P1C4 - Tirez parti de l'ORM de Django

Vérifiez votre travail

Le service. Il ne filtre plus en Python, et l'approbation forme un tout :

```
class LeaveService:
    def __init__(
        self,
        policies: LeavePolicyResolver,
        notifier: LeaveNotifier,
        today: Callable[[], date] = date.today,
    ) -> None:
        self._policies = policies
        self._notifier = notifier
        self._today = today

    def get_balance(self, employee_id: int) -> float | None:
        employee = Employee.objects.filter(pk=employee_id).first()
        if employee is None:
            return None
        return self._compute_balance(employee)

    @transaction.atomic
    def approve(self, request_id: int) -> ApprovalResult:
        leave_request = (
            LeaveRequest.objects.select_related("employee").filter(pk=request_id).first()
        )
        if leave_request is None:
            return ApprovalResult.NOT_FOUND
        employee = leave_request.employee

        if self._compute_balance(employee) < leave_request.working_days:
            leave_request.status = "Refused"
            leave_request.save()
            return ApprovalResult.INSUFFICIENT_BALANCE

        leave_request.status = "Approved"
        leave_request.save()
        employee.days_taken += leave_request.working_days
        employee.save()

        self._notifier.notify(employee, "Votre demande de congés est approuvée.")
        return ApprovalResult.APPROVED

    def _compute_balance(self, employee: Employee) -> float:
        contract_type = ContractType(employee.contract_type)
        policy = self._policies.policy_for(contract_type)
        return policy.compute_entitlement(employee, self._today()) - employee.days_taken
```

Le constructeur n'a pas changé depuis le chapitre 3 : c'est le signe que les dépendances étaient déjà bien placées. Seul l'accès aux données a évolué, et il reste dans le service, comme Hamac l'a choisi.

Les vues. Elles ne connaissent plus que le service et ses résultats :

```
from typing import assert_never

from django.http import JsonResponse
```

```
from .composition import build_leave_service
from .services import ApprovalResult


def balance(request):
    # Contrat de l'API inchangé jusqu'au chapitre 5.
    employee_id = request.GET.get("employeeId", "")
    result = None
    if employee_id.isdigit():
        result = build_leave_service().get_balance(int(employee_id))
    if result is None:
        return JsonResponse({"error": "Employee not found"})
    return JsonResponse({"employeeId": int(employee_id), "balance": result})
```

La vue `approve` change de la même façon : elle appelle `build_leave_service()`, et son `match` ne bouge pas.

La racine de composition est celle présentée plus haut. Pour ajouter les stages, c'est le seul module d'assemblage qui recevra une ligne de plus.

Les tests. Le test de la politique des CDD fonctionne toujours tel quel, puisque le taux par défaut n'a pas changé. Ajoutez-lui le cas du taux refusé :

```
def test_fixed_term_policy_rejects_non_positive_rate():
    with pytest.raises(ValueError):
        FixedTermContractPolicy(days_per_full_month=-1)
```

Le test de la notification utilise le faux notificateur et la base de test :

```
import pytest

from leave.composition import build_policy_resolver
from leave.models import Employee
from leave.services import ApprovalResult, LeaveService


class FakeNotifier:
    def __init__(self) -> None:
        self.sent: list[tuple[Employee, str]] = []

    def notify(self, employee: Employee, message: str) -> None:
        self.sent.append((employee, message))


@pytest.mark.django_db
def test_approval_notifies_employee(demo_data):
    notifier = FakeNotifier()
    service = LeaveService(build_policy_resolver(), notifier)

    result = service.approve(1)

    assert result is ApprovalResult.APPROVED
    assert [message for _, message in notifier.sent] == [
        "Votre demande de congés est approuvée."
    ]
```

Ce test a besoin de la base, car le service interroge l'ORM : c'est la conséquence assumée du choix fait au chapitre 3. Il réutilise la vraie construction des politiques, avec `build_policy_resolver`, et ne remplace que ce qui doit l'être : le notificateur. Pour vérifier un calcul à une date précise, vous pourriez de la même façon passer `today=lambda: date(2026, 10, 20)`.

La distinction attendue est la suivante : déplacer `_build_leave_service` dans un autre fichier aurait simplement **déplacé l'assemblage** ; en créant une racine de composition, seul module à connaître les

classes concrètes, vous lui avez **donné une place**. Les vues, le service et les politiques ne savent plus rien de la façon dont ils sont reliés, et chaque test choisit les collaborateurs qu'il veut.

Erreurs fréquentes

• **Garder** `objects.all()` **et filtrer avec** `next(…)` **ou une boucle** : la syntaxe change, mais la base renvoie toujours toute la table. Seul ce qui est enchaîné sur le `QuerySet` devient du SQL.

• **Utiliser** `get(pk=…)` **sans gérer** `DoesNotExist` : une demande inexistante redevient une erreur 500. `filter(…).first()` ou un `try/except LeaveRequest.DoesNotExist` traitent ce cas.

• **Importer** `settings` **dans la politique** : le calcul fonctionne, mais votre test unitaire dépend désormais de la configuration de Django.

• **Placer la transaction dans la vue** ou activer `ATOMIC_REQUESTS` : l'approbation n'est plus protégée quand elle est appelée hors de HTTP.

• **Remplacer le notificateur avec** `@patch("leave.views.NotificationService")` : le test fonctionne jusqu'au jour où l'import change de module, et il cache le fait que le service accepte déjà un notificateur en paramètre.

• **Tout faire en un seul commit** : si un test échoue, vous ne savez plus si la cause est l'accès aux données, la composition ou la configuration.