COURS

Écrivez du code Python maintenable avec MVC et SOLID

P1C3 - Organisez votre code Django avec les principes SOLID

Vérifiez votre travail

Votre code correspond maintenant à la **version intermédiaire** du dépôt.

Le test du bug. Il passe par l'API, car c'est le comportement observé par Leïla que vous voulez vérifier :

```
@pytest.mark.django_db
def test_balance_of_fixed_term_employee_counts_only_full_months(client, demo_data):
    response = client.get("/api/leave/balance", {"employeeId": 4})

    assert response.status_code == 200
    assert response.json()["balance"] == pytest.approx(4.56, abs=0.01)
```

Sur le starter, il échoue avec une valeur d'environ 6,64. `pytest.approx` tolère l'imprécision des nombres à virgule flottante : `7 * 2.08 - 10` donne `4.5600000000000005`.

Les politiques. Les règles des CDI et de l'alternance utilisent des constantes nommées :

```
ANNUAL_DAYS = 25
EXAM_PREPARATION_DAYS = 5


class PermanentContractPolicy:
    def compute_entitlement(self, employee: Employee, today: date) -> float:
        return ANNUAL_DAYS


class ApprenticeshipPolicy:
    def compute_entitlement(self, employee: Employee, today: date) -> float:
        return ANNUAL_DAYS + EXAM_PREPARATION_DAYS
```

Ces deux politiques reçoivent la date du jour sans l'utiliser : c'est le prix du contrat commun. Ruff, que vous lancerez au chapitre 6, peut signaler ces paramètres inutilisés selon sa configuration : c'est un cas où l'on désactive l'avertissement pour ces lignes, en expliquant pourquoi.

Le service. L'affichage et l'approbation utilisent la même méthode `_compute_balance` : la divergence est devenue impossible.

```
class ApprovalResult(Enum):
    APPROVED = auto()
    INSUFFICIENT_BALANCE = auto()
    NOT_FOUND = auto()


class LeaveNotifier(Protocol):
    def notify(self, employee: Employee, message: str) -> None: ...


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)

    def approve(self, request_id: int) -> ApprovalResult:
        # Filtrage en Python conservé : il sera confié à la base au chapitre 4.
        matching = [r for r in LeaveRequest.objects.all() if r.id == request_id]
        if not matching:
            return ApprovalResult.NOT_FOUND
        leave_request = matching[0]
        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
```

`approve` renvoie un `ApprovalResult`, et `get_balance` renvoie `None` pour un salarié introuvable : le service décrit ce qui s'est passé dans les termes du métier, et c'est à la vue de le traduire. La variable s'appelle `leave_request`, car dans Django, `request` désigne toujours la requête HTTP.

Les vues. Elles ne contiennent plus aucune règle RH :

```
def _build_leave_service() -> LeaveService:
    # Assemblage provisoire : il sera déplacé au chapitre 4.
    policies = LeavePolicyResolver({
        ContractType.PERMANENT: PermanentContractPolicy(),
        ContractType.FIXED_TERM: FixedTermContractPolicy(),
        ContractType.APPRENTICESHIP: ApprenticeshipPolicy(),
    })
    return LeaveService(policies, NotificationService())


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})


def approve(request):
    request_id = request.GET.get("id", "")
```

```
    if not request_id.isdigit():
        return JsonResponse({"error": "Invalid id"})

    match _build_leave_service().approve(int(request_id)):
        case ApprovalResult.APPROVED:
            return JsonResponse({"status": "Approved"})
        case ApprovalResult.INSUFFICIENT_BALANCE:
            return JsonResponse({"status": "Refused", "message": "Insufficient balance"})
        case ApprovalResult.NOT_FOUND:
            return JsonResponse({"error": "Leave request not found"})
        case unreachable:
            assert_never(unreachable)
```

Si un quatrième résultat apparaît dans `ApprovalResult`, `assert_never` signalera la branche manquante. Deux cas qui faisaient planter le starter avec une erreur 500, une demande introuvable et un identifiant de salarié non numérique, renvoient désormais un message : ce sont les seuls autres changements de comportement.

Le test unitaire. Dans `tests/test_policies.py` :

```
from datetime import date

import pytest

from leave.models import Employee
from leave.policies import FixedTermContractPolicy


def test_fixed_term_policy_counts_only_full_months():
    policy = FixedTermContractPolicy()
    employee = Employee(hire_date=date(2026, 3, 15))

    entitlement = policy.compute_entitlement(employee, date(2026, 10, 20))

    assert entitlement == pytest.approx(14.56)
```

Ce test ne porte pas le marqueur `django_db` : créer un `Employee` sans appeler `save()` ne déclenche aucune requête. Il s'exécute en quelques millisecondes, et donnera le même résultat dans un an.

La distinction attendue est la suivante : un code **déplacé** se retrouve dans un autre module, avec les mêmes défauts ; un code **réorganisé** a changé de structure, de sorte que la règle n'existe plus qu'à un seul endroit et peut être testée seule.

Erreurs fréquentes

• **Créer les collaborateurs dans le service** : vous avez déplacé les dépendances cachées au lieu de les rendre visibles.

• **Appeler** `date.today()` **dans une politique**, ou **passer** `date.today()` **au lieu de** `date.today` au service : votre test passera aujourd'hui, puis échouera dans quelques semaines.

• **Ranger le calcul dans** `Employee` **avec le même** `if/elif` : l'ajout des stages oblige toujours à modifier une règle existante.

• **Modifier les routes ou corriger le filtrage en même temps** : vous mélangez plusieurs changements.

• **Renvoyer** `-1` **ou lever une exception dans une politique** pour signifier « pas de droits » : le service ne peut plus faire confiance au contrat.