The CRON Way of Building Software: A Development Manifesto (Definitive, Unabridged)

If we are to succeed, we must learn what success feels like.

It does not feel like thundering applause in the face of revolutionary code or ingeniously abstracted classes. Rather, it feels like a slow — almost effortless — journey achieved through curated, repeatable, and well-understood practices.

It is better to be **consistent** than to be clever. That is the mark of wisdom.

1. Core Philosophy: Velocity with Confidence

This document outlines the principles we follow to build robust, maintainable, and high-quality software. Our philosophy is rooted in a pragmatic balance between **Development Velocity** and **Long-Term Confidence**.

We prioritize moving quickly, but not at the cost of creating a brittle or opaque system. We achieve this through:

- **Centralized, Robust Logic:** Logic has a specific, predictable home. The model is the source of truth for its domain, and any othe logic goes into independent, modular, self-contained services. An object's "list of things to do when being created" does not belong in `perform_create`, it belongs in an object's internal `on_create`.
- **Declarative Composition using Mixins:** We heavily favor mixins to compose and reuse functionality across models, serializers, and views. It is our primary tool for keeping code DRY and organized.
- **Confidence Through Integration:** We test our systems as they are used in the real world—from the outside in.
- **Convention over Configuration:** We use a consistent project structure and a shared set of patterns to eliminate ambiguity and cognitive overhead.
- **Pragmatism over Dogma:** These rules are designed to serve us, not the other way around. We understand when a simple solution is the right solution. This manifesto will often use works like "must" which are to be taken with a grain of salt.

2. Project Structure & Entry Points

Consistency starts with a predictable structure. All projects forked from `cron-django-template` adhere to this.

Rule: If Needed, Split Complexity into Multiple Apps. As a project grows, its logic should be split into multiple, well-defined Django apps. A very large, single, monolithic app is an anti-pattern. However, premature optimization is the root of all evil. Only truly split if domains are independent or if the app has become a truly large monolith.

> ***Rationale:*** *Multiple apps enforce clear boundaries between different domains of the project, improving modularity and making the codebase easier to navigate and maintain.*

Rule: Code is Organized by Function. Within each Django app, we maintain a clear separation of concerns:

- `api/`: Contains all Django Rest Framework `APIView` or `ViewSet` classes, typically one view class per file.
- `serializers/`: Contains serializer classes, one per file, unless they are simple and co-located (see API section).
- `services/`: Contains business logic classes (`SomethingService`) that are too complex for a model, interact with external systems, or contain rather model-orthogonal logic (e.g. pemissions)
- `tasks.py`: Contains all background tasks decorated for `RQ`.

Rule: The `scripts/` Directory is the Only Entry Point. No developer should ever need to run `python manage.py` or `docker-compose` directly. We provide a set of standardized shell scripts that handle the environment's complexity. These include:

- `scripts/deploy.sh redeploy-fg`: The single command to build, migrate, and run the entire project. It is idempotent and safe to run anytime.
- `scripts/manage.sh <command>`: Our wrapper for Django management commands (e.g., `scripts/manage.sh shell`).
- `scripts/make-migrations.sh`: The dedicated script for creating new migrations.
- `scripts/run-just-tests.sh`: The script to run the test suite.

The `scripts/deploy.sh` script wraps `docker-compose` and any arguments are forwarded to it too.

```
# GOOD: The one command to set up and run everything.
scripts/deploy.sh redeploy

# GOOD: The standardized way to run management commands.
scripts/manage.sh shell

# BAD: Direct, environment-specific commands.
python manage.py runserver
docker-compose up -d --build
```

> ***Rationale:*** *This abstraction ensures that every developer interacts with every project in the exact same way, regardless of its specific Docker or environment setup. It lowers the barrier to entry and eliminates "it works on my machine" issues. It also ensures deployment follows standardized practice and integrates easly with our CI/CD templates.*

Rule: Use Standardized Tooling.

- **Dependency Management:** We use **Poetry** for robust dependency management.
- **Containerization:** We use **multi-stage Docker builds** to create optimized, lean production images.
- **Leverage Well-Maintained Third-Party Packages:** Do not reinvent the wheel. For established patterns, use trusted packages. A prime example is using `django-polymorphic` to implement polymorphic models.

3. The Data Layer: "Fat Models" with Rich QuerySets

Rule: All models MUST inherit from `TimestampedModelMixin`. Every model must include our standard mixin (e.g., `class MyModel(TimestampedModelMixin, models.Model):`) to

ensure all tables have `created_at`, `updated_at`, and other essential base fields automatically. This is a prime example of our mixin-first philosophy.

Rule: Models are "Fat," and QuerySets are Rich (*What, not how*) We encapsulate as much domain-specific logic as possible within the Model and its Manager/QuerySet.

- **Model Methods:** Contain logic related to a single model instance (e.g., `order.calculate_total()`).
- **Object Information Deduction:** Small properties or functions that help us deduce information about the object should live in the object itself.
- **Presentation Logic Centralization:** Even small presentation things, like getting a shorter version of the object's name, should be centralized in the model rather than external layers.
- **QuerySet Methods:** Contain logic for filtering or acting on a collection of models. These methods should read like sentences and abstract away the underlying query complexity.

```
# GOOD: Abstract, readable, and chainable
class BookingQuerySet(models.QuerySet):
    def of_user(self, user):
        return self.filter(user=user)

    def should_trigger_reminder(self):
        # The complex logic of what "should trigger" means is hidden here
        return self.filter(status='CONFIRMED', reminder_sent=False,
start_time__lte=now() + timedelta(days=1))

    def with_prefetched_guest_details(self):
        # Performance logic is encapsulated here
        return self.prefetch_related("guests")

# Usage in a view:
Bookings.objects.of_user(request.user).should_trigger_reminder()

# BAD: Leaking implementation details and performance logic into the calling code
Booking.objects.filter(
    user=request.user, status='CONFIRMED', reminder_sent=False, ...
).prefetch_related("guests")
```

> ***Rationale:*** *This pattern—"what, not how"—makes our business logic reusable, testable, and highly readable. We separate the idea of what is being done from how it is being done. Even though a field may exist in the database, we don't want whoever is using the object to know that. We want to deeply encapsulate this knowledge. When the definition of "should trigger reminder" changes, we change it in exactly one place. Performance optimizations (like `select_related` or `prefetch_related`) are also encapsulated within dedicated QuerySet methods, keeping them out of the API layer.*

Rule: APIs, admin, management do not contain business logic: it lies in the model and its services A model is an **independent**, **self-contained**, unit that acts as the **souce of truth** and **knower of its capabilities**. The API layer, the admin, and management commands are all different interfaces that simply read/write from/to a model. This is why models (and their services) contain the business logic. An API should never have

to override `perform_create` or manually call an `on_create`, much like the admin shouldn't do it. All the API and admin are doing is creating objects, and the model should internally detect that it's being created and call the internal `on_create`, which may, itself, instantiate some complex service (e.g. to send e-mails). The model is the one single source of truth.

Rule: The Model is the Authoritative Interface for its Domain. A model should be as self-contained as possible, managing not just its own state but also the lifecycle of its closely-related children.

```
# GOOD: The ToDoList model provides a clean interface for adding items.
class ToDoList(models.Model):
    # ... fields ...

    def add_item(self, text: str) -> "ToDoItem":
        # The caller doesn't know or care if this creates a new row
        # in a ToDoItem table or appends to an internal JSONField.
        return ToDoItem.objects.create(list=self, text=text)
```

Rule: Use the `raise_exc` Pattern for Flexible Error Handling. Many QuerySet and business logic methods should include a `raise_exc` parameter to make exception raising optional.

- **QuerySet Validation:** QuerySet methods that validate data should accept a `raise_exc` flag.
- **Business Logic Methods:** Methods that check constraints or limits should use the `raise_exc` pattern.
- **Testing Flexibility:** This pattern is especially useful in testing, where sometimes you want to check the boolean result and sometimes you want the exception.

```
# GOOD: QuerySet method with raise_exc pattern
class BookingQuerySet(models.QuerySet):
    def validate_ids_exist(self, booking_ids: List[int], raise_exc=False) -> bool:
        missing_ids = set(booking_ids) -
set(self.filter(id__in=booking_ids).values_list('id', flat=True))
        if missing_ids:
            if raise_exc:
                raise ValidationError(f"Booking IDs not found: {', '.join(map(str,
missing_ids))}")
            return False
        return True

# GOOD: Business logic method with raise_exc pattern
def can_actor_submit_slips(self, actor: SessionActor, raise_exc=False) -> bool:
    """Checks if this actor can submit slips."""
    # Check limits...
    daily_count = BettingSlip.objects.of_actor(actor).daily().exclude_failed().count()

    if daily_count >= settings.ANONYMOUS_USER_MAX_SLIPS_PER_DAY:
        if raise_exc:
            raise UsageLimitExceeded(
                f"Daily limit ({settings.ANONYMOUS_USER_MAX_SLIPS_PER_DAY} slips)
```

```
reached for free users."
            )
        return False
    return True
```

> ***Rationale:*** *The `raise_exc` pattern provides flexibility. We don't like our models to randomly raise things unless we tell them to. This allows the same method to be used for both validation checking (boolean result) and enforcement (exception raising).*

Rule: Avoid Django Signals (with One Exception). We are completely against using Django signals for business logic. The only exception is the deletion signal when absolutely necessary.

> ***Rationale:*** *Signals create implicit, hard-to-trace execution flows that make debugging difficult and violate the principle of explicit control flow.*

Rule: The `save()` Method is the Ultimate Gatekeeper. Critical, must-pass validations belong in the `save()` method to guarantee data integrity. Prefer overriding the `save()` method instead of using signals for model lifecycle events.

4. The Logic Layer: Services

Rule: Use Services for Complex Orchestration, Orthogonal Logic, and All External Interactions. A piece of logic moves from a model method to a Service class when:

1. **It becomes too large or complex:** If a model method starts orchestrating a multi-step process across several different models, it's a candidate for a service.
2. **It interacts with anything external:** All communication with third-party APIs, email servers, file systems, etc., MUST be encapsulated within a service.
3. **It deals with mostly model-orthogonal logic**: Permission classes, for example, *may* be services. In a project, we have an `AnalysisPermissionService` with the signature `def can_actor_analyse_slip(self, actor: SessionActor, slip: "BettingSlip", raise_exc=False) -> bool`. Since this logic is so self-contained, it can go to a service.

```
# services/llm_service.py
class LLMInformationStorageService:
    def __init__(self, llm_response: dict):
        self.data = llm_response

    def store_in_model(self, product: Product):
        # Complex logic for parsing the LLM response and mapping it
        # to our internal Product model fields.
        product.description = self.data.get("description")
        # ... more logic
        product.save()
```

> ***Rationale:*** *This keeps our models clean and focused on their core domain. It also makes external interactions explicit and easy to mock for testing. Services are always classes ending in `Service` and live in the `services/` directory.*

Rule: Embrace Advanced Service Patterns for Complex Workflows.

- **Orchestration:** For complex processes, a service should act as an orchestrator, coordinating smaller, single-responsibility components (e.g., a `SlipEvaluator` service managing a sequence of individual `PipelineStep` objects).
- **Configuration-Driven Behavior:** For *truly complex and highly variable* workflows, a service's behavior can be defined in external configuration files (e.g., YAML "profiles"). This is an advanced pattern reserved for situations where flexibility is required without code changes.

Rule: Services Must Be Configurable and Testable by Design. The configuration of a service should be explicit and passed via its constructor. These parameters must have default values that are pulled from Django's `settings`.

```
# GOOD: Configurable, testable, and explicit.
from django.conf import settings

class ExternalAPIClient:
    def __init__(
        self,
        api_key: str = None,
        timeout: int = None,
    ):
        self.api_key = api_key if api_key is not None else settings.EXTERNAL_API_KEY
        self.timeout = timeout if timeout is not None else
settings.EXTERNAL_API_TIMEOUT
        # ... setup ...

# In tests, this allows for easy instantiation with mock values:
client = ExternalAPIClient(api_key="test_key")

# Note: We prefer to **resolve settings inside the constructor**, because, otherwise,
# override_settings might not resolve correctly, and we also want to be able to use
it.
```

> ***Rationale:*** *This makes services incredibly flexible and easy to test with or without `override_settings`. The configuration of a service should be explicit and passed via its constructor. These parameters must have default values that are pulled from Django's `settings`.*

Rule: Prefer to wrap services within easy to read model methods. Even when some logic is complex or orthogonal to the model and warrants a service class, it is good practice to expose it in the model. As an example, even though we may offload the logic of analysing a `BettingSlip` to its own `AnalysisPermissionService`, it is still recommended to wrap a call to it within the model like so:

```
    @property
    def can_owner_actor_validate(self):
        return AnalysisPermissionService().can_actor_analyse_slip(self.actor, self)
```

This way, users of a `betting_slip` can do `betting_slip.can_owner_actor_validate()` without needing to know the implementation details. **They should not have to know if this is implemented as a service with 500 lines, or as a one-liner!**

> ***Rationale:*** *This ensures that **developers on a codebase can always go to the model to know how to act on it and what it allows for**, even though the **actual operations***

> ***are implemented elsewhere**. It means it's easy for developers to know where to look for things, and also easy to write them in self-contained ways.*

An **alternative approach** would be to create a `BettingSlipPermissionMixin` which simultaneously contains the logic of the service and the `can_owner_actor_validate` method.

5. The API Layer: Views, Serializers, and URL rules

The API layer is a thin, declarative interface responsible for request/response handling and validation. We prefer to add reusable functionality to views and serializers via **mixins**.

Do not think of an API as where business logic lives. Most business logic lives in the models and its services. This ensures that API, admin, and management commands all get their business logic "for free".

Rule: Separate Actions from Queries in URL Structure. In addition to following established REST practices (POST/DELETE/GET `/resources/`, PATCH/PUT/GET `/resources/{id}`), we use a specific URL structure to make the intent of an endpoint immediately clear.

- **Commands/Actions:** Endpoints that change state. They are always `POST` requests.
  - `POST /users/{id}/actions/reset-password/`
- **Complex Queries:** Endpoints for custom, complex lookups that don't fit standard filtering.
  - `GET /matches/queries/resolve-match/?team1=A&team2=B`

> ***Rationale:** This convention provides a clean, predictable API structure that separates state-changing commands from data-retrieving queries. All URLs use **dashes**, not underscores. The `actions/` and `queries/` are predictable, reserved parts of our APIs. Note that this namespacing only applies to what is otherwise not possible with typical REST CRUD.*

Rule: Serializers Validate and Shape; They Do Not Create Logic. The primary role of a serializer is to validate incoming data and define the shape of outgoing data.

- **Avoid `SerializerMethodField`:** We dislike serializer method fields and they should be used very, very, very sparingly. If you need a computed value, add it as a property or method on the model itself. The model is the source of truth.
- **Serializers Are Tight Validators:** Serializers should be tight validators of information, not logic containers. The only **exception** to this rule is for projects where serialization itself is very fine-grained and permission-based, in which case we encourage the use of our custom `PermissionedModelSerializer`.
- **Reusability is the Default:** Strive to reuse existing serializers. Create new, more specific serializers only when variations in fields or validation rules become necessary. For context, here is a list of seralizers and mixins we already have, and which you should check out: `DisableUniqueValidatorsMixin`, `MultiQuerySetMixin`, `MultiSerializerMixin`, `PermissionedModelSerializer`, and `ReadOnlyModelSerializer`.
- **Pragmatism for Simple Cases:** For simple endpoints with validation-only needs, it is acceptable and preferred to define `InputSerializer` and `OutputSerializer` classes directly inside the `APIView` file. This improves locality and reduces boilerplate.

> ***Rationale:*** *Keeping serializers "thin" reinforces the "Fat Model" pattern. Logic lives in one place, making the system easier to reason about. SerializerMethodFields scatter computed logic across serializers instead of centralizing it in models. Reusing mixins enforces a declarative approach, where we know* ***what*** *to use, regardless of* ***how*** *it works (less cognitive overhead).*

Rule: Encapsulate Authorization in Custom Permission Classes. Complex authorization logic (e.g., "is this user a premium member of the organization that owns this resource?") belongs in a custom DRF `Permission` class, not in the view. And if it becomes too complex, it should be offloaded to a service (permissions are quite orthogonal concerns to models). **Keep permission classes simple and short**.

> ***Rationale:*** *This keeps view logic clean and focused on orchestration, while making security logic reusable and testable in isolation.*

```
# GOOD: Clean, reusable, and testable permission logic.
class IsPremiumSubscriber(BasePermission):
    def has_object_permission(self, request, view, obj):
        return request.user.subscription.is_premium

class MyViewSet(ModelViewSet):
    permission_classes = [IsAuthenticated, IsPremiumSubscriber]

# BAD: Logic is trapped inside the view, hard to reuse or test.
class MyViewSet(ModelViewSet):
    def check_object_permissions(self, request, obj):
        super().check_object_permissions(request, obj)
        if not request.user.subscription.is_premium:
            self.permission_denied(request)
```

Rule: Extract Small, Reusable Logic into View Helpers. Repetitive logic within views, such as complex request parsing that happens on multiple views, should be extracted into dedicated helper functions close to the API definition. If it becomes large enough, it may belong in its own service.

Rule: Use Custom Exceptions Throughout the Stack. For domain-specific errors, create custom exception classes that inherit from `APIException`. These should be raised in all layers—data, logic, and API—to provide a specific, meaningful error language for the entire application.

Rule: Balance Development Speed with Error Detail. We don't mind if an API sometimes returns a generic error if that means the code is easier to read and maintain.

- **Team-Focused Approach:** Our backend projects are usually made by teams that also work on the frontend, so we prioritize fast iteration over perfect external API error reporting.
- **Never Expose Internal Exceptions:** We never want to report raw exceptions or implementation details to users.
- **Always Log to Sentry:** Whenever an unexpected exception is caught, it must be logged with `logger.exception()` so it reaches Sentry for debugging.
- **Generic Errors Are Acceptable:** Sometimes it's easier to have a `try` at the start and a `catch` at the end to turn any error into a 400, rather than have lots of tiny try-catches that make code harder to read.

> ***Rationale:*** *For close-knit teams, development velocity is more valuable than perfect error messages. When there's a problem, team members can communicate directly. The important thing is that all unexpected errors are logged to Sentry for proper debugging.*

Rule: Use Existing Mixins, Build New Ones, Avoid Overriding DRF If you find yourself tempted to override `update()`, `perform_create()` or other methods, pause for a moment. This is very often a bad idea, unless in the rare situations where it is generic enough to be extracted into a reusable mixin. You will notice that most of our mixins are structured that way.

> ***Rationale:*** *Overriding `perform_create()` and other methods goes directly against our declarative, model-first approach, and it makes code more diffcult to reason about, placing business logic in the wrong place. The model is the one who knows what to do when it is created — not the API. So any business logic regarding creating an object should go inside that object (even if it just forwards to/calls a service). This way, creating a user in the admin is the same as in the API, and business logic can always be found in services and models.*

6. Background Job Processing & Periodic Task Management

Background tasks (i.e. periodic/scheduled) are critical components and must be designed with reliability in mind.

Rule: Use RQSC for Periodic Background Tasks. For periodic background tasks, we use RQSC (Redis Queue Scheduler's Cousin), our own library that wraps `django-rq` and provides an admin interface for scheduling tasks.

- **Task Registration:** Tasks must be registered with the `@RQSCTask` decorator.
- **Admin Interface:** RQSC provides an admin interface where periodic tasks can be scheduled and managed.
- **Cron-Based Scheduling:** Each RQSC task entry in the Django database leads to a periodic task being scheduled according to a cron schedule.
- **Management Command Creation:** Use `RQSCScheduledTask.GetOrCreateForTask()` in management commands to set up periodic tasks.

```
# GOOD: RQSC task registration and setup
from rqsc.decorators import RQSCTask
from rqsc.models import RQSCScheduledTask

@RQSCTask("rq_cleanup_failed_job_registry", "Cleans up the failed job registry")
def task_cleanup_failed_job_registry():
    from alts_betanalyser_backend.rq_utils import cleanup_failed_job_registry
    cleanup_failed_job_registry()

# In a management command:
class Command(BaseCommand):
    help = "Creates background tasks for the project."

    def handle(self, *args, **options):
        RQSCScheduledTask.GetOrCreateForTask(
            name="Clean up failed job registry",
            cron_expression="0 1 * * *",  # Daily at 1:00 AM
```

```
            task=task_cleanup_failed_job_registry
        )
```

Rule: Design Tasks for Failure and State Tracking. Our `RQ`-based tasks are not fire-and-forget. They are a core part of our model's lifecycle.

- **Use the `@job` Decorator:** All background tasks that modify model state should be decorated for RQ.
- **Handle Failure Gracefully:** Every task must have a `try...except` block to catch failures, log the exception, and update the associated model's status to a `FAILED` state.
- **Update Model Status:** The task is responsible for transitioning the model's status (e.g., `PENDING` -> `PROCESSING` -> `COMPLETED` / `FAILED`).
- **Store Job IDs:** The job ID returned by the async call should be stored on the model instance that initiated it, enabling monitoring and cancellation.
- **Consider a service:** Very often, a background job can simply be a wrapper around a service. If the task grows large, it's time to turn it into one.

```python
# GOOD: Handles failure, updates status, and is self-contained.
from django_rq import job
import logging

logger = logging.getLogger(__name__)

@job
def process_data(model_id: int):
    model = Model.objects.get(id=model_id)
    try:
        model.status = Model.Status.PROCESSING
        model.save(update_fields=["status"])

        # ... perform the actual work ...

        model.status = Model.Status.COMPLETED
        model.save(update_fields=["status"])
    except Exception as e:
        logger.exception(f"Task for model {model_id} failed: {e}")
        model.status = Model.Status.FAILED
        model.save(update_fields=["status"])
```

Rule: Management Commands Are Thin Interfaces. Management commands should be abstract and contain minimal logic. They are just another interface to the system, not the system itself.

- **Simple Wrappers:** Management commands should be simple wrappers over services or fat model methods.
- **No Complex Business Logic:** It is very rare that a management command should contain complex logic.
- **Delegate to Services:** Usually, management commands will delegate to services (or model methods) because they tend to orchestrate multiple operations.

> ***Rationale:*** *Management commands are an entry point, like API views or admin actions. The actual business logic should live in models or services where it can*

> *be tested and reused.*

7. Testing Philosophy: Confidence from End-to-End

Our tests are designed to provide maximum confidence that the system works as a whole.

Rule: All Tests Inherit from a Common Base Class. All test classes must inherit from our project's `BaseTest` class, which is the home for shared utility and helper methods. These helper methods tend to allow for complex testing (tests within tests) and reusability.

Rule: Test from the Outside In; Mock Only at External Boundaries. Nearly all tests should be written against the API, using the API client. We do not write isolated unit tests for internal model methods.

- **Mocking:** We only mock the "last hop" to a truly external service (e.g., the `requests.post` call in our `LLMService`). We never mock internal components like our own models or services.
- **Test Helpers:** We build a library of reusable test helper functions (e.g., `create_user_via_api(client, ...)`). These helpers contain their own assertions and act as small, self-contained tests that are composed into larger test cases.
- **Test Helpers as Mini-Tests:** Helper methods are tests in themselves. They usually have `assert_ok` or `assert_status_code` parameters to toggle their internal assertions, making them flexible for different test scenarios.
- **Advanced Techniques:** We embrace patterns that improve test coverage and clarity, such as:
  - **Parameterized testing** (e.g., using the `parameterized` library) to run the same test with multiple data scenarios.
  - **Decorator factories for mocking** complex external services in a clean, reusable way.

```
# GOOD: Test helper with assert_ok pattern
def logout(self, assert_ok=True):
    """
    Logout from the current account and clear the current token.
    Returns the logout response.
    """
    # > Act
    response = self.client.post(
        self.LOGOUT_URL,
        headers=self.headers
    )

    # > Assert
    if assert_ok:
        if response.status_code != 204:
            raise AssertionError(f"Logout failed with {response.status_code}:
{response.content}")

    # Clear current token after logout attempt
    if assert_ok or response.status_code == 204:
        self.current_token = None
```

```
    return response

# GOOD: Test helper with assert_status_code pattern
def register_account(self, email, password, assert_ok=True, assert_status_code=None):
    """
    Register an account with the given email and password.
    Returns the registration response.
    """
    # > Arrange
    data = {'email': email, 'password': password}

    # > Act
    response = self.client.post(
        self.REGISTER_URL,
        data=data,
        content_type='application/json',
        headers=self.headers
    )

    # > Assert
    if assert_status_code is not None:
        self.assertEqual(response.status_code, assert_status_code)
    elif assert_ok:
        self.assertEqual(response.status_code, 201)

    return response

# GOOD: Parameterized testing for comprehensive validation
from parameterized import parameterized

@parameterized.expand([
    ('invalid.email', 'Invalid email format'),
    ('', 'Empty email'),
    ('user@', 'Missing domain'),
    ('@domain.com', 'Missing user part'),
    ('user..name@domain.com', 'Double dots'),
    ('user name@domain.com', 'Space in email'),
])
def test_emails_must_be_valid(self, email, description):
    """
    Test that invalid email formats are rejected.
    """
    # Test implementation here
```

> ***Rationale:*** *Testing through the API ensures that our serializers, views, services, and models all work together correctly. This provides far greater confidence than isolated unit tests. It proves the entire stack works. Calling* `User.objects.create()` *in a test, even with our fat models approach, still bypasses all the crucial layers we need to validate: serializers, permissions, view logic, and the* `save()` *method.*

Rule: Structure Tests with the Arrange-Act-Assert Pattern. All tests must be structured with comments to clearly delineate the three phases of a test.

> ***Rationale:*** *This makes tests highly readable, self-documenting, and easy to debug. It enforces a clean separation between setup, execution, and verification.*

```python
def test_user_can_reset_password(self):
    # > Arrange
    # The setup phase: create necessary objects and prepare the state.
    user = self.create_user_via_api(email="test@example.com")
    url = f"/api/users/{user.id}/actions/reset-password/"

    # > Act
    # The action phase: execute the single action being tested.
    response = self.client.post(url)

    # > Assert
    # The verification phase: check the results.
    self.assertEqual(response.status_code, 200)
    user.refresh_from_db()
    self.assertIsNotNone(user.password_reset_token)
```

Sometimes, especially with complex tests using our reusable methods, we do `> Act & Assert`, perhaps even multiple times in the same test.

8. The Admin: A First-Class Citizen

The Django Admin is not an afterthought; it is a core part of the application and our primary tool for support and debugging.

Combining ou fat models approach with a rich admin is incredibly powerful, because editing in the admin means fully exercising all the layers of the system. This has been the secret to our success in many projects: fat models, and rich admin interfaces.

Rule: A Task is Not Done Until It's Reflected in the Admin. Every new model, field, or state must be properly represented in the `admin.py`.

- All fields should be visible in `list_display` or `readonly_fields`.
- Filters (`list_filter`) should be added for important fields.
- Nested objects should use `TabularInline` for a rich, editable interface.

> ***Rationale:*** *A powerful and comprehensive admin allows the entire team to understand the system's state, debug issues, and manage data without ever needing direct database access.*

Rule: Never use pgAdmin or similar tools. If you feel the need to use pgAdmin or a related tool, you almost certainly should enhance the admin to satisfy your needs.

> ***Rationale:*** *Relying on pgAdmin means handling production will be more difficult, and cooperation trickier. It also means that the models don't fully "own" themselves, as we have to go to their "data layer" to work with them.*

9. Utility Maximization & Code Reuse

Rule: Maximize the Use of Utilities. We hate repetition. There should be an attempt at maximizing the use of utilities whenever writing code. On top of that, we want to take something from every project that we can reuse in other projects. This exponentially decreases our development times.

- **Browse Available Utils:** Always browse the `utils` folder and the `cron-django-apps` project to see what utilities are available before writing new code.
- **Centralize Common Patterns:** Whenever we find ourselves writing similar code multiple times, it should be abstracted into a utility function.
- **Use Descriptive Utility Names:** Utility functions should have clear, descriptive names that explain their purpose.

```
# GOOD: Using the nested_get utility for safe dictionary access
from utils.dict_utils import nested_get

# Safe access with dot syntax - returns None if any key is missing
user_email = nested_get(api_response, 'user.profile.email')
address_city = nested_get(data, 'address.city.name')

# BAD: Manual nested access with multiple checks
try:
    user_email = api_response['user']['profile']['email']
except (KeyError, TypeError):
    user_email = None
```

> ***Rationale:*** *Utility maximization eliminates code duplication, reduces bugs, and makes the codebase more maintainable. When we fix a bug in a utility function, it's fixed everywhere it's used.*

10. Code Style & Quality

Clean, readable code is a cornerstone of our philosophy.

Rule: Use Guard Clauses. Exit early to reduce nesting and improve readability.

```
# GOOD
def process_request(request):
    if not request.user.is_active:
        return  # Exit early
    # ... proceed with logic for active users
```

Rule: Embrace Typing, Enums, and Derived Types.

- Use type hints everywhere.
- Use `models.TextChoices` or `Enum` for any field with a fixed set of choices.
- **Create derived types to give meaning to primitive types.** Instead of passing `dict[str, Any]` everywhere, create a named alias. This makes code self-documenting and prevents accidental misuse of similarly shaped dictionaries.

```
# GOOD: The type name explains the intent, even if the structure is simple.
from typing import TypeAlias, Any

APIResponse: TypeAlias = dict[str, Any]
```

```
UserData: TypeAlias = dict[str, Any]

def process_api_response(response: APIResponse): ...
def create_user_from_data(data: UserData): ...
```

- Use **Pydantic** for validating complex, untrusted dictionary structures (like configuration files or complex API inputs) and for creating clear, derived types for internal data transfer. Always configure Pydantic models with `extra="forbid"` to prevent unexpected data.

```
# GOOD: Pydantic defines a clear and strict data contract.
from pydantic import BaseModel, ConfigDict

class StepConfig(BaseModel):
    model_config = ConfigDict(extra="forbid")
    name: str
    params: dict[str, Any]
```

> ***Rationale:*** *These practices make our code easier to read, reason about, and refactor with confidence. They turn potential runtime errors into static analysis errors.*