Docs

Symfony Bundles

Symfony Bundles

Anonymous · 24/08/2026

Table of Contents

Symfony Bundles

Symfony Bundles

Anonymous · 24/08/2026

Symfony Bundles

Last updated on 24/08/2026

Platform Bundle

Platform Bundle

Anonymous · 24/08/2026 · #php, #symfony

Platform Bundle

Last updated on 24/08/2026

Introduction

What this bundle does, its modules, architecture, and mental model.

Anonymous · 24/08/2026

Introduction

Derafu Platform Bundle is a Symfony bundle that provides the foundational infrastructure for SaaS applications: user management, authentication, multi-tenant organizations, a plugin/app ecosystem, and an event-driven notification system. It supplies reusable building blocks that every SaaS needs — without imposing opinions on your business logic.

Modules

The bundle is organized into six modules, each independently useful but designed to work together:

Module	What it provides
Core	Base resource abstraction, seeders, registry, settings infrastructure
Identity	Users, roles, permissions, organizations, teams, API keys, JWT, 2FA
IdentityUi	Ready-made web UI for auth flows and account settings
Apps	Plugin/integration ecosystem with per-user and per-organization installations
Notifications	Event-driven notifications via email, in-app, and webhooks
Api	API Platform integration helpers (query string adapters, scope enforcement)

Architecture pattern

Every ORM relation in the bundle targets an **interface**, not a concrete class. Doctrine resolves interfaces to the app's concrete entities at runtime via `resolve_target_entities`, wired automatically by the bundle's DI extension. The bundle has zero knowledge of your `App\Entity\*` classes.

The entity pattern is **MappedSuperclass + ResolveTargetEntityListener**:

```
Bundle                                  App
──────                                  ───
BaseUser (MappedSuperclass)  ◄──  User (#[ORM\Entity])
UserInterface ──────────────────►   resolved to User at runtime
```

All public service contracts live under `Contract/{Entity,Service,Integration}/` folders. Services are `final` with companion interfaces — consumers inject the interface, not the class.

Mental model

```
User alice@example.com
 ├── personal resources (UserOwnedInterface)
 └── member of Organization "acme"
      ├── OrganizationMembership (role = org.owner)
      ├── shared resources (OrganizationOwnedInterface)
      ├── Team "backend" (subset of members → resource access)
      └── other invited members
```

App entities (Project, Invoice, Document, …) integrate with the bundle by implementing one or more interfaces. The bundle provides traits with ready-made Doctrine mapping.

What it is

- A **MappedSuperclass** bundle — the bundle owns the ORM schema; the app owns the concrete entity classes, table names, and extra fields.
- A **cascading RBAC engine** — permissions are checked at four levels: resource → team → organization → global, with automatic fallback.
- A **user lifecycle toolkit** — registration, email verification, password reset, email change, magic links, sudo mode, and lockout.
- A **multi-tenant membership system** — GitHub-style organizations with roles, teams, and invitations.
- A **plugin/app system** — apps installable per user, per organization, or per resource, with encrypted configuration storage.
- A **notification system** — event-defined notifications delivered via email, in-app messages, or webhooks, with per-user preferences.
- A **built-in web UI** — controllers, templates, and routes for all auth and account-settings flows (IdentityUi module).

What it is not

- Not a replacement for Symfony Security — it builds on top of it.
- Not a JWT library — it wraps Lexik JWT for convenience.
- Not a 2FA solution — it integrates Scheb 2FA Bundle.
- Not an OAuth or social-login solution.

Last updated on 24/08/2026

Features

Everything the bundle provides, grouped by module.

Anonymous · 24/08/2026

Features

A reference of every capability the bundle provides, grouped by module. Each item links to its dedicated page where applicable.

Core

The foundation shared by all other modules.

- **Base resource abstraction** — `BaseResource` MappedSuperclass with `id`, `uuid` (v7), `slug`, owner, optional organization, and timestamps. App entities extend it to inherit the standard shape.
- **Seeder pattern** — `SeederInterface` + `derafu:platform:seed` command. Seeders are auto-discovered via service tags; run them at deploy time or via cron.
- **Resource registry** — `ResourceRegistryInterface` collects every app resource that implements `ResourceDescriptorInterface`, used by the Apps and Notifications modules to know which entities support integrations or webhooks.
- **Settings section registry** — `SettingsSectionProviderInterface` lets any module contribute sections to the settings sidebar; IdentityUi, Apps, and Notifications each register theirs automatically.

Identity

User management, authentication, organizations, and authorization. See User lifecycle, Organizations, and Authorization for full details.

Users and authentication

- **User entity** — UUID v7, `email` as login identifier, hashed password, `active` flag, `emailVerifiedAt`, and a free-form `config` JSON column for per-app data. Implements Symfony's `UserInterface`.
- **Registration with email verification** — `Registrar` creates the user, issues a single-use `email_verify` token, and dispatches `UserRegistered` with the plaintext token. The bundle stays transport-agnostic; the app's listener sends the email.
- **Password reset** — `PasswordResetter` handles request and confirm phases. Silent no-op when

the email is not registered (anti-enumeration). Dispatches `PasswordResetRequested` and `UserPasswordChanged`.
- **Email change** — `EmailChanger` binds the new address to the token payload; the current email is not touched until the token is consumed. Dispatches `EmailChangeRequested` and `UserEmailChanged` (with both old and new addresses).
- **Magic links (passwordless login)** — `MagicLinkManager` issues a short-lived token; `MagicLinkAuthenticator` consumes `_magic_link_token` from the query string and authenticates without a password.
- **Secure one-use tokens** — All lifecycle tokens use 256 bits of entropy, SHA-256 hashed at rest, `hash_equals` for timing-safe validation, configurable TTL per type.
- **JWT tokens** — `JwtTokenManager` wraps Lexik JWT behind a stable bundle boundary; issued tokens are persisted for audit and revocation.
- **Locale and timezone** — `Locale` and `Timezone` entities store supported options; `LocaleResolver` resolves the user's preferred locale from the configured list.

Organizations and teams

- **GitHub-style organizations** — `Organization` entity with UUID, no hardcoded owner column. The owner is the membership with `role.code = 'org.owner'`, which lets ownership transfer be a role swap.
- **Organization memberships** — `OrganizationMembership` pivot (user + organization + role). Unique constraint `(user, organization)`. `OrganizationManager` handles create, addMember, removeMember, and transferOwnership atomically.
- **Organization invitations** — `InvitationManager` issues SHA-256 hashed, email-bound invitations. Accept validates email match (case-insensitive) and creates the membership transactionally. Invitations are retained as audit trail.
- **Teams** — Named groups of org members with a URL-friendly `slug`. Assign a team to a resource with a role and all members inherit access. `TeamManager` handles create/addMember/removeMember with org-membership validation.
- **Ownership transfer** — Atomic swap of the owner role to a new member with caller-controlled demotion of the previous owner.

Authorization (RBAC)

- **Four-level cascading permission checker** — `PermissionChecker` resolves: resource-access → team-access → organization-membership → global. A super-admin passes every check without explicit grants. See Authorization.
- **Symfony voters (auto-registered)** — `PermissionVoter` (global), `OrganizationPermissionVoter` (org subject), `ResourcePermissionVoter` (resource subject). All add human-readable deny reasons to Symfony's Vote for Web Profiler debugging.
- **API Platform integration** — Voters fire via `is_granted('read', object)` with no extra configuration. The Doctrine tenant filter applies to listing operations automatically.

Security

- **Sudo mode** — Session-based re-authentication window (default 15 min). Auto-granted on every login; no DB column. `SudoManager` checks and refreshes the window.
- **API keys with scopes** — Long-lived `dik_` tokens; SHA-256 hashed at rest. `ApiKeyAuthenticator` fires on `Authorization: Bearer dik_...`. `#[RequiresScope]` enforces per-endpoint scope checks. Session users bypass scopes entirely.
- **Rate limiting per API key** — Fixed-window algorithm with GitHub-style `X-RateLimit-*` headers. Disabled by default. PSR-6 default implementation; a Redis atomic-`INCR` variant is also provided.
- **Account lockout** — Automatic lock after N failed login attempts (default 5, 30 min lock). `AccountLockListener` hooks into Symfony's authentication events; zero wiring needed.
- **Login history** — Every successful login is recorded (IP, User-Agent, authenticator). `LoginHistoryManager` provides queries for a security-activity page; `identity:login-history:purge` prunes old records.
- **Impersonation audit** — `ImpersonationAuditListener` records every `switch_user` session (admin, target, IP, start/end time). Fully automatic.
- **Two-factor authentication** — TOTP-based 2FA via Scheb 2FA Bundle, with QR code enrollment. `TwoFactorEnrollmentManager` manages enable/disable flows.

Identity UI

Ready-made controllers, routes, and Twig templates. All templates extend your `parent_layout` and can be overridden individually. See Identity UI.

- **Auth pages** — Login, register, password reset, magic link request, email verification wall, sudo confirmation, 2FA code entry.
- **Account settings** — Name, avatar, locale, timezone, password change, email change, active sessions, login history, API keys, JWT tokens.
- **Organization settings** — Create orgs, manage members, invitations, teams.
- **Locale negotiation** — `LocaleNegotiationListener` negotiates the user's locale from the configured list on every request.
- **Email verification enforcement** — `require_email_verification: true` redirects unverified users to the verification wall automatically.
- **Built-in transactional emails** — `IdentityMailerListener` sends verification, password reset, invitation, and other identity emails out of the box.

Apps

Plugin/integration ecosystem. See Apps.

- **App definitions** — Apps are services implementing `AppDefinitionInterface`, auto-discovered via service tags. Each declares code, name, category, icon, and supported scopes.

- **Three installation scopes** — Per-user (`BaseUserAppInstallation`), per-organization (`BaseOrganizationAppInstallation`), per-resource (`BaseResourceAppInstallation`).
- **Encrypted configuration** — `AppConfigEncryptor` transparently encrypts `sensitiveConfigKeys` at rest in the installation's JSON column.
- **Capability system** — Apps implement capability interfaces; `CapabilityRegistryInterface` answers "which installed apps for this org support capability X?".
- **App registry** — `AppRegistryInterface` collects all apps and applies per-deployment catalog overrides (scopes, flags, tags, supported resources) from YAML config.
- **Settings UI** — Users and org admins browse, install, configure, and remove apps from a dedicated settings section; no custom controllers needed.

Notifications

Event-driven delivery via email, in-app, and webhooks. See Notifications.

- **Event definitions** — Implement `EventDefinitionProviderInterface` to register notification events; they appear in preferences UI and webhook triggers automatically.
- **Email channel** — Renders Twig templates and delivers via Symfony Mailer.
- **In-app channel** — Persists `InAppNotification` entities; displayed in the UI with read/unread state.
- **Webhook channel** — HTTP POST to user/org-configured endpoints, HMAC-signed (`X-Webhook-Signature`), delivered async via Symfony Messenger. Delivery history is retained and cleaned up via cron.
- **Recipient resolvers** — `NotificationRecipientResolverInterface` decouples recipient logic from dispatch logic; multiple resolvers can run per event.
- **Per-user preferences** — Users control which channels they receive per event type; the dispatcher respects preferences before routing.
- **Async delivery** — Messenger integration keeps webhook delivery out of the main request path.
- **Retention policy** — Configurable `max_age_days` and `max_per_endpoint` limits; `notifications:webhook-deliveries:cleanup` command for periodic pruning.

API

Helpers for API Platform integration.

- **Scope enforcement** — `#[RequiresScope]` on API Platform state processors; a compiler pass builds a static processor → scopes map at compile time.
- **Tabulator.js adapter** — `TabulatorQueryStringSubscriber` translates Tabulator.js query parameters to API Platform filter parameters.
- **Serializer helpers** — Custom serializers for bundle value objects and entities.

CLI commands

Command	What it does
`derafu:platform:seed`	Run registered seeders
`identity:tokens:purge`	Delete expired verification tokens
`identity:login-history:purge --days=N`	Delete login records older than N days
`notifications:webhook-deliveries:cleanup`	Apply webhook delivery retention policy

Last updated on 24/08/2026

Installation

Install the bundle, create entities, configure services, and run migrations.

Anonymous · 24/08/2026

Installation

1. Require the bundle

```
composer require derafu/symfony-platform-bundle
```

If the bundle is consumed as a local path repository during development:

```
{
  "repositories": [
    { "type": "path", "url": "./packages/DerafuPlatformBundle" }
  ],
  "require": {
    "derafu/symfony-platform-bundle": "*"
  }
}
```

2. Register the bundle

If Symfony Flex did not register it automatically:

```
// config/bundles.php
return [
    // ...
    Derafu\PlatformBundle\PlatformBundle::class => ['all' => true],
];
```

3. Create the concrete entities

The bundle provides abstract `Base*` MappedSuperclass classes. The app must create concrete entities that extend them. By convention, entities are grouped by domain under `App\Entity\`.

Identity entities (required)

```
// src/Entity/Auth/User.php
namespace App\Entity\Auth;

use Derafu\PlatformBundle\Identity\Entity\BaseUser;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\Table(name: 'auth_users')]
class User extends BaseUser {}
```

Repeat for every identity entity:

Base class	Suggested concrete	Suggested table
`BaseUser`	`App\Entity\Auth\User`	`auth_users`
`BaseRole`	`App\Entity\Auth\Role`	`auth_roles`
`BasePermission`	`App\Entity\Auth\Permission`	`auth_permissions`
`BaseUserRole`	`App\Entity\Auth\UserRole`	`auth_user_roles`
`BaseRolePermission`	`App\Entity\Auth\RolePermission`	`auth_role_permissions`
`BaseOrganization`	`App\Entity\Auth\Organization`	`auth_organizations`
`BaseOrganizationMembership`	`App\Entity\Auth\OrganizationMembership`	`auth_organization_memberships`
`BaseOrganizationInvitation`	`App\Entity\Auth\OrganizationInvitation`	`auth_organization_invitations`
`BaseVerificationToken`	`App\Entity\Auth\VerificationToken`	`auth_verification_tokens`
`BaseApiKey`	`App\Entity\Auth\ApiKey`	`auth_api_keys`
`BaseJwtToken`	`App\Entity\Auth\JwtToken`	`auth_jwt_tokens`
`BaseLoginRecord`	`App\Entity\Auth\LoginRecord`	`auth_login_records`
`BaseImpersonationRecord`	`App\Entity\Auth\ImpersonationRecord`	`auth_impersonation_records`
`BaseTeam`	`App\Entity\Auth\Team`	`auth_teams`
`BaseTeamMembership`	`App\Entity\Auth\TeamMembership`	`auth_team_memberships`
`BaseLocale`	`App\Entity\Auth\Locale`	`auth_locales`
`BaseTimezone`	`App\Entity\Auth\Timezone`	`auth_timezones`

All base classes live under `Derafu\PlatformBundle\Identity\Entity\`.

Apps entities (required if using the Apps module)

Base class	Suggested concrete	Suggested table
`BaseUserAppInstallation`	`App\Entity\Apps\UserAppInstallation`	`apps_user_installations`
`BaseOrganizationAppInstallation`	`App\Entity\Apps\OrganizationAppInstallation`	`apps_organization_installations`

Base classes: `Derafu\PlatformBundle\Apps\Entity\`.

Notifications entities (required if using the Notifications module)

Base class	Suggested concrete	Suggested table
`BaseInAppNotification`	`App\Entity\Notif\InAppNotification`	`notif_inapp`
`BaseUserWebhookEndpoint`	`App\Entity\Notif\UserWebhookEndpoint`	`notif_user_webhook_endpoints`
`BaseOrganizationWebhookEndpoint`	`App\Entity\Notif\OrganizationWebhookEndpoint`	`notif_org_webhook_endpoints`
`BaseUserWebhookDelivery`	`App\Entity\Notif\UserWebhookDelivery`	`notif_user_webhook_deliveries`
`BaseOrganizationWebhookDelivery`	`App\Entity\Notif\OrganizationWebhookDelivery`	`notif_org_webhook_deliveries`
`BaseUserNotificationPreference`	`App\Entity\Notif\UserNotificationPreference`	`notif_user_preferences`
`BaseUserDigestPreference`	`App\Entity\Notif\UserDigestPreference`	`notif_user_digest_preferences`

Base classes: `Derafu\PlatformBundle\Notifications\Entity\`.

4. Create the concrete repositories

For each entity, create a repository that calls the bundle's base with your concrete class:

```
// src/Repository/Auth/UserRepository.php
namespace App\Repository\Auth;

use App\Entity\Auth\User;
use Derafu\PlatformBundle\Identity\Repository\BaseUserRepository;
use Doctrine\Persistence\ManagerRegistry;

class UserRepository extends BaseUserRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, User::class);
    }
}
```

Base repositories exist for all identity, apps, and notifications entities.

5. Configure the bundle

If you follow the `App\Entity\Auth\*`, `App\Entity\Apps\*`, and `App\Entity\Notif\*` naming conventions, **most configuration is optional** — the defaults match. See Configuration for the full reference.

Minimum required configuration when the Notifications module is active:

```
# config/packages/derafu_platform.yaml
derafu_platform:
    notifications:
```

```
        from_email: '%env(MAILER_FROM_EMAIL)%'
        from_name: '%env(MAILER_FROM_NAME)%'
```

6. Configure Symfony Security

Register the bundle's authenticators in your security firewall:

```
# config/packages/security.yaml
security:
    firewalls:
        main:
            form_login:
                login_path: derafu_platform_identity_login
                check_path: derafu_platform_identity_login
            custom_authenticators:
                - Derafu\PlatformBundle\Identity\Security\ApiKeyAuthenticator
                - Derafu\PlatformBundle\Identity\Security\MagicLinkAuthenticator
```

7. Configure Messenger (for async notifications)

The Notifications module dispatches messages via Symfony Messenger. Route them to a transport for async delivery:

```
# config/packages/messenger.yaml
framework:
    messenger:
        transports:
            async: '%env(MESSENGER_TRANSPORT_DSN)%'
        routing:
            'Derafu\PlatformBundle\Notifications\Messenger\*': async
```

8. Run migrations

```
bin/console doctrine:migrations:diff
bin/console doctrine:migrations:migrate
```

9. Seed initial data

The bundle does not ship seed data. Create the roles your app needs and seed locales/timezones:

```
INSERT INTO auth_roles (code, name) VALUES
  ('system.admin', 'System Administrator'),
```

```
  ('org.owner',    'Organization Owner'),
  ('org.admin',    'Organization Admin'),
  ('org.member',   'Organization Member');
```

For locales and timezones the bundle provides a `SeedCommand` that you can trigger with the `derafu:platform:seed` console command, provided your app registers seeders implementing `SeederInterface`.

Last updated on 24/08/2026

Configuration reference

Full YAML config reference for all bundle modules.

Anonymous · 24/08/2026

Configuration reference

The bundle is configured under the `derafu_platform:` key. All values have sensible defaults — most apps need minimal configuration.

```
# config/packages/derafu_platform.yaml

derafu_platform:

    # ── Identity ──────────────────────────────────────────────────────────
    identity:

        # Entity FQCNs — defaults follow App\Entity\Auth\{Name} convention.
        user_class:                     App\Entity\Auth\User
        organization_class:             App\Entity\Auth\Organization
        role_class:                     App\Entity\Auth\Role
        permission_class:               App\Entity\Auth\Permission
        user_role_class:                App\Entity\Auth\UserRole
        role_permission_class:          App\Entity\Auth\RolePermission
        verification_token_class:       App\Entity\Auth\VerificationToken
        organization_membership_class:  App\Entity\Auth\OrganizationMembership
        organization_invitation_class:  App\Entity\Auth\OrganizationInvitation
        api_key_class:                  App\Entity\Auth\ApiKey
        jwt_token_class:                App\Entity\Auth\JwtToken
        login_record_class:             App\Entity\Auth\LoginRecord
        impersonation_record_class:     App\Entity\Auth\ImpersonationRecord
        team_class:                     App\Entity\Auth\Team
        team_membership_class:          App\Entity\Auth\TeamMembership
        locale_class:                   App\Entity\Auth\Locale
        timezone_class:                 App\Entity\Auth\Timezone

        # Supported locales shown in the language selector.
        supported_locales:
            en: English
            es: Español

        # How long an organization invitation stays valid (seconds).
        invitation_ttl: 604800          # 7 days

        # Seconds after login before sudo mode expires.
```

```
        sudo_ttl: 900                   # 15 minutes

        # Account lockout after N consecutive failed logins.
        lockout_max_attempts: 5
        lockout_duration: 1800          # 30 minutes

        # Per-type TTL for one-use verification tokens (seconds, min 60).
        verification_ttl:
            email_verify:  86400        # 1 day
            password_reset: 3600        # 1 hour
            email_change:  86400        # 1 day
            magic_link:    900          # 15 minutes

        # Fixed-window rate limiting per API key. Disabled by default.
        rate_limit:
            enabled: false
            default_limit: 1000         # max requests per window
            default_window: 3600        # window size in seconds
            scope_limits:               # per-scope overrides (optional)
                'write:invoices':
                    limit: 100
                    window: 3600

    # ── Identity UI ──────────────────────────────────────────────────────
    identity_ui:

        # Your app's base Twig layout. The bundle's templates extend this.
        parent_layout: 'layouts/app.html.twig'

        # Sender address for transactional emails (verification, reset, etc.)
        from_email: noreply@example.com
        from_name:  App

        # Force users to verify their email before accessing the app.
        require_email_verification: true

    # ── Apps ─────────────────────────────────────────────────────────────
    apps:

        # Entity FQCNs — defaults follow App\Entity\Apps\{Name} convention.
        user_installation_class:         App\Entity\Apps\UserAppInstallation
        organization_installation_class: App\Entity\Apps\OrganizationAppInstallation

        # Catalog overrides: configure app scopes, flags, tags, and
        # which resource classes each app supports. Apps registered as
        # Symfony services (AppDefinitionInterface) are discovered
        # automatically; the catalog section lets you override metadata.
        catalog:
            stripe:
                scopes: [user, organization]
                flags: [featured]
```

```
            telegram:
                scopes: [user]
                flags: [featured]
            smtp:
                scopes: [resource]
                resources:
                    - App\Entity\Resource\Project

    # ── Notifications ──────────────────────────────────────────────────────
    notifications:

        # Sender address for notification emails.
        from_email: '%env(MAILER_FROM_EMAIL)%'
        from_name:  '%env(MAILER_FROM_NAME)%'

        # Entity FQCNs — defaults follow App\Entity\Notif\{Name} convention.
        inapp_notification_class:              App\Entity\Notif\InAppNotification
        user_webhook_endpoint_class:           App\Entity\Notif\UserWebhookEndpoint
        organization_webhook_endpoint_class:
App\Entity\Notif\OrganizationWebhookEndpoint
        user_webhook_delivery_class:           App\Entity\Notif\UserWebhookDelivery
        organization_webhook_delivery_class:
App\Entity\Notif\OrganizationWebhookDelivery
        user_notification_preference_class:
App\Entity\Notif\UserNotificationPreference
        user_digest_preference_class:          App\Entity\Notif\UserDigestPreference

        # Webhook delivery log retention policy.
        webhook_delivery_retention:
            max_age_days:    30   # delete deliveries older than N days
            max_per_endpoint: 500 # keep at most N deliveries per endpoint

    # ── API ────────────────────────────────────────────────────────────────
    api: ~
```

Container parameters

Every config value is exposed as a container parameter for injection into custom services:

Parameter	Example value
`derafu_platform.identity.user_class`	`App\Entity\Auth\User`
`derafu_platform.identity.organization_class`	`App\Entity\Auth\Organization`
`derafu_platform.identity.invitation_ttl`	`604800`
`derafu_platform.identity.sudo_ttl`	`900`
`derafu_platform.identity.lockout_max_attempts`	`5`
`derafu_platform.identity.lockout_duration`	`1800`
`derafu_platform.identity.verification_ttl.email_verify`	`86400`

Parameter	Example value
`derafu_platform.identity.verification_ttl.password_reset`	`3600`
`derafu_platform.identity.rate_limit.enabled`	`false`
`derafu_platform.identity.rate_limit.default_limit`	`1000`
`derafu_platform.identity.supported_locales`	`['en' => 'English', ...]`
`derafu_platform.identity_ui.parent_layout`	`layouts/app.html.twig`
`derafu_platform.identity_ui.from_email`	`noreply@example.com`
`derafu_platform.identity_ui.require_email_verification`	`true`
`derafu_platform.apps.user_installation_class`	`App\Entity\Apps\UserAppInstallation`
`derafu_platform.notifications.from_email`	`noreply@example.com`
`derafu_platform.notifications.webhook_delivery_retention.max_age_days`	`30`

Service aliases

The bundle aliases every service interface to its default implementation. Type-hint the interface in your constructors:

```
use Derafu\PlatformBundle\Identity\Contract\Service\RegistrarInterface;
use Derafu\PlatformBundle\Identity\Contract\Service\PermissionCheckerInterface;
use Derafu\PlatformBundle\Identity\Contract\Service\OrganizationManagerInterface;
use Derafu\PlatformBundle\Identity\Contract\Service\InvitationManagerInterface;

public function __construct(
    private readonly RegistrarInterface $registrar,
    private readonly PermissionCheckerInterface $checker,
    private readonly OrganizationManagerInterface $orgManager,
    private readonly InvitationManagerInterface $invitations,
) {}
```

To swap an implementation, bind your service to the interface in `services.yaml`:

```
services:
    Derafu\PlatformBundle\Identity\Contract\Service\PermissionCheckerInterface:
        alias: App\Service\CachingPermissionChecker
```

Last updated on 24/08/2026

Entities

MappedSuperclass pattern, entity map, fields, and extending.

Anonymous · 24/08/2026

Entity model

Pattern: MappedSuperclass + ResolveTargetEntityListener

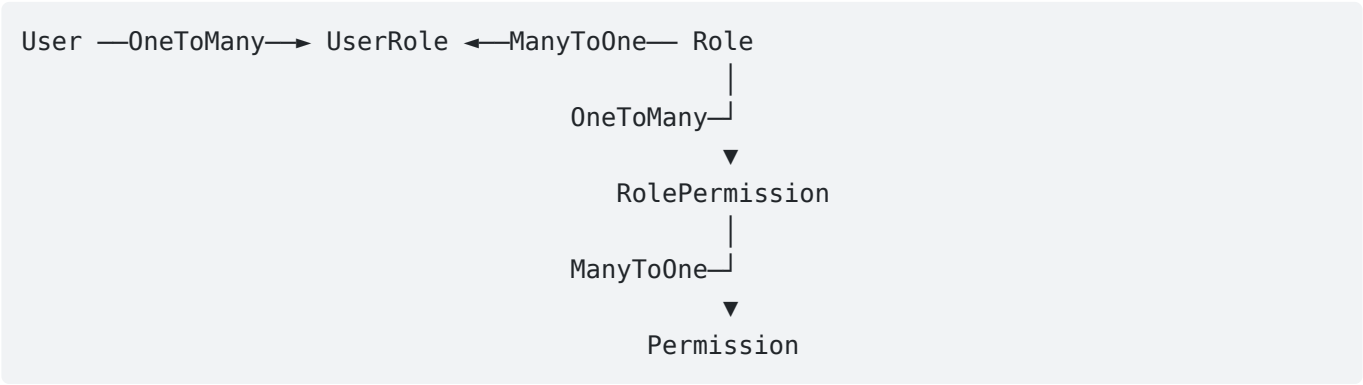
Every entity is split in two:

- **`Base*`** (bundle) — abstract `#[ORM\MappedSuperclass]` with all fields, relations, and logic. Lives in `Derafu\PlatformBundle\{Module}\Entity\`.
- **Concrete** (app) — extends the Base, adds `#[ORM\Entity]` + `#[ORM\Table]`. Lives in `App\Entity\{Group}\`.

ORM relations in the bundle target **interfaces** (not concrete classes). Doctrine resolves them at runtime via `resolve_target_entities`, wired automatically by the bundle's DI extension.

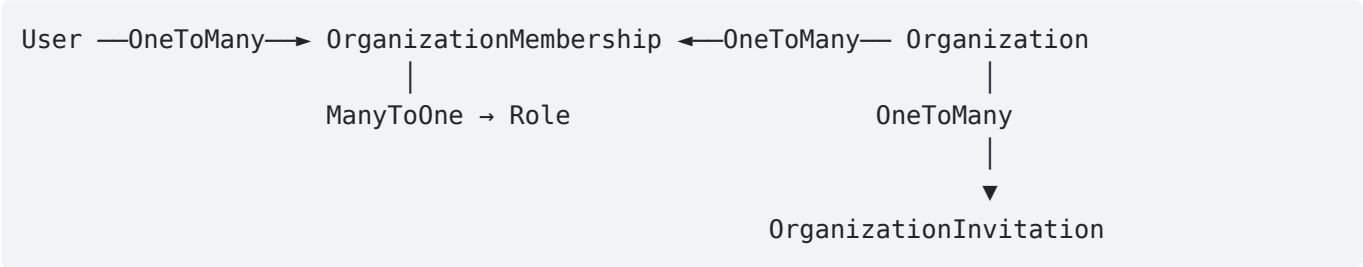
Identity entities

Core identity



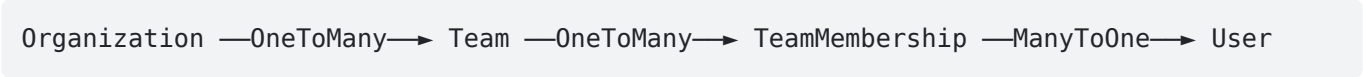
- **User** — the authenticated subject. Fields: `id`, `uuid` (v7), `name`, `email` (unique, login identifier), `password` (hashed), `active`, `emailVerifiedAt`, `createdAt`, `config` (JSON). Implements Symfony's `UserInterface` and `PasswordAuthenticatedUserInterface`.
- **Role** — reusable role identified by a unique `code` (e.g. `org.admin`). No scope column — the scope is defined by the relation that uses the role.
- **Permission** — atomic capability identified by a unique `code` (e.g. `invoice.create`).
- **UserRole** — global system-level role assignment (super-admin, auditor).
- **RolePermission** — which permissions a role grants.

Organizations



- **Organization** — a tenant. Fields: `id`, `uuid`, `name`, `active`, `createdAt`, `config`. No `owner` column — the owner is the membership with `role.code = 'org.owner'`.
- **OrganizationMembership** — user ↔ organization ↔ role with unique constraint `(user, organization)`. Field `joinedAt`.
- **OrganizationInvitation** — pending invite. Fields: `email`, `tokenHash` (SHA-256), `expiresAt`, `invitedBy`, `acceptedAt`, `acceptedBy`, `revokedAt`, `createdAt`.

Teams



- **Team** — a named group of org members with a URL-friendly `slug`.
- **TeamMembership** — user ↔ team pivot, validates org membership.

Verification tokens

- **VerificationToken** — one-use token for user lifecycle operations. Fields: `tokenHash` (SHA-256), `user`, `type` (enum), `payload` (JSON), `expiresAt`, `usedAt`, `createdAt`. Types: `email_verify`, `password_reset`, `email_change`, `magic_link`.

API and JWT tokens

- **ApiKey** — long-lived programmatic access token. Format: `dik_` + 40 hex chars. Fields: `name`, `tokenHash`, `displayPrefix`, `scopes` (JSON), `lastUsedAt`, `expiresAt`, `createdAt`.
- **JwtToken** — persisted JWT for audit and revocation. Fields: `jti` (JWT ID), `user`, `issuedAt`, `expiresAt`, `revokedAt`.

Security records

- **LoginRecord** — one row per successful login: IP, User-Agent, authenticator, `loginAt`.
- **ImpersonationRecord** — one row per `switch_user` session: admin, target, IP, `startedAt`, `endedAt`.

Localization

- **Locale** — supported locale code and display name.
- **Timezone** — supported timezone identifier and display name.

Apps entities

```
User / Organization ──OneToMany──► UserAppInstallation / OrganizationAppInstallation
                                          └── config (JSON, encrypted sensitive keys)
```

- **BaseUserAppInstallation** — app installed for a specific user. Fields: `user`, `appCode`, `config` (JSON), `installedAt`, `active`.
- **BaseOrganizationAppInstallation** — app installed for an organization. Fields: `organization`, `appCode`, `config` (JSON), `installedAt`, `active`.
- **BaseResourceAppInstallation** — app installed for a specific app resource. Extend per resource type with a typed FK to the resource.

All live under `Derafu\PlatformBundle\Apps\Entity\`.

Notifications entities

```
User ──OneToMany──► UserWebhookEndpoint ──OneToMany──► UserWebhookDelivery
User ──OneToMany──► UserNotificationPreference
User ──OneToMany──► UserDigestPreference
User ──OneToMany──► InAppNotification

Organization ──OneToMany──► OrganizationWebhookEndpoint
                                └──OneToMany──► OrganizationWebhookDelivery
```

- **InAppNotification** — in-app notification persisted for the user. Fields: `user`, `eventCode`, `data` (JSON), `readAt`, `createdAt`.
- **UserWebhookEndpoint** / **OrganizationWebhookEndpoint** — registered webhook URL. Fields: `url`, `secret` (for HMAC signing), `active`, `eventCodes` (JSON array of subscribed events), `createdAt`.
- **UserWebhookDelivery** / **OrganizationWebhookDelivery** — delivery attempt log. Fields: `endpoint`, `eventCode`, `payload` (JSON), `status`, `statusCode`, `responseBody`, `attemptedAt`.
- **UserNotificationPreference** — per-user, per-event channel preferences.
- **UserDigestPreference** — digest schedule preferences.

All live under `Derafu\PlatformBundle\Notifications\Entity\`.

Extending entities

The concrete entity can add fields without breaking bundle services:

```
#[ORM\Entity]
#[ORM\Table(name: 'auth_users')]
class User extends BaseUser
{
```

```
    #[ORM\Column(type: Types::STRING, length: 20, nullable: true)]
    private ?string $phone = null;

    public function getPhone(): ?string
    {
        return $this->phone;
    }

    public function setPhone(?string $phone): static
    {
        $this->phone = $phone;
        return $this;
    }
}
```

Bundle services, voters, and events only see the entity via its interface (`UserInterface`) — extra fields are transparent to the bundle.

Constructors

- `BaseUser` and `BaseOrganization` auto-generate UUID v7 and freeze `createdAt` at construction time.
- `BaseUserRole` and `BaseRolePermission` require both FK sides as constructor arguments.
- `BaseOrganizationMembership` requires `(user, organization, role)`.
- `BaseOrganizationInvitation` requires `(organization, email, role, tokenHash, expiresAt)`.
- `BaseVerificationToken` requires `(user, type, expiresAt, tokenHash)`.

Last updated on 24/08/2026

User lifecycle

Registration, email verification, password reset, email change, and more.

Anonymous · 24/08/2026

User lifecycle

The bundle provides services for all common user lifecycle flows. Each service emits events so the app can send emails, log activity, or trigger side effects — the bundle stays transport-agnostic.

Token security model

All tokens (email verification, password reset, email change, magic link) share the same security model:

- **256 bits of entropy**: `bin2hex(random_bytes(32))` → 64-char hex string.
- **SHA-256 hash stored**: the database holds `hash('sha256', $plaintext)`, never the plaintext. A leaked database cannot be replayed.
- **Single-use**: consumption sets `usedAt`.
- **Time-boxed**: each token type has a configurable TTL.
- **`hash_equals()`**: timing-safe comparison on validation.

The plaintext is returned exactly once in a `GeneratedToken` wrapper and is meant to be emailed immediately.

Registration

```
use Derafu\PlatformBundle\Identity\Contract\Service\RegistrarInterface;

class RegisterController
{
    public function __invoke(RegistrarInterface $registrar, Request $request):
Response
    {
        $registration = $registrar->register(
            email: 'alice@example.com',
            plainPassword: 'secret',
            name: 'Alice'
        );

        // $registration->user                              — the persisted User
        // $registration->verificationToken->plaintext     — for the email
```

```
        return new JsonResponse(['message' => 'Check your inbox.'], 201);
    }
}
```

The `UserRegistered` event is dispatched with the plaintext token. Subscribe to it to send the confirmation email:

```
#[AsEventListener]
class SendWelcomeEmail
{
    public function __invoke(UserRegistered $event): void
    {
        // Build URL: /verify-email?token={$event->verificationTokenPlaintext}
        // Send email to $event->user->getEmail()
    }
}
```

Email verification

```
use Derafu\PlatformBundle\Identity\Contract\Service\EmailVerifierInterface;

// GET /verify-email?token=abc123...
$user = $emailVerifier->verify($request->query->get('token'));
// $user->isEmailVerified() is now true
```

Emits `UserEmailVerified`.

Password reset

Request phase (user submits email):

```
use Derafu\PlatformBundle\Identity\Contract\Service\PasswordResetterInterface;

$passwordResetter->requestReset('alice@example.com');
// Silent no-op if email not registered (anti-enumeration).
```

Emits `PasswordResetRequested` (only when user exists). Subscribe to send the reset email with the plaintext token.

Reset phase (user clicks link, submits new password):

```
$user = $passwordResetter->reset($token, $newPassword);
```

Emits `UserPasswordChanged`.

Email change

```
use Derafu\PlatformBundle\Identity\Contract\Service\EmailChangerInterface;

// Step 1: request (sends confirmation to the NEW email)
$generated = $emailChanger->requestChange($currentUser, 'newemail@example.com');

// Step 2: confirm (user clicks link in the new email)
$user = $emailChanger->confirmChange($token);
// $user->getEmail() is now 'newemail@example.com'
```

The new email is bound to the token's payload — it cannot be swapped after issuance. Emits `EmailChangeRequested` (step 1) and `UserEmailChanged` (step 2, carries both old and new addresses).

Magic links (passwordless login)

Magic links let users log in by clicking a link sent to their email — no password needed.

Request a link:

```
use Derafu\PlatformBundle\Identity\Contract\Service\MagicLinkManagerInterface;

// POST /magic-link (user submits their email)
$magicLinkManager->requestLink('alice@example.com');
// Silent no-op if email not registered (anti-enumeration).
```

Emits `MagicLinkRequested` (only when user exists). Subscribe to send the login email:

```
#[AsEventListener]
class SendMagicLinkEmail
{
    public function __invoke(MagicLinkRequested $event): void
    {
        $url = 'https://app.example.com/magic-link?_magic_link_token='
             . $event->plaintextToken;
        // Send email to $event->user->getEmail() with $url
    }
}
```

Authenticate via the link:

The bundle ships a Symfony Security authenticator. Enable it in `security.yaml`:

```
security:
    firewalls:
```

```
        main:
            custom_authenticators:
                - Derafu\PlatformBundle\Identity\Security\MagicLinkAuthenticator
```

The authenticator fires on any request with a `_magic_link_token` query parameter, consumes the token, and authenticates the user.

Token cleanup

Expired tokens pile up over time. Run the purge command periodically:

```
bin/console identity:tokens:purge
```

Typically as a daily cron job.

Sudo mode (re-authentication for sensitive actions)

Sudo mode is a time-limited elevated state that proves the user recently confirmed their password. Actions that require sudo will throw `SudoRequiredException` when the window expires.

Confirm password:

```
use Derafu\PlatformBundle\Identity\Contract\Service\SudoManagerInterface;

$sudoManager->confirmPassword($this->getUser(), $request->get('password'));
// Sudo window refreshed.
```

Check sudo state:

```
$sudoManager->isGranted($user); // true within the TTL window
```

Auto-granted on login. The `SudoOnLoginListener` calls `grant()` after every successful login — users are not asked to re-confirm immediately after typing their password.

Session-based, no DB. The timestamp lives in the session. Works naturally in PHP-FPM.

Account lockout

After N consecutive failed login attempts (default 5), the account is automatically locked for a configurable duration (default 30 min).

Fully automatic. The `AccountLockListener` hooks into Symfony's authentication events — no app-

side wiring needed.

Admin unlock:

```
use Derafu\PlatformBundle\Identity\Contract\Service\AccountLockManagerInterface;

$lockManager->unlock($user);
// Resets counter + clears lock. Emits AccountUnlocked.
```

Check programmatically:

```
$lockManager->isLocked($user); // true when lockedUntil > now
```

Login history

Every successful login is automatically recorded — IP, User-Agent, and which authenticator succeeded. Zero wiring needed.

Query recent logins:

```
use Derafu\PlatformBundle\Identity\Contract\Service\LoginHistoryManagerInterface;

$records = $loginHistoryManager->getRecentForUser($user, limit: 10);

foreach ($records as $record) {
    echo $record->getIpAddress();
    echo $record->getUserAgent();
    echo $record->getAuthenticator();
    echo $record->getCreatedAt()->format('Y-m-d H:i');
}
```

Purge old records:

```
bin/console identity:login-history:purge --days=90
```

TTL & lockout configuration

```
derafu_platform:
    identity:
        sudo_ttl: 900               # 15 minutes (default)
        lockout_max_attempts: 5     # lock after 5 failures (default)
        lockout_duration: 1800      # lock for 30 minutes (default)
        verification_ttl:
            email_verify: 86400     # 1 day (default)
```

```
            password_reset: 3600    # 1 hour (default)
            email_change: 86400     # 1 day (default)
            magic_link: 900         # 15 minutes (default)
```

Last updated on 24/08/2026

Organizations & invitations

Multi-tenant orgs, memberships, teams, invitations, and context.

Anonymous · 24/08/2026

Organizations & invitations

Creating an organization

```
use Derafu\PlatformBundle\Identity\Contract\Service\OrganizationManagerInterface;

class CreateOrgController
{
    public function __invoke(
        OrganizationManagerInterface $orgManager,
        RoleRepository $roleRepo
    ): Response {
        $ownerRole = $roleRepo->findOneBy(['code' => 'org.owner']);

        $org = $orgManager->create(
            owner: $this->getUser(),
            name: 'Acme Inc',
            ownerRole: $ownerRole
        );

        // $org is persisted with a membership (role = org.owner)
        return new JsonResponse(['id' => $org->getId()]);
    }
}
```

`create()` is transactional: the organization and the owner-membership are persisted atomically. Emits `OrganizationCreated`.

Adding members directly

```
$membership = $orgManager->addMember($org, $user, $memberRole);
```

Throws `OrganizationOperationException` if the user is already a member. Emits `OrganizationMemberAdded`.

Removing members

```
$orgManager->removeMember($membership);
```

Throws `OrganizationOperationException` if the membership is the owner (`role.code = 'org.owner'`). Transfer ownership first. Emits `OrganizationMemberRemoved`.

Transferring ownership

```
$adminRole = $roleRepo->findOneBy(['code' => 'org.admin']);

$orgManager->transferOwnership($org, $newOwner, demoteCurrentOwnerTo: $adminRole);
```

The new owner must already be a member. Both role changes happen in a single transaction. Emits `OrganizationOwnershipTransferred` with the previous and new owners.

Finding the owner

```
$owner = $orgManager->getOwner($org); // ?UserInterface
```

Scans memberships for `role.code = 'org.owner'`. Returns `null` if none found (should not happen if the invariant is maintained).

Teams

Teams are named groups of organization members. They simplify resource access grants: instead of adding 20 users individually to a resource, assign a team with a role.

```
use Derafu\PlatformBundle\Identity\Contract\Service\TeamManagerInterface;

// Create a team
$team = $teamManager->create($org, 'backend', 'Backend Team');

// Add a member (must already be an org member)
$teamManager->addMember($team, $user);

// Remove a member
$teamManager->removeMember($team, $user);
```

For team-level resource access, the resource entity must implement `TeamAccessibleInterface`. The `PermissionChecker` cascade automatically checks team access between direct resource access and org-level access.

Invitations

Invite by email

```
use Derafu\PlatformBundle\Identity\Contract\Service\InvitationManagerInterface;

$generated = $invitationManager->invite(
    organization: $org,
    email: 'bob@example.com',
    role: $memberRole,
    invitedBy: $currentUser   // optional, for audit trail
);

// $generated->plaintext — build accept URL and email it
// $generated->invitation — the persisted entity
```

Emits `InvitationCreated` with the plaintext token. Subscribe to send the invitation email.

Accept an invitation

```
// GET /invitations/accept?token=abc123...
$membership = $invitationManager->accept($token, $authenticatedUser);
```

Validates:

1. Token is known and active (not expired, not revoked, not accepted).
2. `$authenticatedUser->getEmail()` matches `$invitation->getEmail()` (case-insensitive). This prevents session hijacking — someone logged in with a different email cannot consume another person's invitation.

On success: marks the invitation as accepted, creates the membership via `OrganizationManager::addMember()`, emits `InvitationAccepted`.

Revoke an invitation

```
$invitationManager->revoke($invitation);
```

Only active invitations can be revoked. Emits `InvitationRevoked`.

Purge expired invitations

```
$invitationManager->purgeExpired();
```

Deletes pending-then-expired invitations. Accepted and revoked invitations are kept as audit trail.

Organization context

For request-scoped “which org am I operating in?” resolution:

```
use Derafu\PlatformBundle\Identity\Contract\Service\OrganizationContextInterface;

// In a kernel.request listener or middleware:
$orgId = $request->headers->get('X-Organization-Id');
$org = $orgRepository->find($orgId);
$organizationContext->setCurrent($org);

// Anywhere downstream:
$org = $organizationContext->getCurrent();
```

The default implementation (`InMemoryOrganizationContext`) is a stateful singleton. In PHP-FPM this works naturally (container rebuilt per request). For long-running workers (Swoole, RoadRunner), reset the context between requests.

TTL configuration

```
derafu_platform:
    identity:
        invitation_ttl: 604800  # 7 days (default)
```

Last updated on 24/08/2026

Authorization (RBAC)

Cascading RBAC: global, organization, team, and resource voters.

Anonymous · 24/08/2026

Authorization (RBAC)

The bundle provides a **four-level cascading permission system**:

```
Resource-level → Team-level → Organization-level → Global system-level
```

Each level falls back to the next. A global super-admin passes every check without being explicitly named in each organization or resource.

Roles and permissions

Roles are identified by a unique `code` string. There is no `scope` column — the context is defined by the **relation** that uses the role:

Relation	Context	Example codes
`UserRole`	Global system	`system.admin`, `system.auditor`
`OrganizationMembership`	Organization	`org.owner`, `org.admin`, `org.member`
`TeamAccess` (app)	Team within org	`team.lead`, `team.contributor`
`ResourceAccess` (app)	Resource	`project.admin`, `project.read`

Permissions are atomic capabilities attached to roles via the `RolePermission` pivot:

```
Role "org.admin" ──► Permission "org.invite"
                 ──► Permission "org.billing"
                 ──► Permission "org.settings"
```

Using voters in controllers

The bundle registers three voters automatically:

Global (no subject)

```
#[IsGranted('system.manage')]
```

```
public function adminDashboard(): Response { ... }
```

Checks: `user → UserRole → Role → RolePermission → Permission.code`.

Organization-scoped (subject = Organization)

```
#[IsGranted('org.invite', subject: 'organization')]
public function inviteMember(Organization $organization): Response { ... }
```

Checks: `user → OrganizationMembership(org) → Role → Permissions`. Falls back to global.

Resource-scoped (subject = ResourceAccessible entity)

```
#[IsGranted('write', subject: 'project')]
public function editProject(Project $project): Response { ... }
```

Checks: `user → ResourceAccess(resource) → Role → Permissions`. Falls back to team-level (if resource implements `TeamAccessibleInterface`), then org-level (if resource implements `OrganizationOwnedInterface`), then global.

Programmatic checks

```
use Derafu\PlatformBundle\Identity\Contract\Service\PermissionCheckerInterface;

class SomeService
{
    public function __construct(
        private readonly PermissionCheckerInterface $checker
    ) {}

    public function doSomething(UserInterface $user, Organization $org): void
    {
        // Global
        if ($this->checker->hasPermission($user, 'system.manage')) { ... }

        // Organization-scoped
        if ($this->checker->hasOrganizationPermission($user, $org, 'org.invite')) {
... }

        // Resource-scoped
        if ($this->checker->hasResourcePermission($user, $project, 'write')) { ... }
    }
}
```

How the cascade works

`hasResourcePermission($user, $resource, 'write')`:

1. If `$resource` implements `ResourceAccessibleInterface`: iterate the resource's direct access entries for this user → check each role's permissions for `'write'`.
2. If not found and `$resource` implements `TeamAccessibleInterface`: iterate team access entries → for each team, check if the user is a member → if yes, check the team's role permissions for `'write'`.
3. If not found and `$resource` implements `OrganizationOwnedInterface`: call `hasOrganizationPermission($user, $resource->getOrganization(), 'write')`.
4. If not found: call `hasPermission($user, 'write')` (global).

Any level returning `true` short-circuits the rest.

API Platform integration

The bundle's voters work with API Platform's `security` attribute out of the box — no extra configuration. API Platform calls Symfony's `AuthorizationCheckerInterface`, which triggers the voters:

```
use ApiPlatform\Metadata\ApiResource;
use ApiPlatform\Metadata\Get;
use ApiPlatform\Metadata\GetCollection;
use ApiPlatform\Metadata\Post;

#[ApiResource(
    operations: [
        new GetCollection(security: "is_granted('ROLE_USER')"),
        new Get(security: "is_granted('read', object)"),
        new Post(security: "is_granted('write', object)"),
    ],
)]
class Project { ... }
```

The `Get` and `Post` operations pass `object` as the subject, so the `ResourcePermissionVoter` fires and resolves the full cascade (direct → teams → org → global). The `GetCollection` operation uses a global check (no subject), so the `PermissionVoter` fires.

For automatic **listing filters** (which rows the user sees), see Multi-tenancy & RLS.

Vote reasons (debugging)

All three voters add human-readable reasons to Symfony's `Vote` object when they deny. Check the

Web Profiler’s Security panel to see why a specific check failed:

```
User "alice@example.com" does not hold permission "org.invite"
in organization #42.
```

Last updated on 24/08/2026

Multi-tenancy & RLS

Row-level security with automatic Doctrine filtering by tenant.

Anonymous · 24/08/2026

Multi-tenancy & row-level security

The bundle provides automatic row-level filtering for entities that belong to a tenant — an Organization, a User, or (with a small amount of app code) any custom resource.

How it works

Three pieces work together:

1. **Tenant contexts** — services that hold "who/what is the current tenant for this request". The bundle provides `OrganizationContext` and `UserContext`; the app can create its own.
2. **Integration interfaces** — `OrganizationOwnedInterface` and `UserOwnedInterface` mark which entities get filtered.
3. **`IdentityTenantFilter`** — a Doctrine SQLFilter that appends `WHERE organization_id = :id` and/or `WHERE owner_id = :id` to every query on entities that implement those interfaces.

A `TenantFilterListener` enables the filter automatically on each request when at least one context has a tenant set.

Built-in: Organization as tenant

```
// App middleware (kernel.request listener, priority >= 0):
$orgId = $request->headers->get('X-Organization-Id');
$org = $orgRepository->find($orgId);
$this->organizationContext->setCurrent($org);

// All queries on OrganizationOwnedInterface entities now auto-filter:
$contribuyentes = $contribuyenteRepository->findAll();
// SQL: SELECT ... FROM contribuyentes WHERE organization_id = 42
```

No changes to repositories, no manual WHERE clauses.

Built-in: User as tenant

```
// App middleware:
```

```
$this->userContext->setCurrent($this->getUser());

// All queries on UserOwnedInterface entities now auto-filter:
$notes = $noteRepository->findAll();
// SQL: SELECT ... FROM personal_notes WHERE owner_id = 7
```

Both at once

An entity can implement both interfaces:

```
class Contribuyente implements OrganizationOwnedInterface, UserOwnedInterface
{
    use OrganizationOwnedTrait;
    use UserOwnedTrait;
}
```

When both contexts are active, both conditions apply (AND):

```
WHERE organization_id = 42 AND owner_id = 7
```

Timing

Set the context in a `kernel.request` listener at priority **0 or higher**. The `TenantFilterListener` runs at priority **-10** — after your context is set but before most controller logic.

Setting the context in a controller is too late: queries during authentication or in other listeners would not be filtered.

Disabling the filter

The filter only activates when at least one context has a value. For admin panels, public endpoints, or CLI commands where you want unfiltered access, simply don't set any context.

To explicitly disable mid-request:

```
$this->entityManager->getFilters()->disable('identity_tenant');
```

Custom tenant: resource-level (e.g. Contribuyente)

The bundle covers Organization and User as tenants. For resource-level tenancy (e.g. all queries scoped to a specific Contribuyente), follow the same pattern:

1. Create a context interface + implementation

```
// src/Service/ContribuyenteContextInterface.php
interface ContribuyenteContextInterface extends TenantContextInterface
{
    public function getCurrent(): ?Contribuyente;
    public function setCurrent(?Contribuyente $contribuyente): void;
}

// src/Service/InMemoryContribuyenteContext.php
final class InMemoryContribuyenteContext implements ContribuyenteContextInterface
{
    private ?Contribuyente $current = null;

    public function getCurrent(): ?Contribuyente { return $this->current; }
    public function setCurrent(?Contribuyente $c): void { $this->current = $c; }
    public function getTenantId(): ?int { return $this->current?->getId(); }
    public function getTenantEntity(): ?object { return $this->current; }
}
```

2. Create a marker interface for filterable entities

```
interface ContribuyenteOwnedInterface
{
    public function getContribuyente(): ?Contribuyente;
}
```

3. Create a Doctrine SQLFilter

```
use Doctrine\ORM\Query\Filter\SQLFilter;

class ContribuyenteTenantFilter extends SQLFilter
{
    public function addFilterConstraint(ClassMetadata $entity, string $alias): string
    {
        if
($entity->getReflectionClass()?->implementsInterface(ContribuyenteOwnedInterface::clas
s)) {
            try {
                return sprintf('%s.contribuyente_id = %s', $alias,
$this->getParameter('contribuyente_id'));
            } catch (\InvalidArgumentException) {}
        }
        return '';
    }
}
```

4. Register and activate

```
# config/packages/doctrine.yaml
doctrine:
    orm:
        filters:
            contribuyente_tenant:
                class: App\Doctrine\Filter\ContribuyenteTenantFilter
                enabled: false
```

```
// In your middleware (kernel.request listener):
$contribuyente = $contribuyenteRepository->find($request->get('contribuyente_id'));
$this->contribuyenteContext->setCurrent($contribuyente);

$filter = $this->entityManager->getFilters()->enable('contribuyente_tenant');
$filter->setParameter('contribuyente_id', (string) $contribuyente->getId(),
'integer');
```

That's it — the exact same pattern the bundle uses internally.

API Platform

The `IdentityTenantFilter` works transparently with API Platform. Since API Platform uses Doctrine under the hood, the filter applies to all listing operations automatically:

```
GET /api/contribuyentes
→ SQL: SELECT ... FROM contribuyentes WHERE organization_id = 42
```

No custom state provider needed. Just set the tenant context in your middleware and API Platform listings are filtered.

For **per-object authorization** (can this user access THIS specific resource?), use API Platform's `security` attribute with the bundle's voters — see Authorization → API Platform.

Summary

Tenant type	Provided by	Interface to implement	Context service
Organization	Bundle	`OrganizationOwnedInterface`	`OrganizationContextInterface`
User	Bundle	`UserOwnedInterface`	`UserContextInterface`
Custom resource	App	App-defined interface	App-defined context

Last updated on 24/08/2026

API keys

Long-lived tokens for programmatic access with optional scopes.

Anonymous · 24/08/2026

API keys

Long-lived tokens for scripts, CI pipelines, and third-party integrations. Distinct from JWT (short-lived session) and from verification tokens (one-use).

Creating a key

```
use Derafu\PlatformBundle\Identity\Contract\Service\ApiKeyManagerInterface;

$generated = $apiKeyManager->create(
    user: $currentUser,
    name: 'CI Pipeline',
    scopes: ['read:projects', 'write:reports'],
    expiresAt: new \DateTimeImmutable('+1 year'),
);

// $generated->plaintext — the full token (dik_...). Show once, never again.
// $generated->apiKey    — the persisted entity with displayPrefix.
```

Emits no event — the UI is responsible for displaying the plaintext token immediately after creation.

Token format

```
dik_<40 hex chars>
```

- `dik_` — “Derafu Identity Key” prefix, never appears in the database.
- 40 hex chars = 160 bits of entropy from `random_bytes(20)`.
- **SHA-256 hash** is stored in the database (`tokenHash`).
- Only the first 8 chars after `dik_` are stored as `displayPrefix` for UI (“ends in …ab3f”) — the full plaintext is irrecoverable after creation.

Authentication

The `ApiKeyAuthenticator` fires on requests with an `Authorization` header matching `Bearer dik_...`. No session is created — every API request re-validates the token.

```
Authorization: Bearer dik_3a7c9b2f...
```

Session users (browser logins) are never affected by API key logic.

Scopes

Scopes are app-defined strings stored in a JSON column on the `ApiKey` entity. An empty scopes array means “unrestricted.”

Defining allowed scopes

Implement `PermissionProviderInterface` to advertise your app’s scopes (they appear in the UI when creating keys):

```
use Derafu\PlatformBundle\Identity\Contract\Service\PermissionProviderInterface;

final class MyAppScopes implements PermissionProviderInterface
{
    public function getPermissions(): array
    {
        return [
            'read:projects'  => 'Read project data',
            'write:projects' => 'Create and update projects',
            'admin:billing'  => 'Manage billing settings',
        ];
    }
}
```

The service is auto-discovered via `PermissionProviderInterface`’s `AutoconfigureTag`.

Enforcing scopes on endpoints

Use the `#[RequiresScope]` attribute on API Platform state processors:

```
use Derafu\PlatformBundle\Identity\Attribute\RequiresScope;

#[RequiresScope('write:projects')]
final class CreateProjectProcessor implements ProcessorInterface
{
    // ...
}
```

A compiler pass (`ProcessorScopePass`) builds a static class → scopes map at compile time. The `ApiKeyAuthenticator` rejects requests where the key lacks the required scope with `403 Forbidden`.

Session users bypass scope checks entirely — scopes restrict API tokens, not interactive users.

Rate limiting

Per-key fixed-window rate limiting. Disabled by default; enable via config:

```
derafu_platform:
    identity:
        rate_limit:
            enabled: true
            default_limit: 1000
            default_window: 3600   # 1 hour
            scope_limits:
                'write:projects':
                    limit: 100
                    window: 3600
```

When enabled, every API-key-authenticated response includes:

```
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 993
X-RateLimit-Reset: 1718000000
```

When the quota is exceeded, the response is `429 Too Many Requests` with a JSON error body. Session users are never rate-limited.

The default implementation uses PSR-6 cache (works with any adapter, including Redis or Memcached). For high-concurrency environments, a `RedisApiKeyRateLimiter` using atomic `INCR` is also provided. Swap it by binding `ApiKeyRateLimiterInterface` to the Redis implementation:

```
services:
    Derafu\PlatformBundle\Identity\Contract\Service\ApiKeyRateLimiterInterface:
        alias: Derafu\PlatformBundle\Identity\Service\RedisApiKeyRateLimiter
```

Listing and revoking keys

```
// All keys for a user
$keys = $apiKeyRepository->findByUser($user);

// Revoke (hard delete)
$apiKeyManager->revoke($apiKey);
```

The bundle’s settings UI (IdentityUi module) provides pages for listing, creating, and revoking API keys.

Last updated on 24/08/2026

Identity UI

Built-in web UI for authentication and account management.

Anonymous · 24/08/2026

Identity UI

The IdentityUi module provides ready-made Symfony controllers, routes, and Twig templates for all authentication flows and account settings pages. You get a fully functional user-facing UI with minimal configuration.

What's included

Authentication

Route name	Path	What it does
`derafu_platform_identity_login`	`/login`	Login form
`derafu_platform_identity_register`	`/register`	Registration form
`derafu_platform_identity_logout`	`/logout`	Logout
`derafu_platform_identity_password_reset_request`	`/password-reset`	Forgot password
`derafu_platform_identity_password_reset`	`/password-reset/{token}`	Reset password
`derafu_platform_identity_magic_link_request`	`/magic-link`	Request magic link
`derafu_platform_identity_verify_email_required`	`/verify-email/required`	"Please verify your email" wall
`derafu_platform_identity_sudo`	`/sudo`	Re-authentication for sudo mode
`2fa_login`	`/2fa/check`	2FA code entry

Account settings

Section	What it manages
Account	Display name, avatar, locale, timezone
Security	Password change, 2FA enrollment, active sessions, login history
Email	Email change with verification
API Keys	Create, list, revoke API keys

Section	What it manages
JWT Tokens	List and revoke JWT tokens
Organizations	Create orgs, manage members, invitations
Teams	Create and manage teams within organizations

Webhook settings (Notifications module)

When the Notifications module is active, a Webhooks section appears in settings for managing per-user and per-organization webhook endpoints.

App settings (Apps module)

When the Apps module is active, an Apps section appears for installing and configuring integrations.

Configuration

```
# config/packages/derafu_platform.yaml
derafu_platform:
    identity_ui:
        # Your app's base Twig layout that the bundle's templates extend.
        parent_layout: 'layouts/app.html.twig'

        # Sender address for transactional emails (verification, reset, etc.)
        from_email: '%env(MAILER_FROM_EMAIL)%'
        from_name:  '%env(MAILER_FROM_NAME)%'

        # When true, users who haven't verified their email are redirected
        # to the verification wall on every request.
        require_email_verification: true
```

Extending the parent layout

Your `parent_layout` template must define the blocks that the bundle's templates fill in. At minimum:

```
{# templates/layouts/app.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}My App{% endblock %}</title>
    {% block stylesheets %}{% endblock %}
</head>
<body>
    {% block body %}{% endblock %}
```

```
    {% block javascripts %}{% endblock %}
</body>
</html>
```

The bundle's templates use `{% extends derafu_platform_parent_layout %}`, which resolves to your configured `parent_layout`.

Overriding templates

Override any template by creating a file at the same path under your app's `templates/` directory using Symfony's standard bundle template override:

```
templates/bundles/PlatformBundle/IdentityUi/page/auth/login.html.twig
```

Any template the bundle ships can be overridden this way — the bundle's templates are a starting point, not a constraint.

Locale negotiation

The `LocaleNegotiationListener` automatically negotiates the user's preferred locale on each request, matching against the `supported_locales` list. It checks (in order):

1. The user's saved locale preference (if authenticated).
2. The browser's `Accept-Language` header.
3. The Symfony default locale.

```
derafu_platform:
    identity:
        supported_locales:
            en: English
            es: Español
            fr: Français
```

Email verification wall

When `require_email_verification: true`, any authenticated user whose email is not yet verified is redirected to `derafu_platform_identity_verify_email_required` on every request. This enforcer runs as a `kernel.request` listener and can be scoped to specific firewalls.

A "Resend verification email" button is available on that page.

Settings section providers

The settings UI is assembled from registered section providers. Any bundle or module can contribute a settings section by implementing `SettingsSectionProviderInterface`:

```
use Derafu\PlatformBundle\Core\Contract\Service\SettingsSectionProviderInterface;
use Symfony\Component\DependencyInjection\Attribute\AutoconfigureTag;

#[AutoconfigureTag('derafu.settings_section_provider')]
final class BillingSettingsSection implements SettingsSectionProviderInterface
{
    public function getSection(): array
    {
        return [
            'code'     => 'billing',
            'label'    => 'Billing',
            'icon'     => 'credit-card',
            'route'    => 'app_settings_billing',
            'priority' => 50,
        ];
    }
}
```

The `SettingsExtension` Twig extension collects all registered sections and renders them in the settings sidebar.

Email delivery

The bundle uses Symfony Mailer for all transactional emails (verification, password reset, invitation, etc.). Configure your mailer transport in `config/packages/mailer.yaml`. The `from_email` and `from_name` values in `identity_ui` configuration are used as the sender on all emails.

Emails are triggered by subscribing to the bundle's events:

```
#[AsEventListener]
class IdentityMailerListener
{
    public function __invoke(UserRegistered $event): void
    {
        // Bundle's built-in IdentityMailerListener already handles this
        // if you leave it registered. Override it by replacing the alias.
    }
}
```

The bundle's built-in `IdentityMailerListener` handles all standard emails automatically. You can override individual email templates or swap the entire listener.

Last updated on 24/08/2026

Apps (plugin system)

Plugin/integration ecosystem with per-user, per-organization, and per-resource installations.

Anonymous · 24/08/2026

Apps (plugin system)

The Apps module provides a plugin/integration ecosystem for your SaaS. Apps are integrations or capabilities that users and organizations can enable — payment gateways, messaging platforms, storage services, email providers, and any custom capability your domain requires.

Concepts

Concept	Description
App	A registered integration (e.g. Stripe, Telegram, Dropbox).
Scope	Where an app can be installed: `user`, `organization`, or `resource`.
Installation	An app enabled for a specific user, org, or resource with config.
Capability	An interface an app can implement (e.g. payment, storage).
Catalog override	YAML configuration to adjust app metadata.

Defining an app

Create a service implementing `AppDefinitionInterface` (extend `AbstractApp` for convenience):

```
use Derafu\PlatformBundle\Apps\Abstract\AbstractApp;
use Derafu\PlatformBundle\Apps\Enum\AppScope;

final class StripeApp extends AbstractApp
{
    protected string $code = 'stripe';
    protected string $name = 'Stripe';
    protected string $description = 'Accept payments via Stripe.';
    protected string $icon = 'stripe';
    protected string $category = 'payments';

    protected array $scopes = [AppScope::Organization];

    // Keys to encrypt in the installation config
    protected array $sensitiveConfigKeys = ['secret_key', 'webhook_secret'];
```

```
    // Optional: a form key for the configuration form
    protected ?string $configFormKey = 'stripe_config';
}
```

The service is auto-discovered via the `AppDefinitionInterface` `AutoconfigureTag`. No manual service registration needed.

Scopes

An app can support one or more scopes:

Scope	Entity	Use case
`AppScope::User`	`BaseUserAppInstallation`	Personal integration (e.g. Telegram notifications)
`AppScope::Organization`	`BaseOrganizationAppInstallation`	Shared integration for the whole org
`AppScope::Resource`	`BaseResourceAppInstallation`	Integration for a specific resource

A multi-scope app (e.g. a storage app) can be installed at the user, organization, and/or resource level simultaneously.

Installing and managing apps

```
use Derafu\PlatformBundle\Apps\Contract\Service\AppInstallationManagerInterface;

// Install for a user
$installation = $appInstallationManager->installForUser(
    appCode: 'stripe',
    user: $currentUser,
    config: ['public_key' => 'pk_live_...', 'secret_key' => 'sk_live_...']
);

// Install for an organization
$installation = $appInstallationManager->installForOrganization(
    appCode: 'stripe',
    organization: $org,
    config: ['public_key' => 'pk_live_...', 'secret_key' => 'sk_live_...']
);

// Check if installed
$installed = $appInstallationManager->isInstalledForOrganization('stripe', $org);

// Get installation
$installation = $appInstallationManager->getForOrganization('stripe', $org);
```

```
// Uninstall
$appInstallationManager->uninstallForOrganization('stripe', $org);
```

Sensitive config keys declared in `$sensitiveConfigKeys` are transparently encrypted at rest by `AppConfigEncryptor`. Retrieval also decrypts transparently.

The app registry

The `AppRegistryInterface` collects all registered apps. It applies catalog overrides from configuration (scopes, flags, tags, supported resources):

```
use Derafu\PlatformBundle\Apps\Contract\Service\AppRegistryInterface;

$app = $appRegistry->get('stripe');       // AppDefinitionInterface
$apps = $appRegistry->all();              // all registered apps
$apps = $appRegistry->byScope(AppScope::User); // filtered by scope
$apps = $appRegistry->byCategory('payments'); // filtered by category
```

Catalog configuration

The YAML catalog section adjusts metadata of registered apps without modifying the app class. Useful for overriding defaults per-deployment:

```
derafu_platform:
    apps:
        catalog:
            stripe:
                scopes: [user, organization]
                flags: [featured]
            telegram:
                scopes: [user]
                flags: [featured]
            dropbox:
                scopes: [user, organization, resource]
                resources:
                    - App\Entity\Resource\Project
```

`flags` controls UI presentation (e.g. `featured` items appear prominently). `resources` lists which FQCN resource classes the app supports when installed at the resource scope.

Capabilities

Apps can implement capability interfaces to advertise features. The `CapabilityRegistryInterface`

answers “which installed apps for this organization support X?”:

```
use Derafu\PlatformBundle\Apps\Contract\Capability\PaymentGatewayInterface;

// Define a capability interface (app side)
interface PaymentGatewayInterface
{
    public function processPayment(int $amountCents, string $currency): string;
}

// An app implements it
final class StripeApp extends AbstractApp implements PaymentGatewayInterface
{
    public function processPayment(int $amountCents, string $currency): string
    {
        // Stripe API call...
    }
}

// Query at runtime
use Derafu\PlatformBundle\Apps\Contract\Service\CapabilityRegistryInterface;

$gateways = $capabilityRegistry->forOrganization(PaymentGatewayInterface::class,
$org);
// Returns all installed apps for $org that implement PaymentGatewayInterface
```

Resource apps

For apps that target a specific resource (e.g. “email provider for this Project”), create a resource installation entity:

```
use Derafu\PlatformBundle\Apps\Entity\BaseResourceAppInstallation;

#[ORM\Entity]
#[ORM\Table(name: 'project_app_installations')]
class ProjectAppInstallation extends BaseResourceAppInstallation
{
    #[ORM\ManyToOne(targetEntity: Project::class)]
    #[ORM\JoinColumn(nullable: false, onDelete: 'CASCADE')]
    private Project $project;

    public function getProject(): Project { return $this->project; }
    public function setProject(Project $project): static { $this->project = $project;
return $this; }
}
```

Settings UI

The Apps module registers a settings section automatically. Users and org admins can browse, install, configure, and uninstall apps from a dedicated settings page — no custom controllers needed.

Last updated on 24/08/2026

Notifications

Event-driven notifications via email, in-app messages, and webhooks.

Anonymous · 24/08/2026

Notifications

The Notifications module delivers notifications to users and organizations through multiple channels — email, in-app messages, and outbound webhooks. It is entirely event-driven: apps define which events generate notifications, and users control how they receive them.

Architecture overview

```
App event (Symfony EventDispatcher)
    └─► NotificationDispatcher
            ├─ Resolves recipients
            ├─ Checks user preferences
            └─ Routes to channels:
                    ├─► EmailChannel       → Symfony Mailer
                    ├─► InAppChannel       → InAppNotification entity
                    └─► WebhookChannel     → HTTP POST (async via Messenger)
```

Defining notification events

Implement `EventDefinitionProviderInterface` to register the events your module can generate:

```
use
Derafu\PlatformBundle\Notifications\Contract\Service\EventDefinitionProviderInterface;
use Derafu\PlatformBundle\Notifications\ValueObject\EventDefinition;

final class ProjectEventDefinitions implements EventDefinitionProviderInterface
{
    public function getEvents(): array
    {
        return [
            new EventDefinition(
                code: 'project.created',
                name: 'Project created',
                description: 'A new project was created in your organization.',
                channels: ['email', 'inapp', 'webhook'],
            ),
            new EventDefinition(
```

```
                code: 'project.updated',
                name: 'Project updated',
                description: 'A project you are a member of was updated.',
                channels: ['inapp', 'webhook'],
            ),
        ];
    }
}
```

The service is auto-discovered via the `EventDefinitionProviderInterface` `AutoconfigureTag`. Events automatically appear in the user preferences UI and as available webhook triggers.

Dispatching a notification

When something happens in your app, dispatch a notification via the `NotificationDispatcherInterface`:

```
use
Derafu\PlatformBundle\Notifications\Contract\Service\NotificationDispatcherInterface;
use Derafu\PlatformBundle\Notifications\ValueObject\NotificationEvent;

final class ProjectService
{
    public function __construct(
        private readonly NotificationDispatcherInterface $dispatcher,
    ) {}

    public function create(Project $project, UserInterface $creator): void
    {
        // ... create project logic ...

        $this->dispatcher->dispatch(new NotificationEvent(
            code: 'project.created',
            subject: $project,
            actor: $creator,
            data: [
                'project_name' => $project->getName(),
                'project_url'  => '/projects/' . $project->getId(),
            ],
        ));
    }
}
```

The dispatcher resolves recipients, checks each user's preferences, and routes to the appropriate channels.

Recipient resolvers

The dispatcher does not know who should receive a notification — that is your domain knowledge. Implement `NotificationRecipientResolverInterface` to define the recipient logic for each event:

```
use
Derafu\PlatformBundle\Notifications\Contract\Service\NotificationRecipientResolverInte
rface;
use Derafu\PlatformBundle\Notifications\ValueObject\NotificationEvent;

final class ProjectNotificationResolver implements
NotificationRecipientResolverInterface
{
    public function supports(NotificationEvent $event): bool
    {
        return str_starts_with($event->code, 'project.');
    }

    public function resolve(NotificationEvent $event): array
    {
        $project = $event->subject; // the Project entity

        // Return all org members who have access to this project
        return $project->getOrganization()->getMembers()->map(fn($m) =>
$m->getUser())->toArray();
    }
}
```

The service is auto-discovered via the `NotificationRecipientResolverInterface` `AutoconfigureTag`. Multiple resolvers can be registered — the dispatcher collects all unique recipients across resolvers that support the event.

Delivery channels

Email

Notifications are rendered using Twig templates and sent via Symfony Mailer. The bundle looks for templates at:

```
templates/notifications/email/{event_code}.html.twig
```

Create a template for each event that should generate an email:

```
{# templates/notifications/email/project.created.html.twig #}
{% extends 'emails/base.html.twig' %}
```

```
{% block subject %}New project: {{ data.project_name }}{% endblock %}

{% block body %}
<p>A new project <strong>{{ data.project_name }}</strong> was created.</p>
<a href="{{ data.project_url }}">View project</a>
{% endblock %}
```

In-app

In-app notifications are persisted as `InAppNotification` entities and displayed in your UI. The bundle provides a Twig helper to fetch and render them. Mark as read via:

```
$inAppNotificationRepository->markAsRead($notification);
$inAppNotificationRepository->markAllAsReadForUser($user);
```

Webhooks

Users and organizations configure webhook endpoints in the settings UI. When a notification is dispatched, the bundle:

1. Finds active webhook endpoints subscribed to the event code.
2. Renders the payload as JSON.
3. Dispatches a Messenger message for async delivery.
4. Signs the payload with `X-Webhook-Signature: sha256=<hmac>` using the endpoint's secret.
5. Records the delivery attempt in `WebhookDelivery`.

Signature verification (on the receiving end)

```
$signature = hash_hmac('sha256', $rawBody, $secret);
$expected = 'sha256=' . $signature;
hash_equals($expected, $request->headers->get('X-Webhook-Signature'));
```

User preferences

Users control which channels they receive for each event from the settings UI. The preferences are stored in `UserNotificationPreference` entities.

The dispatcher automatically respects preferences — if a user has disabled email notifications for `project.created`, the email channel is skipped for that user even if the email resolver resolves them as a recipient.

Async delivery via Messenger

Webhook delivery happens asynchronously via Symfony Messenger to avoid blocking the request. Configure the routing:

```
# config/packages/messenger.yaml
framework:
    messenger:
        routing:
            'Derafu\PlatformBundle\Notifications\Messenger\DeliverWebhookMessage':
async
```

Webhook delivery retention

Delivery logs are cleaned up automatically. Configure the retention policy:

```
derafu_platform:
    notifications:
        webhook_delivery_retention:
            max_age_days:     30   # delete logs older than 30 days
            max_per_endpoint: 500  # keep at most 500 logs per endpoint
```

Run the cleanup command periodically (daily cron recommended):

```
bin/console notifications:webhook-deliveries:cleanup
```

Configuration

```
derafu_platform:
    notifications:
        from_email: '%env(MAILER_FROM_EMAIL)%'
        from_name:  '%env(MAILER_FROM_NAME)%'

        inapp_notification_class:              App\Entity\Notif\InAppNotification
        user_webhook_endpoint_class:           App\Entity\Notif\UserWebhookEndpoint
        organization_webhook_endpoint_class:
App\Entity\Notif\OrganizationWebhookEndpoint
        user_webhook_delivery_class:           App\Entity\Notif\UserWebhookDelivery
        organization_webhook_delivery_class:
App\Entity\Notif\OrganizationWebhookDelivery
        user_notification_preference_class:
App\Entity\Notif\UserNotificationPreference
        user_digest_preference_class:          App\Entity\Notif\UserDigestPreference
```

Required entities

Create concrete entities extending the bundle's base classes:

```
// src/Entity/Notif/InAppNotification.php
namespace App\Entity\Notif;

use Derafu\PlatformBundle\Notifications\Entity\BaseInAppNotification;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\Table(name: 'notif_inapp')]
class InAppNotification extends BaseInAppNotification {}
```

Repeat for: `UserWebhookEndpoint`, `OrganizationWebhookEndpoint`, `UserWebhookDelivery`, `OrganizationWebhookDelivery`, `UserNotificationPreference`, `UserDigestPreference`.

See Installation for the full entity table.

Last updated on 24/08/2026

Integration with app resources

Traits and interfaces to plug app entities into identity, RBAC, and tenancy.

Anonymous · 24/08/2026

Integration with app resources

The bundle ends where business logic begins. Entities like Project, Invoice, or Document belong to the app — but they often need to participate in the identity model (belong to an org, have collaborators with roles). The bundle provides **interfaces and traits** for this.

OrganizationOwnedInterface + Trait

“This entity belongs to an organization.”

```
use Derafu\PlatformBundle\Identity\Contract\Integration\OrganizationOwnedInterface;
use Derafu\PlatformBundle\Identity\Entity\Trait\OrganizationOwnedTrait;

#[ORM\Entity]
#[ORM\Table(name: 'projects')]
class Project implements OrganizationOwnedInterface
{
    use OrganizationOwnedTrait;

    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private string $name;

    // ... app-specific fields
}
```

The trait adds a `ManyToOne` to `OrganizationInterface` (resolved to your concrete Organization at runtime). The field is **nullable** by default — an entity can exist without an organization (personal account pattern). Add `nullable: false` if org ownership is mandatory.

The `ResourcePermissionVoter` uses this interface for cascading: if a user doesn’t have direct resource-level access, the voter automatically checks their org-level permission on the resource’s owning organization.

UserOwnedInterface + Trait

“This entity belongs directly to a user.”

```
use Derafu\PlatformBundle\Identity\Contract\Integration\UserOwnedInterface;
use Derafu\PlatformBundle\Identity\Entity\Trait\UserOwnedTrait;

class Project implements OrganizationOwnedInterface, UserOwnedInterface
{
    use OrganizationOwnedTrait;
    use UserOwnedTrait;
    // ...
}
```

Both traits are composable. A single entity can be owned by a user OR by an organization (or both, with app logic deciding which applies).

The `IdentityTenantFilter` uses `UserOwnedInterface` to automatically append `WHERE owner_id = :id` to queries when a user context is active.

ResourceAccessibleInterface + BaseResourceAccess

“This entity supports per-resource collaborators with roles.”

Step 1: Create the access pivot entity

```
use Derafu\PlatformBundle\Identity\Entity\BaseResourceAccess;

#[ORM\Entity]
#[ORM\Table(name: 'project_access')]
#[ORM\UniqueConstraint(columns: ['user_id', 'project_id'])]
class ProjectAccess extends BaseResourceAccess
{
    #[ORM\ManyToOne(targetEntity: Project::class, inversedBy: 'accesses')]
    #[ORM\JoinColumn(name: 'project_id', nullable: false, onDelete: 'CASCADE')]
    private Project $project;

    public function __construct(UserInterface $user, RoleInterface $role, Project
$project)
    {
        parent::__construct($user, $role);
        $this->project = $project;
    }

    public function getProject(): Project { return $this->project; }
}
```

Each resource type gets its own table — real FK integrity, no polymorphic hacks.

Step 2: Implement the interface on the resource

```
use Derafu\PlatformBundle\Identity\Contract\Integration\ResourceAccessibleInterface;

class Project implements OrganizationOwnedInterface, ResourceAccessibleInterface
{
    use OrganizationOwnedTrait;

    #[ORM\OneToMany(mappedBy: 'project', targetEntity: ProjectAccess::class,
                    cascade: ['persist', 'remove'], orphanRemoval: true)]
    private Collection $accesses;

    public function __construct()
    {
        $this->accesses = new ArrayCollection();
    }

    public function getResourceAccesses(): Collection
    {
        return $this->accesses;
    }
}
```

Step 3: Use it

```
// In a controller:
#[IsGranted('write', subject: 'project')]
public function edit(Project $project): Response { ... }
```

The `ResourcePermissionVoter` kicks in, iterates the Project's access entries for the current user, and checks permissions. If no direct access is found, it cascades to org-level and then global.

TeamAccessibleInterface

"This entity's access can be granted to teams."

Implement `TeamAccessibleInterface` and add a `BaseTeamResourceAccess` pivot per resource type (similar to `BaseResourceAccess`). The `PermissionChecker` cascade checks team memberships between direct resource access and org-level.

Summary

Interface	Trait	What it adds
`OrganizationOwnedInterface`	`OrganizationOwnedTrait`	`$organization` ManyToOne
`UserOwnedInterface`	`UserOwnedTrait`	`$owner` ManyToOne
`ResourceAccessibleInterface`	(manual collection)	Per-resource RBAC
`TeamAccessibleInterface`	(manual pivot entity)	Team-level RBAC
`BaseResourceAccess`	—	Access pivot MappedSuperclass
`BaseTeamResourceAccess`	—	Team access pivot MappedSuperclass

All integration interfaces live under `Derafu\PlatformBundle\Identity\Contract\Integration\`.

Last updated on 24/08/2026

Events

All dispatched events and how to subscribe to them.

Anonymous · 24/08/2026

Events

All identity events live under `Derafu\PlatformBundle\Identity\Event\` and extend `Symfony\Contracts\EventDispatcher\Event`. The bundle dispatches them after successful operations — subscribe to them to send emails, log activity, seed data, or trigger side effects.

Subscribing

```
use Derafu\PlatformBundle\Identity\Event\UserRegistered;
use Symfony\Component\EventDispatcher\Attribute\AsEventListener;

#[AsEventListener]
class SendConfirmationEmail
{
    public function __invoke(UserRegistered $event): void
    {
        $user = $event->user;
        $token = $event->verificationTokenPlaintext;

        // Build URL and send email...
    }
}
```

User lifecycle events

Event	When	Key fields
`UserRegistered`	After `Registrar::register()`	`user`, `verificationTokenPlaintext`
`UserEmailVerified`	After `EmailVerifier::verify()`	`user`
`PasswordResetRequested`	After `PasswordResetter::requestReset()` (only if user exists)	`user`, `verificationTokenPlaintext`
`UserPasswordChanged`	After `PasswordResetter::reset()`	`user`
`EmailChangeRequested`	After `EmailChanger::requestChange()`	`user`, `newEmail`, `verificationTokenPlaintext`
`UserEmailChanged`	After `EmailChanger::confirmChange()`	`user`, `oldEmail`, `newEmail`

Event	When	Key fields
`MagicLinkRequested`	After `MagicLinkManager::requestLink()` (only if user exists)	`user`, `plaintextToken`
`AccountLocked`	After `AccountLockManager::handleFailedLogin()` when threshold reached	`user`, `lockedUntil`, `failedAttempts`
`AccountUnlocked`	After `AccountLockManager::unlock()`	`user`

Note on `PasswordResetRequested` and `MagicLinkRequested`: not dispatched when the email does not belong to any user — this is intentional to avoid leaking account existence.

Note on `UserEmailChanged`: carries the `oldEmail` so a listener can notify the old address ("if this wasn't you, contact support").

Organization events

Event	When	Key fields
`OrganizationCreated`	After `OrganizationManager::create()`	`organization`, `owner`
`OrganizationMemberAdded`	After `OrganizationManager::addMember()` or invitation accept	`membership`
`OrganizationMemberRemoved`	After `OrganizationManager::removeMember()`	`organization`, `user`
`OrganizationOwnershipTransferred`	After `OrganizationManager::transferOwnership()`	`organization`, `previousOwner`, `newOwner`

Note on `OrganizationMemberRemoved`: the membership entity is already deleted when listeners run — the event carries `organization` and `user` separately.

Invitation events

Event	When	Key fields
`InvitationCreated`	After `InvitationManager::invite()`	`invitation`, `plaintextToken`
`InvitationAccepted`	After `InvitationManager::accept()`	`invitation`, `membership`
`InvitationRevoked`	After `InvitationManager::revoke()`	`invitation`

Note on `InvitationCreated`: the `plaintextToken` is the only chance to build the accept URL. Once the request ends, only the hash in the database survives.

Team events

Event	When	Key fields
`TeamCreated`	After `TeamManager::create()`	`team`
`TeamMemberAdded`	After `TeamManager::addMember()`	`team`, `user`
`TeamMemberRemoved`	After `TeamManager::removeMember()`	`team`, `user`

Events that carry plaintext tokens

These events expose a plaintext token that **must not be logged or persisted**. They exist solely so the app's mailer can build a URL and send it:

- `UserRegistered.verificationTokenPlaintext`
- `PasswordResetRequested.verificationTokenPlaintext`
- `EmailChangeRequested.verificationTokenPlaintext`
- `MagicLinkRequested.plaintextToken`
- `InvitationCreated.plaintextToken`

Last updated on 24/08/2026

Query Bundle

Query Bundle

Anonymous · 24/08/2026 · #php, #symfony

Query Bundle

This bundle integrates derafu/query — an expressive path-based query builder for PHP — into a Symfony application by registering its core services in the dependency injection container. If you are not yet familiar with derafu/query, reading its documentation first will give you the necessary context to understand what the bundle exposes.

Requirements

- PHP 8.5 or higher
- Symfony 8.0 or higher (`symfony/framework-bundle`)

Installation

Install the bundle via Composer:

```
composer require derafu/symfony-query-bundle
```

Because the `derafu/query` package is currently tracked on its `dev-main` branch, you may need to set the minimum stability in your `composer.json` to `dev` if it is not already:

```
{
    "minimum-stability": "dev",
    "prefer-stable": true
}
```

Enabling the Bundle

If your application uses Symfony Flex and has bundle auto-discovery enabled, the bundle will be registered automatically. Otherwise, add it manually to `config/bundles.php`:

```
return [
```

```
    // ... other bundles
    Derafu\QueryBundle\QueryBundle::class => ['all' => true],
];
```

No other files need to be created or modified for the bundle to function. The services are private by default and are intended to be consumed via autowiring.

Configuration

The bundle works out of the box without any configuration. It automatically resolves the path to the `operators.yaml` file that ships with `derafu/query`.

If you need to use a custom operators file — for example, to add your own operators or override the defaults — create the file `config/packages/derafu_query.yaml` in your application and set the `operators_path` option:

```
derafu_query:
    operators_path: '%kernel.project_dir%/config/query/operators.yaml'
```

When `operators_path` is omitted or set to `null`, the bundle falls back to the operators file bundled with `derafu/query`.

Available Services

All services are registered with autowiring enabled. Inject them by their interface in your controllers, services, or repositories.

Operator system

Interface	Description
`Derafu\Query\Operator\Contract\OperatorLoaderInterface`	Reads operator definitions from a YAML file or array and produces `Operator` objects.
`Derafu\Query\Operator\Contract\OperatorManagerFactoryInterface`	Builds an `OperatorManager` instance from a given operators file path.
`Derafu\Query\Operator\Contract\OperatorManagerInterface`	Provides access to the configured set of operators at runtime.

Parsers

Interface	Description
`Derafu\Query\Filter\Contract\PathParserInterface`	Parses a property path string (e.g. `user.address.city`).
`Derafu\Query\Filter\Contract\ExpressionParserInterface`	Parses a single filter expression (e.g. `name:eq:John`).
`Derafu\Query\Filter\Contract\CompositeExpressionParserInterface`	Parses a composite filter expression combining multiple conditions.
`Derafu\Query\Filter\Contract\FilterParserInterface`	High-level parser that coordinates path and expression parsing.

Doctrine ORM bridge

Interface	Description
`Derafu\Query\Bridge\Contract\QueryBuilderConditionApplierInterface`	Applies a parsed filter condition to a Doctrine ORM `QueryBuilder` instance.

API Platform bridge

The bundle also registers `Derafu\Query\Bridge\ApiPlatform\SmartFilter` and tags it as an `api_platform.filter`. This filter accepts derafu/query expressions as `QueryParameter` values. You can reference it directly in your API Platform resources:

```
use ApiPlatform\Metadata\QueryParameter;
use Derafu\Query\Bridge\ApiPlatform\SmartFilter;

#[QueryParameter(name: 'status', property: 'status', filter: SmartFilter::class)]
```

The `SmartFilter` service is only meaningful if `api-platform/core` is installed. If the package is absent the service definition will still be loaded, but it will not be usable.

Basic Usage

Injecting parsers

```
use Derafu\Query\Filter\Contract\FilterParserInterface;
```

```
final class ProductRepository
{
    public function __construct(
        private readonly FilterParserInterface $filterParser,
    ) {}

    public function findByFilter(string $filterString): array
    {
        $filter = $this->filterParser->parse($filterString);
        // use $filter to build your query
    }
}
```

Applying conditions to a Doctrine ORM QueryBuilder

```
use Derafu\Query\Bridge\Contract\QueryBuilderConditionApplierInterface;
use Derafu\Query\Filter\Contract\FilterParserInterface;

final class ProductRepository extends ServiceEntityRepository
{
    public function __construct(
        ManagerRegistry $registry,
        private readonly FilterParserInterface $filterParser,
        private readonly QueryBuilderConditionApplierInterface $conditionApplier,
    ) {
        parent::__construct($registry, Product::class);
    }

    public function search(string $filterString): array
    {
        $qb = $this->createQueryBuilder('p');
        $filter = $this->filterParser->parse($filterString);
        $this->conditionApplier->apply($qb, $filter);

        return $qb->getQuery()->getResult();
    }
}
```

Refer to the derafu/query documentation for the full expression syntax, the list of available operators, path navigation rules, and security recommendations.

Last updated on 24/08/2026

Form Bundle

Form Bundle

Anonymous · 24/08/2026 · #php, #symfony

Form Bundle

This bundle integrates the derafu/form declarative forms library with Symfony. Forms are defined as plain PHP files returning a JSON Schema + UI Schema structure rather than PHP builder classes, and the bundle wires the loader, renderer, and Twig extension into the Symfony container automatically.

Requirements

- PHP 8.5 or higher
- Symfony 8.0 or higher (framework-bundle + twig-bundle)
- Symfony AssetMapper (required for the JavaScript widgets)

Installation

Install the package via Composer:

```
composer require derafu/symfony-form-bundle
```

Because the bundle does not ship a Symfony Flex recipe, you must register it manually in `config/bundles.php`:

```
return [
    // ...
    Derafu\FormBundle\FormBundle::class => ['all' => true],
];
```

That is the only mandatory step. Everything else — asset mapper path registration, `importmap.php` entries, and form-path discovery across bundles — is handled automatically during container compilation.

What the bundle configures automatically

When the container is compiled (either on first boot or after `cache:clear`), the bundle performs three automatic setup tasks.

Asset mapper path. The extension registers `vendor/derafu/form/resources/js` under the `derafu-form` namespace in Symfony's asset mapper, so the JavaScript widgets become available as logical paths like `derafu-form/collection-widget.js`.

ImportMap entries. A compiler pass scans that same JS directory and adds any missing entries to `importmap.php`. The operation is idempotent: entries already present are never duplicated. If your project does not use AssetMapper and therefore has no `importmap.php`, this step is silently skipped.

Form-path discovery. Another compiler pass iterates over every registered bundle and checks whether it contains a `resources/forms/` directory. Any directory found is registered with the `PhpFormLoader` in bundle registration order. The application's own forms path (see the configuration section below) is added last, giving it the highest priority so the application can override any form defined by a bundle.

Optional configuration

The bundle works out of the box with its defaults. If you need to change either option, create `config/packages/derafu_form.yaml`:

```
derafu_form:
    twig_prefix: 'derafu_'
    forms_path: '%kernel.project_dir%/resources/forms'
```

`twig_prefix` controls the prefix of every Twig function registered by the bundle. The default `derafu_` avoids collisions with Symfony's own `symfony/form` functions (`form()`, `form_start()`, etc.). With the default prefix the functions are called `derafu_form()`, `derafu_form_start()`, `derafu_form_end()`, and so on. The value must not be empty.

`forms_path` is the directory where the application stores its own form definition files. It defaults to `resources/forms/` at the project root. This directory does not need to exist at installation time; the compiler pass only registers it if it is present when the container is compiled.

Writing form definition files

Form definitions are plain PHP files that return either an array or a callable. Place them inside `resources/forms/` at the project root (or inside `resources/forms/` within any registered bundle).

A static definition returns an associative array with `schema`, `uischema`, and optionally `data` keys:

```
// resources/forms/auth/login.form.php
return [
    'schema' => [
        'type' => 'object',
        'properties' => [
            'email'    => ['type' => 'string', 'format' => 'email'],
            'password' => ['type' => 'string'],
        ],
        'required' => ['email', 'password'],
    ],
    'uischema' => [
        'type'     => 'VerticalLayout',
        'elements' => [
            ['type' => 'Control', 'label' => 'Email',    'scope' =>
'#/properties/email'],
            ['type' => 'Control', 'label' => 'Password', 'scope' =>
'#/properties/password'],
        ],
    ],
    'data' => ['email' => '', 'password' => ''],
];
```

A dynamic definition returns a callable that receives a `$context` array and returns the same structure. This is useful when the form data or schema depends on values only known at runtime:

```
// resources/forms/user/edit.form.php
return function (array $context = []): array {
    return [
        'schema' => [...],
        'data'   => ['name' => $context['name'] ?? ''],
    ];
};
```

Files are addressed by their path relative to the forms directory, without the `.form.php` suffix. A file at `resources/forms/auth/login.form.php` is loaded with the key `auth/login`.

Loading and rendering forms in a controller

Inject `FormLoaderInterface` to load a form definition and `FormDataProcessorInterface` to validate a submission:

```
use Derafu\Form\Contract\FormDataProcessorInterface;
use Derafu\Form\Contract\FormLoaderInterface;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
```

```
final class LoginController extends AbstractController
{
    public function __invoke(
        Request $request,
        FormLoaderInterface $formLoader,
        FormDataProcessorInterface $processor,
    ): Response {
        $form = $formLoader->load('auth/login');

        if ($request->isMethod('POST')) {
            $result = $processor->process($form, $request->request->all());

            if ($result->isValid()) {
                $data = $result->getProcessedData();
                // handle valid submission...
            }

            $form = $form->withData($result->getData(), $result->getErrors());
        }

        return $this->render('auth/login.html.twig', ['form' => $form]);
    }
}
```

In the Twig template, render the form using the prefixed helper functions:

```
{{ derafu_form(form) }}
```

Or, if you need to wrap the form in custom markup, use the start/end pair:

```
{{ derafu_form_start(form) }}
    {# custom markup #}
{{ derafu_form_end(form) }}
```

JavaScript widgets

The bundle ships three JavaScript modules, all registered automatically in the import map under the `derafu-form/` namespace.

`derafu-form/collection-widget.js` handles dynamic array/collection fields: adding new items, removing existing ones, and keeping the add/remove buttons enabled or disabled according to configured min/max constraints. It has no external dependencies and initialises itself on `DOMContentLoaded`. Import it on any page that contains a collection field:

```
import 'derafu-form/collection-widget.js';
```

`derafu-form/html-editor.js` integrates the Summernote WYSIWYG editor into `textarea` fields marked for rich-text editing. It requires jQuery and Summernote to be loaded beforehand — add them to your import map (or load them from a CDN) before importing this module.

`derafu-form/json-editor.js` integrates the JSONEditor library for fields that hold structured JSON. It requires the JSONEditor library to be available globally before this module runs.

Unless your forms include collection, rich-text, or JSON fields you do not need to import any of these modules manually; the base form rendering has no JavaScript dependency.

Last updated on 24/08/2026