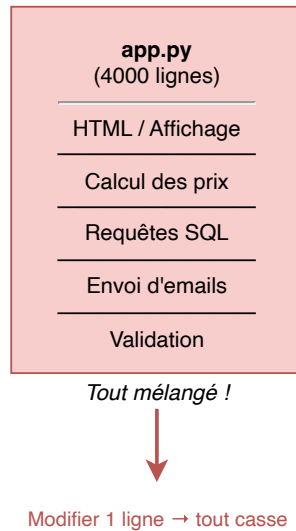


# LAYERING : Architecture en couches

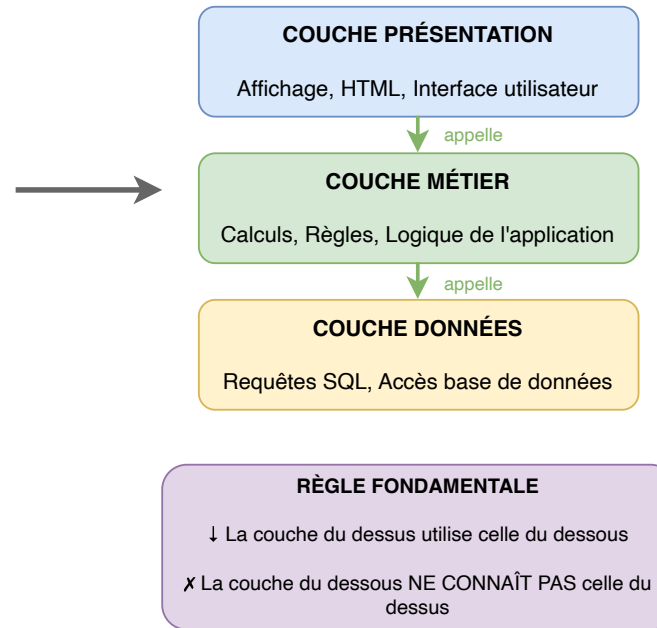
## Sans layering



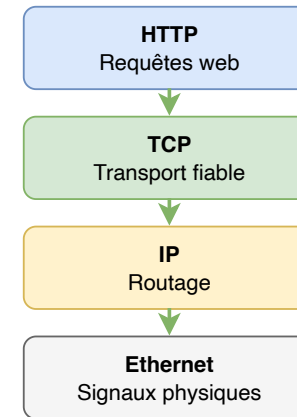
## Avantages du layering

- ✓ Travail en équipe facilité
- ✓ Remplacement d'une couche sans tout casser
- ✓ Réduction de l'effet domino
- ✓ Briques réutilisables

## Avec layering



## Exemple : Protocoles réseau



## Limites

- △ Changements en cascade  
(ajouter 1 champ = toucher toutes les couches)
- △ Coût des transformations de données  
(objet → SQL → objet → JSON...)

# LES 3 COUCHES PRINCIPALES



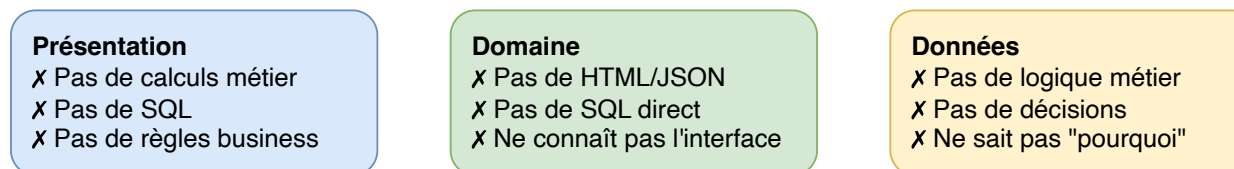
★ COUCHE LA PLUS IMPORTANTE

## FLUX D'APPEL (sens unique !)

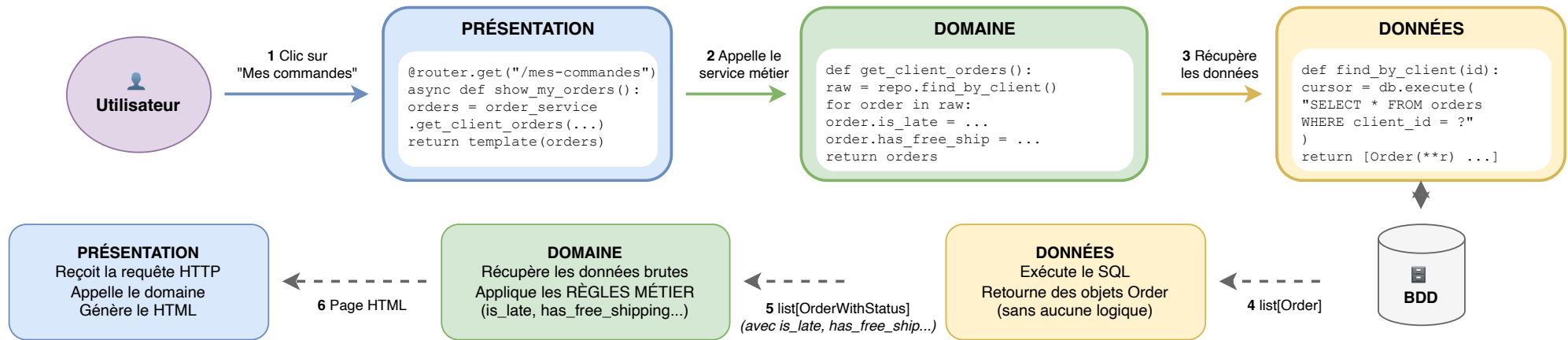


⚠ **JAMAIS dans l'autre sens !**  
Le domaine et les données ne doivent JAMAIS appeler la présentation.

## CE QUE CHAQUE COUCHE NE FAIT PAS



## FLUX D'UNE REQUÊTE : "Voir mes commandes"



# LE PIÈGE : Logique métier dans la présentation

**Scénario** : "Les produits avec ventes en hausse de +10% doivent s'afficher en rouge"

## ❌ MAUVAIS : Logique dans le template

presentation/templates/products.html

```
{% for product in products %}

{% if (product.sales_this_month
- product.sales_last_month)
/ product.sales_last_month
> 0.10 %}

<tr class="highlight-red">
{% endif %}
{% endfor %}
```

### Problèmes :

- Calcul métier caché dans le HTML
- Seuil de 10% impossible à retrouver
- Duplication si app mobile
- Tests impossibles

## ✅ BIEN : Logique dans le domaine

domain/products.py

```
class ProductService:
    THRESHOLD = 0.10 # 10%

    def _has_improving_sales(self, p):
        growth = (p.this_month - p.last_month)
        / p.last_month
        return growth > self.THRESHOLD
```

retourne  
has improving sales

presentation/templates/products.html

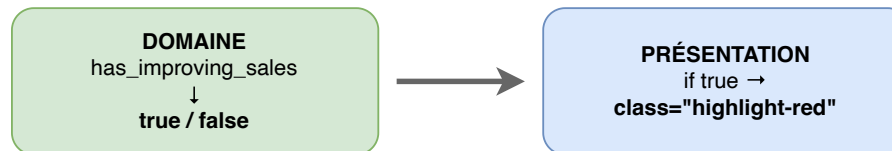
```
{% for product in products %}

{% if product.has_improving_sales %}
<tr class="highlight-red">
{% endif %}
```

### Avantages :

- Seuil centralisé et modifiable
- Réutilisable (web, mobile, API)
- Testable unitairement
- Le template ne fait qu'afficher

## SÉPARATION DÉCISION / AFFICHAGE



### 🔪 TEST POUR DÉTECTER LA FUITE :

"Si j'ajoute une interface CLI, est-ce que je dois **dupliquer** de la logique ?"

Si oui → du code métier s'est glissé dans la présentation !

# STRUCTURE DE PROJET : Architecture 3 couches

## Arborescence

```
mon_app/
├── main.py
├── presentation/
│   ├── routes.py
│   ├── middleware.py
│   ├── templates/
│   └── orders.html
├── domain/
│   ├── entities/
│   │   ├── order.py
│   │   └── promotion.py
│   └── services/
│       └── order_service.py
├── data/
│   ├── database.py
│   ├── repositories/
│   └── order_repo.py
├── pyproject.toml
└── uv.lock
```

### presentation/

- Routes HTTP (endpoints)
- Templates HTML
- Middleware (auth, logs...)
- Sériailisation JSON

### domain/

- **entities/** : objets métier (Order, Client, Promotion...)
- **services/** : logique métier (règles, calculs, validations)

### data/

- **database.py** : connexion BDD
- **repositories/** : accès données (lecture/écriture SQL)

### ⚠ OÙ VIVENT LES ENTITÉS ?

Les entités (Order, Promotion...) sont dans le **domaine**, pas dans data.

Une commande est un **concept métier**.  
Le repository ne fait que lire/écrire ces objets.

## SENS DES IMPORTS

### presentation/

↓ from domain import...

### domain/

↓ from data import...

### data/

### ❌ INTERDIT

# dans data/  
from presentation import...

# dans domain/  
from presentation import...

**Note** : data/ peut importer les **entités** du domaine (pour construire les objets Order, etc.) mais jamais les services.