through `ValidateDonationFormRequest`) first, but they still capture.

[Judgement] `createPayment()` already handles `APPROVED` orders and already receives `payPalOrderId` from the legacy processing path, so it is the natural home for v2 capture too. Until that moves, treat the v3 shape as the model and the v2 endpoint as legacy.

Don't generalize the PayPal shape

`handleValidationRequest` is called from exactly two places: the PayPal Commerce gateway and the multi-step form's `NextButton`. It is a workaround for SDKs that seize the submit, not the house style. A gateway that submits through the normal donate route already gets server-side validation for free and should not add a second round trip.

Webhooks and inbound data

Anything the processor sends you is attacker-reachable. `PayPalStandardWebhook::verifyEventData()` is the reference for what verification means in practice — it checks, before processing:

- **merchant identity** — `receiver_email` matches the site's configured account
- **amount and currency** — match the stored donation record
- **record linkage** — for refunds and reversals, the parent transaction id matches the donation

Verifying the notification's *signature* is not sufficient. A signature proves the message came from the processor; it does not prove the message is about a donation on this site, for this amount, in this currency. Check the payload against your own records.

Stripe webhook listeners live in `Gateways/Stripe/StripePaymentElementGateway/Webhooks/Listeners/` — one class per event.

Gateway route methods reached by redirect are signed: use `generateSecureGatewayRouteUrl()` rather than `generateGatewayRouteUrl()` for anything that mutates state, and see `supportsMethodRoute()` and `callRouteMethod()` for how they dispatch.

Structure

- `src/Framework/PaymentGateways/` — `PaymentGateway` abstract class, `PaymentGatewayInterface`, `PaymentGatewayRegister`, `SubscriptionModule`
- `src/PaymentGateways/Gateways/TestGateway/` — the minimal implementation, start here
- `src/PaymentGateways/Gateways/Stripe/StripePaymentElementGateway/` — full-featured reference
- `src/PaymentGateways/Gateways/PayPalCommerce/` — the SDK-owns-the-button case
- Subscription modes: `src/Subscriptions/ValueObjects/SubscriptionMode.php`

Capability flags on the abstract class tell the rest of the plugin what a gateway can do — `supportsSubscriptions()`, `supportsRefund()`, `supportsFormVersions()`, `canPauseSubscription()`, `canUpdateSubscriptionAmount()`, and friends. Declare them honestly; the admin UI and the subscription code branch on them.

Public docs: https://givewp.com/documentation/developers/how-to-build-a-gateway-add-on-for-givewp/

Checklist for a new or changed gateway

- **Can the client complete a transaction on its own?** If any code path lets the browser capture, charge, or otherwise finalize payment without the server, that's the bug. Setup and authorization on the client are fine; collection is not.
- Does the server re-fetch the payment object from the processor before capturing, rather than trusting the id and amounts the client reported?
- If the SDK takes over the submit button, does the pre-flight run server-side validation, not just client-side?
- Does the webhook handler verify merchant identity, amount, currency, and record linkage — not just the signature?
- Are state-changing gateway routes generated with `generateSecureGatewayRouteUrl()`?
- Are the `supports*` / `can*` flags accurate?
- Is anything sensitive being logged? `Log::redact()` strips keys containing `card`, `password`, `secret`, `token` (filterable via `give_log_redaction_list`) — but only keys it recognizes.

Donor PII: what may cross to the client

This document exists because donor emails once reached the page's JavaScript context. That class of failure is easy to reintroduce, because the data is legitimately present server-side at every step — the leak is never a query that shouldn't have run, it's an object that travelled one layer further than it needed to.

The rule: donor data is exposed only where someone with authority chose to expose it, through an output shape built for that purpose. The default has to be exclusion, not omission — "nothing currently prints it" is not a boundary.

PII here means at minimum: email address, physical address, phone, IP, full name in a context where the donor chose anonymity, and any custom field a form collected.

Why this is not a blanket rule

What the product does — verifiable from the code: GiveWP ships public donor display, and has for a long time.

The **Donor Wall** (`[give_donor_wall]`, `includes/donors/class-give-donor-wall.php`) is the canonical case. Customers put it on public pages specifically to show off who is giving, and its shortcode attributes are the consent model in miniature — `anonymous`, `show_name`, `show_avatar`, `show_company_name`, `show_total`, `show_comments`, `show_tributes`, `show_time`. Every one is a site owner's decision about what appears.

The newer `CampaignDonors` and `CampaignDonations` blocks do the same thing with block attributes (`showAnonymous`, `showAvatar`, `showCompanyName`, `sortBy`, `donorsPerPage`).

This is intended, configured behavior across both the legacy and modern stacks. A rule of "no donor data on public pages" would flag two of the plugin's long-standing features as bugs, and an agent applying it would break them.

Why it's built that way — **[Judgement]**, a design position held by the maintainers rather than a neutral fact: fundraising is treated as a different category from traditional finance. Donors frequently *want* to be seen giving; the public donor list is understood as a feature of the category, as on GoFundMe, so a donor's name on the front end is often the point rather than a leak. This is informed by domain experience and it drives the current design, but it is a judgement call and a

reasonable person could weigh it differently. It's recorded here because it explains decisions you will otherwise read as carelessness — not because it settles the argument.

The rest of this document follows from the first paragraph, which holds regardless of whether you accept the second.

The actual invariant is consent, and it has two layers that both have to hold:

1. **The donor's choice.** A donation carries an anonymous flag. Where it's set, that donor's identity is not displayed regardless of configuration.
2. **The admin's configuration.** Displaying donors at all is something a site owner opts into and tunes, via shortcode attributes on the Donor Wall or block attributes on the campaign blocks. Nothing about a donor appears publicly because code decided it should.

GiveWP never exposes PII without admin consent. So the question for any new surface is not "is this data sensitive?" but "who decided this would be visible, and where is that decision recorded?" If the answer is "nobody, it's just what the query returned," that's the bug — whether or not the field looks sensitive.

Note the asymmetry: the two consent layers cover *identity display* — name, company, avatar. They have never been a licence for contact details. Email, address, phone, and IP have no display configuration, and adding one would be a product decision rather than a code change.

The v3 REST API is the intended data layer

The v3 REST API (`src/API/REST/V3/`) was built to be the **single way data is fetched, for both admin and public consumers** — blocks and shortcodes included, not just the admin React apps. That is the direction of travel, and new work should follow it.

Admin surfaces should essentially always go through REST. There is no reason for an admin page to inline records into HTML when it can fetch them behind a permission check.

Public surfaces are less settled. Some render server-side today; some fetch. Either can be correct for now, but the public side carries the higher cost of a mistake, so the bar is: be deliberate about what reaches the page, and prefer fetching over inlining when there's a choice.

The split is already enforced at the data layer for anything using core-data entities. GiveWP ships two entity bundles — `entities-admin.ts` and `entities-public.ts` — registering the same entities with different defaults:

	admin	public

`includeSensitiveData`	`true`	`false`
`anonymousDonations` / `anonymousDonors`	`include`	`redact`

Those values are the guard. Don't load the admin bundle on a public page, and don't loosen the public one's params to make a public feature show more — that's the decision this document says belongs to an admin setting, not to code. See admin-ui.md.

An open question, deliberately unresolved

Public routes currently return partial PII in places, and whether that is the right long-term shape is **an open design question, not a settled decision**. One direction under consideration is modelling admin consent in the REST API itself, so that what a public route may return is driven by configuration rather than by each route's own judgement.

Treat this as unfinished. If you're working near it: don't "fix" a public route's PII handling by tightening it unilaterally — you may break a configured, intended display. And don't widen one on the grounds that adjacent fields are already exposed. Raise it instead.

Three ways data reaches the client

Each has its own boundary, and they fail differently.

1. Localized script data (`wp_localize_script`)

This is the vector with the worst blast radius — whatever you pass is inlined into the HTML source of the page, readable by anyone who can view it, cached by anything that caches pages.

The rule: localize configuration, never records. The list tables are the pattern to copy — `LoadDonorsListTableAssets` passes an API root, a nonce, column definitions, and admin URLs. It does **not** pass a single donor row. The React app fetches rows at runtime through the REST API, where permission checks apply per request.

```
wp_localize_script($handleName, 'GiveDonors', [
    'apiRoot'  => esc_url_raw(rest_url('give-api/v2/admin/donors')),
    'apiNonce' => wp_create_nonce('wp_rest'),
    'table'    => give(DonorsListTable::class)->toArray(),  // column schema, not data
    // …
```

```
]);
```

If you find yourself localizing an array of donors, donations, or subscriptions, stop — that is the incident shape.

Admin and public localization are separate classes on purpose. Campaigns has `LoadCampaignAdminOptions` and `LoadCampaignPublicOptions`, and the public one is deliberately minimal — currency and an `isAdmin` flag. Don't merge them, and don't add a field to the public one without asking who can read the page it lands on.

2. The REST API

PII leaves through a **ViewModel that has been told what the caller may see**. Never serialize a model straight into a response.

```
(new DonorViewModel($donor))
    ->anonymousMode($donorAnonymousMode)   // exclude | include | redact
    ->includeSensitiveData($includeSensitiveData)
    ->exports();
```

- `DonorAnonymousMode` / `DonationAnonymousMode` are three-state, not boolean: `exclude` drops the record, `include` returns it as-is, `redact` returns it with identifying fields removed. Picking the wrong one is a silent leak.
- `includeSensitiveData` gates custom fields — the ones a form collected, whose contents nobody audited.
- Permission checks live in `permission_callback` and `src/API/REST/V3/Routes/*/Permissions/`.

A new field on a ViewModel is a disclosure decision. Ask which callers reach that route unauthenticated before adding one.

3. Server-rendered templates

The subtlest of the three, because nothing looks like serialization.

The public `CampaignDonors` and `CampaignDonations` blocks select `donors.email` in their queries. That is legitimate — the email is needed to derive a Gravatar URL. But `CampaignDonorsBlockViewModel::formatDonorsData()` **mutates and returns the same objects** rather than building a clean output shape, so every donor object handed to the template still carries `->email`.

The template doesn't print it, so nothing leaks today. But the only thing standing between a public page and a list of donor emails is that template's discipline. Any of these reintroduces the incident:

- a `wp_json_encode($donors)` added for a JS-driven variant of the block
- a debug `var_dump` that survives review
- a third party hooking a filter that receives the donor array
- someone copying the ViewModel as the basis for a new block

When a query pulls PII for a derived value, drop it before the data leaves the ViewModel. Build the output array explicitly; don't hand the query rows onward.

Gravatar URLs contain a hash of the email

`get_avatar_url($entry->email)` produces a Gravatar URL containing the MD5 of the donor's email address, and that URL is rendered on a **public** campaign page for every non-anonymous donor shown. Email hashes are reversible for common addresses via precomputed tables, so this is a weaker disclosure than plaintext but not a non-disclosure.

Anonymous donors are handled correctly — `get_avatar_url(0)` is used, which involves no email. The exposure is limited to donors who did not choose anonymity, which is a defensible position, but it should be a decision someone made rather than a side effect of using the WordPress default.

Logs

`Log::redact()` replaces values whose **key** contains `card`, `password`, `secret`, or `token` (filterable via `give_log_redaction_list`) with `[[redacted]]`.

Note what that does and doesn't cover: it matches on key names, not values, and the default list contains nothing about donors. Logging `['donor' => $donor]` or `['formData' => $data]` writes whatever those hold straight into the log table, which is readable in the admin and included in support exports. `ValidationRoute::logError()` logs `$formData->toArray()` on validation failures — worth knowing before adding fields to that DTO.

Checklist

- **Who decided this would be visible?** If the answer is "nobody, it's what the query returned," that's the bug — regardless of how sensitive the field looks.
- Building an admin surface? Fetch through the REST API rather than inlining records.

- Localizing anything? Is it configuration, or is it records? Records belong behind the REST API.
- Adding a field to a public localization or a public block? Who can read that page, and which admin setting turns it on?
- Identity display (name, company, avatar) can be admin-configured. Contact details (email, address, phone, IP) have no such setting, and adding one is a product decision, not a code change.
- New REST field? Does it pass through a ViewModel, and is it gated on permission and anonymous mode?
- Does a query pull PII for a derived value (avatar, hash, comparison)? Drop it before the data leaves the ViewModel — don't pass query rows to a template.
- Does the donor's anonymity choice reach every place their data is rendered?
- Logging a model, a DTO, or a request payload? Redaction matches key names only, and knows nothing about donors.

Related

- `src/API/REST/V3/Routes/*/ViewModels/` — the REST output boundary
- `src/Donors/Actions/LoadDonorsListTableAssets.php` — the localization pattern to copy
- `src/Campaigns/Actions/LoadCampaignPublicOptions.php` — the deliberately minimal public surface
- `src/Campaigns/Blocks/CampaignDonors/` — the near-miss described above
- `includes/donors/class-give-donor-wall.php` — the Donor Wall, the long-standing public donor display and the clearest example of the consent model
- `src/Log/Log.php` — `redact()` and the redaction list