ANGULAR JAVA SIMULATION ENTRETIEN

BY IDEO-LAB

March 2026

V 1.0

AGENDA DES QUESTIONS/REPONSES

- 1 C’est quoi une transaction en base de données ?
- 2. Peux-tu expliquer les propriétés ACID ??
- 3. À quoi sert un index et quand est-ce qu’on en utilise ?
- 4. Différence entre lazy loading et eager loading ?
- 5. C’est quoi le problème N+1 et comment le résoudre ?”
- 6. Comment éviter le N+1 ?
- 7. C’est quoi une LazyInitializationException ?
- 8. À quoi sert un EntityGraph ?
- 9. Peux-tu expliquer : @OneToMany, @ManyToOne, @OneToOne ?
- 10. Différence entre une interface et une classe abstraite ?
- 11. Pourquoi préférer la composition à l’héritage ?
- 12. Peux-tu expliquer le try-with-resources ?
- 13. Différence entre InputStream et OutputStream ?
- 14. Pourquoi utilise-t-on async/await en JavaScript ?

1. C’est quoi une transaction en base de données ?

Une **transaction en base de données**, c'est un **ensemble d'opérations exécutées comme un tout cohérent**.

L'idée clé est simple : soit **tout se passe correctement**, soit **rien n'est conservé**.

Par exemple, si une application bancaire doit :

1. retirer 100 € du compte A,
2. ajouter 100 € au compte B,

ces deux actions doivent être liées.
On ne veut surtout pas que le retrait soit enregistré si le virement vers le compte B échoue ensuite. C'est précisément le rôle de la transaction.

Définition claire

Une transaction permet de garantir l'**intégrité des données** lorsqu'on effectue plusieurs lectures, insertions, mises à jour ou suppressions en base.

En pratique, elle encadre souvent des opérations entre :

- `BEGIN` / `START TRANSACTION`
- puis `COMMIT` si tout est correct
- ou `ROLLBACK` s'il y a une erreur

Exemple concret

Imaginons une commande e-commerce :

- création de la commande
- décrément du stock
- enregistrement du paiement
- création d'une ligne de facture

Ces opérations doivent rester cohérentes.
Si le paiement échoue, on ne veut pas garder une commande "fantôme" ni décrémenter le stock pour rien.

Donc :

- si tout réussit → `COMMIT`
- si une étape échoue → `ROLLBACK`

Pourquoi c'est indispensable

Sans transaction, on risque :

- des données incohérentes
- des mises à jour partielles
- des erreurs difficiles à corriger
- des problèmes de concurrence quand plusieurs utilisateurs modifient les mêmes données en même temps

Les 4 propriétés ACID

Quand on parle de transaction, on parle presque toujours des propriétés **ACID**.

1. Atomicité

Une transaction est **indivisible**.
Soit tout est validé, soit tout est annulé.

Exemple : sur un virement bancaire, on ne valide pas seulement le débit sans le crédit.

2. Cohérence

La transaction doit faire passer la base d'un **état valide** à un **autre état valide**.

Exemple : si une règle métier interdit un stock négatif, la transaction ne doit jamais laisser la base dans un état où cette règle est violée.

3. Isolation

Deux transactions exécutées en parallèle ne doivent pas se perturber de manière incorrecte.

3. Isolation

Deux transactions exécutées en parallèle ne doivent pas se perturber de manière incorrecte.

Exemple : deux utilisateurs essaient d'acheter le dernier produit en stock. L'isolation sert à éviter les lectures sales, les écrasements de données, ou certains effets de concurrence.

4. Durabilité

Une fois le `COMMIT` effectué, les données sont considérées comme **définitivement enregistrées**, même en cas de crash ou redémarrage du serveur.

Exemple de réponse “entretien”

Tu peux répondre comme ça :

> Une transaction en base de données est un groupe d’opérations exécutées comme une unité logique. Elle garantit que toutes les opérations réussissent ensemble, ou qu’aucune n’est appliquée en cas d’erreur. Cela permet d’assurer l’intégrité et la cohérence des données, surtout lorsqu’un traitement métier touche plusieurs tables. Une transaction repose généralement sur les propriétés ACID : atomicité, cohérence, isolation et durabilité.

Lien avec Java

En Java, surtout avec **Spring**, les transactions sont souvent gérées avec `@Transactional`.

Exemple :

Java

```java
@Transactional
public void transfer(Account from, Account to, BigDecimal amount) {
    from.debit(amount);
    to.credit(amount);
    accountRepository.save(from);
    accountRepository.save(to);
}
```

Ici, si une erreur survient après le débit mais avant la fin du traitement, Spring peut annuler toute la transaction avec un rollback.

Point important à dire en entretien

Une bonne réponse ne consiste pas seulement à dire “ça annule en cas d’erreur”.
Il faut aussi montrer que tu comprends que la transaction sert à :

- **sécuriser un traitement métier**
- **éviter les incohérences**
- **gérer proprement les accès concurrents**
- **fiabiliser les applications critiques**

Formulation plus “senior”

> Une transaction est un mécanisme de contrôle d’intégrité qui encapsule plusieurs opérations SQL dans une unité de travail atomique. Elle garantit qu’un traitement métier complexe ne laisse jamais la base dans un état intermédiaire ou incohérent, même en cas d’erreur, d’accès concurrents ou de panne système.

Si tu veux, je peux maintenant te préparer la **réponse parfaite d’entretien**, plus orale, plus naturelle, en mode **“ce que je dirais face au recruteur”**.

2. Peux-tu expliquer les propriétés ACID ??

🔹 Définition rapide

Les propriétés **ACID** définissent les garanties fondamentales d'une transaction en base de données :

- **A** → Atomicité
- **C** → Cohérence
- **I** → Isolation
- **D** → Durabilité

Elles assurent que les données restent **fiables, même en cas d'erreur, de concurrence ou de crash.**

🔸 1. Atomicité (Atomicity)

👉 Tout ou rien

Une transaction est **indivisible** :

- soit toutes les opérations réussissent → `COMMIT`
- soit une échoue → `ROLLBACK` → tout est annulé

💡 Exemple

Virement bancaire :

- débit du compte A
- crédit du compte B

Si le crédit échoue → on annule aussi le débit.

👉 Sinon → argent perdu = catastrophe.

🔸 2. Cohérence (Consistency)

👉 La base passe d'un **état valide à un autre état valide**

Toutes les règles métier et contraintes doivent être respectées :

- clés étrangères
- contraintes UNIQUE
- règles métier (ex: stock ≥ 0)

💡 Exemple

Si une transaction crée une commande :

- le client doit exister
- le produit doit exister
- le stock ne doit pas devenir négatif

👉 Si une règle est violée → la transaction échoue.

🔸 3. Isolation (Isolation)

👉 Les transactions concurrentes ne doivent pas se perturber

Quand plusieurs transactions s'exécutent en même temps :

- chacune doit "se comporter comme si elle était seule"

💡 Problèmes évités

- **Dirty read** : lire une donnée non validée
- **Non-repeatable read** : lire deux valeurs différentes
- **Phantom read** : apparition de lignes entre deux lectures

💡 Exemple

Deux utilisateurs achètent le dernier produit :

- sans isolation → stock peut devenir -1
- avec isolation → une seule transaction passe

👉 Géré via niveaux d'isolation (READ COMMITTED, REPEATABLE READ, SERIALIZABLE...)

🔶 4. Durabilité (Durability)

👉 Une fois validé → c'est permanent

Après un `COMMIT` :

- les données sont sauvegardées de façon durable
- même en cas de crash serveur

💡 Exemple

Un paiement validé :

- ne doit jamais disparaître après un redémarrage

👉 Assuré via logs, journaux de transactions, disque, réplication…

🔹 Résumé simple (à dire vite en entretien)

> Les propriétés ACID garantissent la fiabilité des transactions :
>
> - "Atomicité : tout ou rien"
> - "Cohérence : respect des règles métier"
> - "Isolation : pas d'interférences entre transactions"
> - "Durabilité : les données sont persistées après validation"

🔥 Version "senior" (très appréciée)

> Les propriétés ACID définissent le cadre de fiabilité des transactions en base. Elles permettent de garantir qu'un traitement métier reste cohérent, même en cas de concurrence, d'erreurs applicatives ou de panne système. C'est essentiel dans les systèmes critiques comme la finance, l'e-commerce ou les systèmes distribués.

⚠️ Petit bonus (qui fait la différence)

Tu peux ajouter :

👉 “En pratique, on fait souvent des compromis sur l’isolation pour des raisons de performance, surtout dans les systèmes à forte charge.”

Ça montre que tu comprends **le monde réel**, pas juste la théorie.

3. À quoi sert un index et quand est-ce qu’on en utilise ?

🔹 Définition simple

Un **index** en base de données est une **structure de données** (souvent un **B-Tree**) qui permet de **retrouver rapidement des lignes sans parcourir toute la table**.

👉 C'est l'équivalent d'un **index dans un livre** :

- sans index → tu lis tout
- avec index → tu vas directement à la page

🔸 À quoi sert un index ?

👉 Accélérer les requêtes SQL, principalement :

✔ 1. Recherches (`WHERE`)

```sql
SELECT * FROM users WHERE email = 'test@mail.com';
```

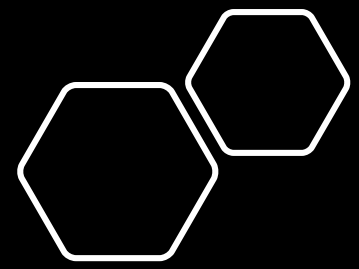
Sans index → scan complet (FULL SCAN)

Avec index → accès direct

✔ 2. Jointures (`JOIN`)

```sql
SELECT * FROM orders o
JOIN users u ON o.user_id = u.id;
```

👉 Index sur `user_id` → jointure beaucoup plus rapide



✔ 3. Tri (`ORDER BY`)

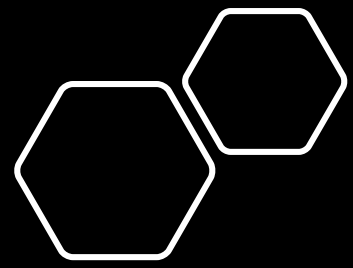
```sql
SELECT * FROM products ORDER BY price;
```

👉 Un index peut éviter un tri coûteux

✔ 4. Groupement (`GROUP BY`)

```sql
SELECT COUNT(*) FROM orders GROUP BY status;
```

👉 Accélère les agrations



✔️ 5. Contraintes (UNIQUE, PK)

👉 Les clés primaires et uniques utilisent automatiquement des index

---

🔸 Quand utiliser un index ?

👉 On met un index quand une colonne est :

✔️ Souvent utilisée dans `WHERE`

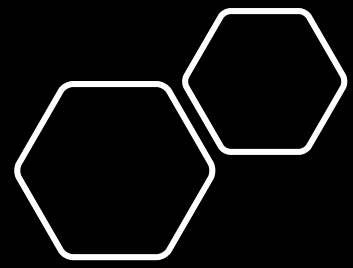
- email
- user_id
- status

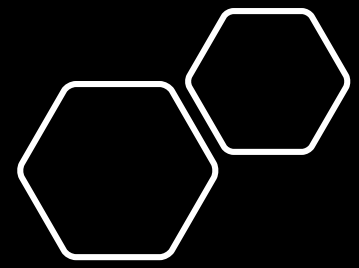
✔️ Utilisée dans des `JOIN`

👉 Très important en Java/Spring avec ORM

✔️ Utilisée pour trier ou grouper

- dates
- prix
- catégories





✔ Avec forte cardinalité

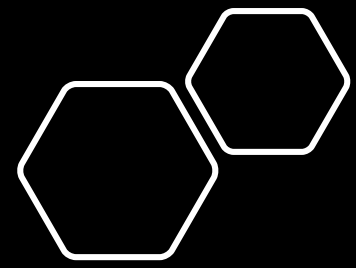
👉 Beaucoup de valeurs différentes

Exemple :

- email → bon index ✅
- boolean (true/false) → mauvais index ❌

🔴 Quand NE PAS en utiliser ?

👉 C'est ça qui fait la différence en entretien.



❌ Trop d'index

- ralentit les `INSERT`, `UPDATE`, `DELETE`
- car chaque index doit être mis à jour

❌ Colonnes peu sélectives

- genre `status = 'ACTIVE'` si 95% des lignes ont cette valeur

  👉 l'index ne sert à rien

❌ Tables très petites

👉 scan complet = plus rapide qu'un index

🔸 Exemple concret

Table `users` :

id	email	status

👉 Bon index :

```sql
CREATE INDEX idx_users_email ON users(email);
```

👉 Mauvais index :

```sql
CREATE INDEX idx_users_status ON users(status);
```

(si peu de variations)

Types d'index (bonus qui fait la différence)

- B-Tree → le plus courant
- Hash → recherche exacte
- Full-text → recherche texte
- Composite index → plusieurs colonnes

Exemple :

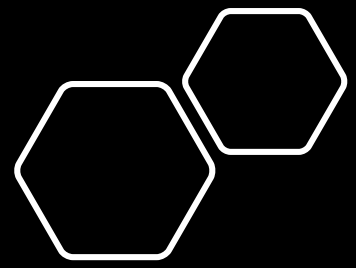
```sql
CREATE INDEX idx_user_status_date ON users(status, created_at);
```

🔥 Réponse parfaite entretien

> Un index est une structure de données qui permet d'accélérer les requêtes en évitant les scans complets de table. On l'utilise principalement sur les colonnes filtrées dans les WHERE, utilisées dans les JOIN ou dans les ORDER BY. Cependant, il faut les utiliser avec parcimonie, car ils ont un coût en écriture et en stockage. Un bon index est généralement placé sur une colonne fortement sélective et fréquemment utilisée dans les requêtes critiques.

⚡ Version "senior" (très impactante)

> Un index est un compromis performance lecture / coût écriture. Il permet d'optimiser les accès aux données mais dégrade les performances des opérations de modification. Le bon design d'index dépend donc des patterns d'accès réels de l'application.



🎯 Petit tips pour ton entretien Angular + Java

Tu peux ajouter :

👉 “Dans un backend Java avec JPA/Hibernate, un mauvais index peut complètement plomber les performances d’une API REST.”

👉 Et là tu passes direct en profil expérimenté.

🔷 1. Différence entre index clusterisé et non clusterisé

🔸 Index clusterisé (Clustered Index)

👉 Les données de la table sont physiquement triées selon l'index

- L'index définit l'ordre de stockage réel des lignes
- Une table ne peut avoir qu'un seul index clusterisé
- Souvent lié à la clé primaire

💡 Exemple

Table triée par `id` :

id	name
1	A
2	B
3	C

👉 Les données sont stockées dans cet ordre sur disque

🔸 Index non clusterisé (Non-Clustered Index)

👉 Structure séparée de la table

- Contient :
  - la valeur indexée
  - un pointeur vers la ligne réelle
- Ne modifie pas l'ordre physique des données
- On peut en avoir plusieurs

💡 Exemple

Index sur `email` :

```
email → pointeur vers ligne
```

👉 La table reste stockée autrement (ex: par id)

🔥 Résumé clair

Type	Stockage	Nombre	Usage
Clusterisé	Trie physiquement la table	1	Accès principal
Non clusterisé	Structure séparée	Plusieurs	Recherches rapides

4. Différence entre lazy loading et eager loading ?

Définition simple

- Eager loading → on charge tout immédiatement
- Lazy loading → on charge uniquement quand nécessaire

🔶 1. Eager Loading (chargement immédiat)

👉 Les données liées sont chargées **en même temps que l'objet principal**

💡 Exemple (Java / JPA)

```java
@Entity
public class Order {

    @ManyToOne(fetch = FetchType.EAGER)
    private User user;
}
```

👉 Quand tu charges une `Order` :

- la `User` est chargée automatiquement

✅ Avantages

- Simple à utiliser
- Pas de surprise (tout est déjà là)
- Moins de requêtes supplémentaires

❌ Inconvénients

- Peut charger **beaucoup trop de données inutilement**
- Risque de **gros ralentissements**
- Peut créer des requêtes très lourdes (JOIN multiples)

⚠️ Cas réel

👉 Tu récupères une liste de commandes →
👉 chaque commande charge son user →
👉 explosion mémoire + réseau

🔶 2. Lazy Loading (chargement à la demande)

👉 Les données sont chargées **uniquement quand on en a besoin**

💡 Exemple (Java / JPA)

```java
@Entity
public class Order {

    @ManyToOne(fetch = FetchType.LAZY)
    private User user;
}
```

👉 Quand tu charges une `Order` :

- `User` n'est PAS chargé immédiatement
- il sera chargé uniquement si tu fais :

```java
order.getUser()
```

✅ Avantages

- Optimise les performances
- Réduit le volume de données
- Idéal pour gros systèmes

❌ Inconvénients

- Peut générer **N+1 queries problem**
- Peut causer des erreurs (LazyInitializationException)
- Plus complexe à maîtriser

🔥 Problème classique : N+1

👉 Exemple :

```java
List<Order> orders = orderRepository.findAll();

for (Order o : orders) {
    o.getUser().getName();
}
```

👉 Résultat :

- 1 requête pour orders
- - N requêtes pour users

👉 catastrophe performance

🔷 Angular (frontend)

👉 Même logique, mais appliquée au chargement de modules/composants

---

🔷 Angular (frontend)

👉 Même logique, mais appliquée au chargement de modules/composants

---

🔸 Lazy Loading Angular

👉 Charger les modules **uniquement quand on navigue dessus**

TypeScript

```
{
  path: 'admin',
  loadChildren: () => import('./admin/admin.module')
    .then(m => m.AdminModule)
}
```

👉 Le module `admin` n'est chargé que si l'utilisateur y accède

✅ Avantages

- Réduit le bundle initial
- Améliore le temps de chargement
- Idéal pour grosses apps

❌ Inconvénients

- Latence lors de la première navigation
- Complexité routing

🔸 Eager Loading Angular

👉 Tous les modules sont chargés au démarrage

```typescript
imports: [AdminModule]
```

🔥 Résumé Angular

Type	Quand chargé
Eager	au démarrage
Lazy	à la navigation

🔥 Réponse parfaite entretien

> Eager loading consiste à charger toutes les données liées immédiatement, tandis que lazy loading charge les données uniquement lorsqu'elles sont nécessaires. En Java avec JPA, cela impacte directement les performances des requêtes et peut entraîner des problèmes comme le N+1. En Angular, lazy loading est utilisé pour charger les modules à la demande afin d'optimiser le temps de chargement initial de l'application.

⚡ Version "senior qui fait la différence"

> Lazy vs eager loading est un compromis entre simplicité et performance. Eager est plus simple mais peut surcharger inutilement le système, tandis que lazy optimise les ressources mais nécessite une bonne maîtrise pour éviter les pièges comme le N+1 ou les erreurs de session Hibernate. En pratique, on privilégie souvent lazy avec des optimisations ciblées comme les fetch join ou les DTO projections.

🎯 Punchline à placer (très fort)

👉

> “Par défaut, je privilégie le lazy loading, mais je contrôle explicitement les chargements dans les requêtes critiques.”

💥 Ça, c’est exactement ce que veut entendre un recruteur.

5. C’est quoi le problème N+1 et comment le résoudre ?”

🔹 Définition simple du problème N+1

👉 Le problème **N+1** se produit quand :

- on fait **1 requête pour récupérer une liste**
- puis **N requêtes supplémentaires** pour charger des données associées

👉 Résultat :

- explosion du nombre de requêtes SQL
- performances catastrophiques

🔸 Exemple concret

💡 Cas classique (JPA / Hibernate)

```java
List<Order> orders = orderRepository.findAll();

for (Order o : orders) {
    System.out.println(o.getUser().getName());
}
```

👉 Ce qui se passe réellement :

1. Requête 1 :

```sql
SELECT * FROM orders;
```

2. Puis pour chaque order (N fois) :

```sql
SELECT * FROM users WHERE id = ?;
```

👉 Total :

- 1 + N requêtes ❌

🔥 Pourquoi c’est un problème ?

- surcharge base de données
- latence réseau
- CPU inutile
- effet amplifié en production

👉 Sur 1000 lignes → 1001 requêtes 😱

🔸 Pourquoi ça arrive ?

👉 À cause du **lazy loading**

- Hibernate ne charge pas les relations immédiatement
- chaque accès déclenche une requête

🔹 Solutions (TRÈS IMPORTANT EN ENTRETIEN)

✅ 1. Fetch Join (LA solution principale)

👉 Charger les données en une seule requête

```java
@Query("SELECT o FROM Order o JOIN FETCH o.user")
List<Order> findAllWithUser();
```

👉 SQL généré :

```sql
SELECT o.*, u.*
FROM orders o
JOIN users u ON o.user_id = u.id;
```

👉 ✅ 1 seule requête

✅ 2. EntityGraph

👉 Alternative propre et déclarative

```java
@EntityGraph(attributePaths = {"user"})
List<Order> findAll();
```

👉 Même effet que fetch join

✅ 3. DTO Projection

👉 Charger uniquement les données nécessaires

```java
@Query("SELECT new com.app.OrderDTO(o.id, u.name) FROM Order o JOIN
List<OrderDTO> findOrders();
```

👉 Très performant (moins de données)

✅ 4. Batch fetching (Hibernate)

👉 Charger plusieurs relations en bloc

```properties
hibernate.default_batch_fetch_size=50
```

👉 Réduit N+1 → N/50

✅ 5. Passer en EAGER (⚠️ avec précaution)

```java
@ManyToOne(fetch = FetchType.EAGER)
```

👉 Peut résoudre N+1... mais :

- risque de surcharge globale
- mauvais choix par défaut

🔥 Résumé clair

Solution	Usage
Fetch Join	⭐ meilleur choix
EntityGraph	propre / flexible
DTO	performance max
Batch	optimisation
EAGER	à éviter par défaut

🔥 Réponse parfaite entretien

> Le problème N+1 se produit lorsqu'une requête principale est suivie de N requêtes supplémentaires pour charger des relations, généralement à cause du lazy loading. Cela entraîne une dégradation importante des performances. Pour le résoudre, on utilise principalement des fetch join, des EntityGraph ou des projections DTO afin de réduire le nombre de requêtes et charger les données de manière optimisée.

⚡ Version "senior / terrain"

> Le N+1 est un problème classique en ORM. La bonne approche n'est pas de désactiver le lazy loading, mais de contrôler explicitement les chargements via des requêtes optimisées, en fonction du cas d'usage. Il faut toujours raisonner en termes de volume de données et analyser les requêtes générées.

Fetch Join

🔹 1. C’est quoi un Fetch Join ?

👉 Un **Fetch Join** est une jointure JPQL qui permet de **charger une entité et ses relations en une seule requête SQL.**

👉 Il sert principalement à **éviter le problème N+1.**

🔸 Exemple

Sans Fetch Join (problème N+1)

```java
List<Order> orders = orderRepository.findAll();

for (Order o : orders) {
    o.getUser().getName();
}
```

👉 1 + N requêtes ❌

Avec Fetch Join

Java

```java
@Query("SELECT o FROM Order o JOIN FETCH o.user")
List<Order> findAllWithUser();
```

👉 SQL généré :

SQL

```sql
SELECT o.*, u.*
FROM orders o
JOIN users u ON o.user_id = u.id;
```

👉 ✅ 1 seule requête

🔥 Ce que ça fait réellement

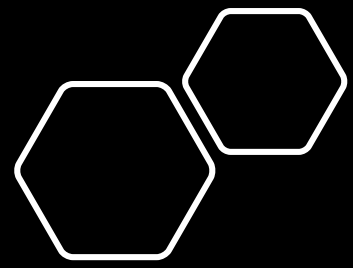
- Hibernate charge `Order`
- ET charge `User` dans la même requête
- ET hydrate les objets directement

👉 Pas de requêtes supplémentaires

🔷 2. Pourquoi c'est important ?

👉 Parce que :

- évite le **N+1**
- améliore drastiquement les performances
- contrôle explicite du chargement



🔷 3. Les limites du Fetch Join (TRÈS IMPORTANT)

👉 C'est là que tu fais la différence en entretien.

❌ 1. Explosion du nombre de lignes (cartesian product)

Si tu fais :

```java
JOIN FETCH o.items
```

👉 et une commande a 10 items :

👉 tu obtiens :

- 1 order → 10 lignes SQL

👉 et Hibernate doit reconstruire les objets

❌ 2. Impossible de paginer correctement

```java
Page<Order> findAll(Pageable pageable);
```

👉 avec fetch join sur collection → ❌

- pagination fausse
- résultats incohérents

👉 car pagination se fait sur lignes SQL, pas sur entités

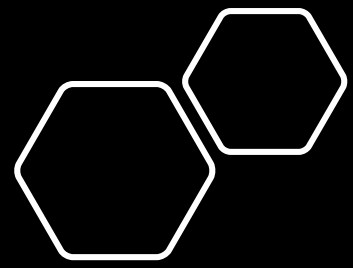
❌ 3. Plusieurs collections = problème

```java
JOIN FETCH o.items
JOIN FETCH o.payments
```

👉 ❌ interdit ou très instable

👉 Hibernate peut lever :

```
cannot simultaneously fetch multiple bags
```



❌ 4. Surcharge mémoire

👉 tu charges potentiellement :

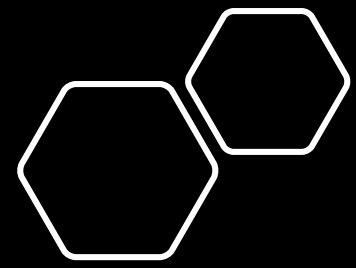
- beaucoup trop de données
- objets lourds

❌ 5. Pas toujours nécessaire

👉 parfois tu charges des données que tu n'utilises pas

🔹 4. Bonnes pratiques

👉 Très important à dire en entretien :



✅ Utiliser Fetch Join :

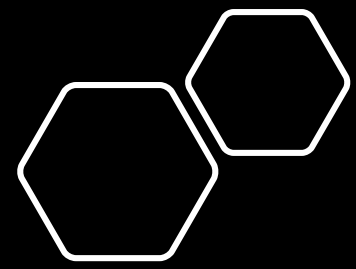
- sur relations simples (`ManyToOne`, `OneToOne`)
- sur petits volumes
- pour éviter N+1 ciblé

❌ Éviter Fetch Join :

- sur grosses collections
- avec pagination
- sur plusieurs relations complexes

✅ Alternatives

- DTO projection → plus performant
- EntityGraph → plus flexible
- Batch fetching → compromis



🔥 Réponse parfaite entretien

> Un fetch join est une jointure JPQL qui permet de charger une entité et ses relations en une seule requête SQL, afin d'éviter le problème N+1. Cependant, il a des limites importantes : il peut provoquer une explosion du nombre de lignes, poser des problèmes de pagination, et ne fonctionne pas bien avec plusieurs collections. Il doit donc être utilisé de manière ciblée en fonction du cas d'usage.

⚡ Version "senior / terrain"

> Le fetch join est un outil puissant pour contrôler le chargement des relations, mais il doit être utilisé avec prudence. Dans les cas complexes ou volumineux, il est souvent préférable d'utiliser des projections DTO ou des stratégies de batch fetching pour garder un bon équilibre entre performance et consommation mémoire.

6. Comment éviter le N+1 ?

Définition rapide

Le problème **N+1** apparaît quand :

- on fait **1 requête principale** pour charger une liste d'objets,
- puis **N requêtes supplémentaires** pour charger une relation associée, souvent à cause du **lazy loading**.

Exemple :

- 1 requête pour charger 100 commandes
- 100 requêtes supplémentaires pour charger les utilisateurs liés

Total : **101 requêtes** au lieu d'1 ou 2.

Comment éviter le N+1 ?

Il n'y a pas une seule méthode.
La bonne réponse en entretien, c'est de montrer que tu sais choisir la stratégie selon le contexte.

---

1. Utiliser `JOIN FETCH`

C'est souvent la **première réponse attendue**.

Le principe :
on demande explicitement à JPA/Hibernate de charger l'entité principale **et** sa relation dans la même requête.

Exemple

Java

```java
@Query("SELECT o FROM Order o JOIN FETCH o.user")
List<Order> findAllWithUser();
```

Pourquoi c'est efficace

Au lieu de :

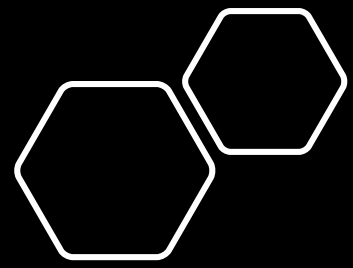
- 1 requête pour `orders`
- puis N requêtes pour `user`

on fait :

- **1 seule requête SQL avec jointure**

Argument entretien

> Le fetch join est la solution la plus directe pour éviter le N+1 quand on sait à l'avance quelles relations doivent être chargées.



2. Utiliser `EntityGraph`

C'est une alternative plus déclarative et souvent plus propre.

Exemple

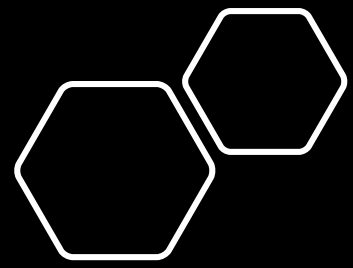
```java
@EntityGraph(attributePaths = {"user"})
List<Order> findAll();
```

Intérêt

- évite le N+1
- évite parfois de multiplier les requêtes custom
- plus flexible que de tout coder à la main dans des `JOIN FETCH`

Argument entretien

> EntityGraph permet de piloter le chargement des relations de manière plus lisible et réutilisable, tout en gardant le bénéfice d'un chargement optimisé.



3. Utiliser des projections DTO

Très bonne réponse en entretien, parce qu'elle montre une vision plus "architecture/performance".

Le principe :
au lieu de charger des entités complètes, on charge uniquement les données nécessaires à l'écran ou à l'API.

Exemple

```java
@Query("""
SELECT new com.app.dto.OrderDTO(o.id, o.reference, u.name)
FROM Order o
JOIN o.user u
""")
List<OrderDTO> findOrderSummaries();
```

Pourquoi c'est souvent meilleur

- moins de données chargées
- pas de lazy loading imprévu
- pas d'hydratation complète d'objets inutiles
- très bien pour les APIs REST

Argument entretien

> Pour des écrans de consultation ou des APIs, la projection DTO est souvent la meilleure approche, car elle évite à la fois le N+1 et le surchargement mémoire.

4. Configurer le batch fetching Hibernate

Quand on ne peut pas tout résoudre facilement par `fetch join`, Hibernate peut regrouper les chargements.

Exemple de config

properties

```
hibernate.default_batch_fetch_size=50
```

Ce que ça change

Au lieu de faire :

- 100 requêtes pour 100 relations

Hibernate peut faire :

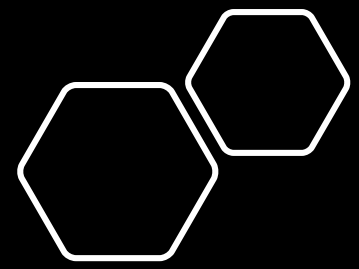
- 2 ou 3 requêtes groupées

Important

Ce n'est pas aussi propre qu'un vrai fetch optimisé, mais c'est une bonne amélioration.

Argument entretien

> Le batch fetching ne supprime pas totalement le principe du chargement différé, mais il réduit fortement l'impact du N+1 en regroupant les accès.



5. Bien choisir entre `LAZY` et `EAGER`

Attention : ici, il faut répondre intelligemment.

Beaucoup de candidats disent :

> "Pour éviter le N+1, il suffit de mettre EAGER."

C'est une **mauvaise réponse** si elle est donnée sans nuance.

Pourquoi ?

`EAGER` peut :

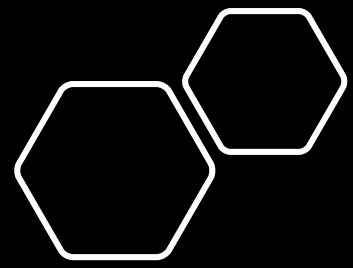
- charger trop de données
- déclencher d'autres surcoûts
- rendre le comportement global difficile à maîtriser

Bonne position

- garder en général `LAZY`
- **contrôler explicitement** les chargements nécessaires via requêtes optimisées

Argument entretien

> Je préfère en général laisser les relations en lazy loading, puis charger explicitement ce dont j'ai besoin avec fetch join, EntityGraph ou DTO selon le cas d'usage.



6. Analyser les requêtes SQL réellement générées

C'est un point très important, souvent oublié.

Pour éviter le N+1, il faut déjà le **voir**.

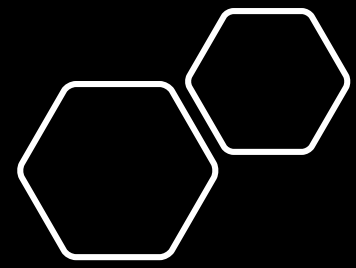
En pratique

- activer les logs SQL Hibernate
- utiliser les statistiques Hibernate
- observer le nombre de requêtes en test ou en préproduction
- profiler les endpoints lents

Argument entretien

> Le N+1 ne se détecte pas uniquement dans le code ; il faut analyser les requêtes réellement émises par l'ORM.

Ça montre une approche terrain.



Ce qu’il ne faut pas dire seul

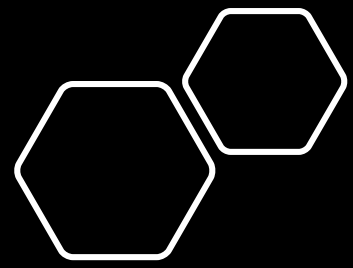
Éviter une réponse du type :

> “On met EAGER et c’est bon.”

C’est trop simpliste et souvent faux en production.

Le recruteur attend plutôt :

- fetch join
- EntityGraph
- DTO
- batch fetching
- analyse des requêtes



Réponse d'entretien prête à dire

> Pour éviter le problème N+1, il faut contrôler explicitement le chargement des relations au lieu de laisser l'ORM déclencher des requêtes au fil de l'eau. La solution la plus classique est le `JOIN FETCH`, qui permet de charger l'entité principale et ses associations dans une seule requête. On peut aussi utiliser `EntityGraph`, qui est plus déclaratif, ou des projections DTO quand on veut charger uniquement les données utiles, ce qui est souvent le meilleur choix pour une API REST. En complément, Hibernate propose le batch fetching pour réduire le nombre de requêtes. En pratique, je privilégie généralement le lazy loading par défaut, puis j'optimise les cas critiques de manière explicite en analysant les requêtes SQL générées.

Version plus courte et plus percutante

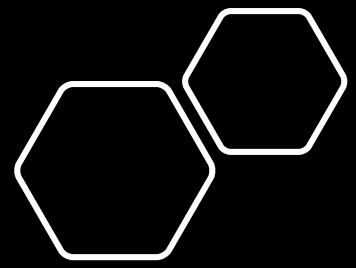
> Pour éviter le N+1, il ne faut pas compter sur le chargement implicite des relations. Il faut piloter le fetch avec des `JOIN FETCH`, des `EntityGraph` ou des projections DTO. Le lazy loading reste souvent le bon choix par défaut, mais il doit être maîtrisé. Et surtout, il faut vérifier les requêtes SQL réellement générées par Hibernate.

Petit bonus très apprécié

Tu peux ajouter cette phrase :

> Le vrai sujet n'est pas seulement d'éviter le N+1, mais de charger exactement ce qui est utile, ni plus, ni moins.

7. C’est quoi une LazyInitializationException ?



🔹 Définition simple

👉 Une **LazyInitializationException** est une erreur Hibernate qui se produit quand :

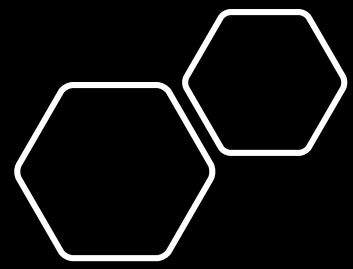
> On essaie d'accéder à une relation **lazy** alors que la session Hibernate est déjà fermée.

🔸 Exemple concret

```java
Order order = orderRepository.findById(1L).get();

// plus tard…
order.getUser().getName();  // 💥 LazyInitializationException
```

👉 Pourquoi ?

- `user` est en `LAZY`
- Hibernate n'a PAS chargé `user`
- la session (transaction) est fermée
- Hibernate ne peut plus aller en base → exception



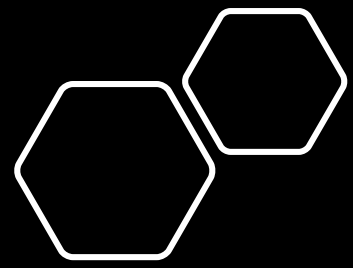
🔥 Message typique

```
failed to lazily initialize a collection of role: ...
could not initialize proxy - no Session
```

🔹 Pourquoi ça arrive ?

👉 Parce que Hibernate fonctionne avec une **session (persistence context)**.

- Tant que la session est ouverte → OK
- Dès qu'elle est fermée → plus de chargement lazy possible



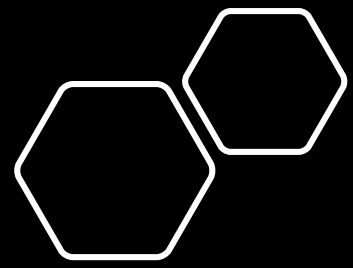
💡 Cas classique en Spring

```java
@Service
public Order getOrder() {
    return orderRepository.findById(1L).get();
}
```

Puis dans un controller :

```java
order.getUser().getName();  // 💥 BOOM
```

👉 La transaction est déjà terminée



🔹 Les causes principales

- Lazy loading + accès hors transaction
- Retour d'entités vers la couche web
- Mauvaise gestion des services / transactions

🔹 Comment éviter cette exception ?

👉 Très important en entretien : **ne pas donner une seule solution**

✅ 1. Utiliser `JOIN FETCH`

```java
@Query("SELECT o FROM Order o JOIN FETCH o.user WHERE o.id = :id")
Order findWithUser(Long id);
```

👉 La relation est déjà chargée → pas d'erreur

✅ 2. Utiliser `EntityGraph`

```java
@EntityGraph(attributePaths = {"user"})
Order findById(Long id);
```

✅ 3. Utiliser des DTO

👉 Meilleure pratique pour API REST

```java
SELECT new OrderDTO(o.id, u.name)
FROM Order o JOIN o.user u
```

👉 Pas de lazy → pas de problème

✅ 4. Accéder aux données dans la transaction

```java
@Transactional
public OrderDTO getOrder() {
    Order o = repo.findById(1L).get();
    return new OrderDTO(o.getUser().getName());
}
```

👉 Tout est chargé pendant que la session est ouverte

❌ 5. Open Session In View (OSIV)

👉 Technique où la session reste ouverte jusqu'au controller

⚠️ Problèmes :

- masque les bugs
- requêtes cachées
- performance imprévisible

👉 À éviter en architecture propre

❌ 6. Passer en EAGER

👉 Mauvaise solution globale :

```java
@ManyToOne(fetch = FetchType.EAGER)
```

⚠️ Peut créer :

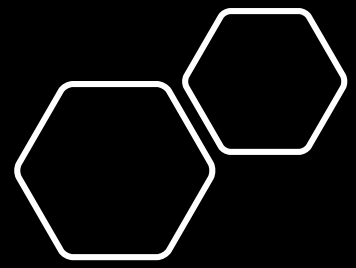
- surcharge
- N+1 ailleurs

🔥 Réponse parfaite entretien

> Une LazyInitializationException se produit lorsqu'on tente d'accéder à une relation lazy alors que la session Hibernate est fermée. Comme Hibernate ne peut plus charger les données depuis la base, il lève une exception. Pour éviter ce problème, il faut charger explicitement les relations dans la requête via un fetch join ou un EntityGraph, ou bien utiliser des DTO pour récupérer uniquement les données nécessaires. En pratique, il est important de maîtriser le cycle de vie des transactions plutôt que de compter sur des solutions comme l'Open Session In View.

⚡ Version "senior"

> La LazyInitializationException est généralement un symptôme d'un mauvais découplage entre les couches. Elle indique qu'on expose des entités en dehors de leur contexte de persistance. La bonne approche consiste à contrôler explicitement le chargement des données au niveau du service, souvent via des projections ou des requêtes optimisées.



8. À quoi sert un EntityGraph ?

🔹 Définition simple

👉 Un **EntityGraph** permet de **définir quelles relations doivent être chargées avec une entité, sans modifier le mapping** (`LAZY` / `EAGER`).

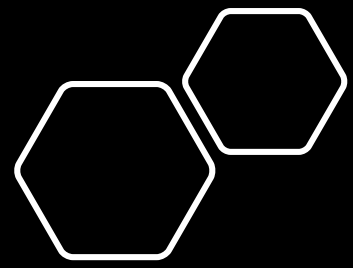
👉 En clair :

> tu gardes `LAZY` par défaut, mais tu choisis **dynamiquement** ce que tu veux charger.

🔸 Pourquoi c'est utile ?

👉 Pour :

- éviter le **N+1**
- éviter les `JOIN FETCH` partout
- garder un code **propre et réutilisable**



🔸 Exemple concret

💡 Entité

```java
@Entity
public class Order {

    @ManyToOne(fetch = FetchType.LAZY)
    private User user;
}
```

👉 Par défaut → `user` n'est PAS chargé

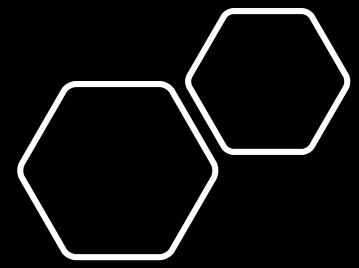
💡 Avec EntityGraph

```java
@EntityGraph(attributePaths = {"user"})
List<Order> findAll();
```

👉 Résultat :

- `Order` + `User` chargés ensemble
- en **1 seule requête**

👉 sans changer le mapping



🔥 Ce que ça change réellement

👉 Sans EntityGraph :

- Lazy → N+1 possible

👉 Avec EntityGraph :

- chargement contrôlé
- requête optimisée
- comportement explicite

🔹 Avantages

✅ 1. Contrôle fin du chargement

👉 tu choisis quoi charger selon le cas d'usage

✅ 2. Réutilisable

👉 pas besoin d'écrire plein de requêtes JPQL

✅ 3. Plus propre que `JOIN FETCH`

👉 séparation :

- mapping (entité)
- logique de chargement (requête)

✅ 4. Compatible avec pagination

👉 contrairement à certains `fetch join`

🔹 Limites

👉 Important à dire en entretien

❌ 1. Moins explicite que JPQL

👉 parfois moins lisible

❌ 2. Moins flexible que DTO

👉 tu charges encore des entités complètes

❌ 3. Peut charger trop de données

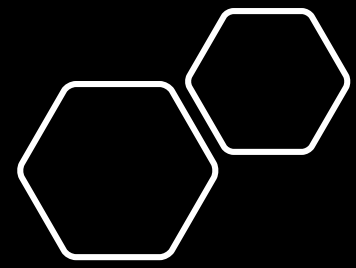
👉 si mal utilisé

❌ 4. Dépend du provider (Hibernate)

👉 comportement parfois subtil

🔥 Réponse parfaite entretien

> Un EntityGraph permet de définir dynamiquement quelles relations d'une entité doivent être chargées lors d'une requête, sans modifier la stratégie de fetch définie dans les annotations. Il est utilisé pour éviter le problème N+1 tout en gardant un code plus propre et réutilisable que des requêtes avec fetch join. Il offre un bon compromis entre lisibilité, performance et flexibilité.



⚡ Version "senior"

> L'EntityGraph permet de découpler le modèle de données de la stratégie de chargement. On peut ainsi garder un lazy loading par défaut et adapter le fetch en fonction du contexte métier. C'est particulièrement utile pour éviter le N+1 sans multiplier les requêtes custom.

9. Peux-tu expliquer les annotations @OneToMany, @ManyToOne, @OneToOne ?

🔹 Vue globale

Ces annotations servent à **modéliser les relations entre tables** :

Annotation	Signification
`@OneToMany`	1 → N
`@ManyToOne`	N → 1
`@OneToOne`	1 → 1

👉 Elles représentent directement les **relations en base de données.**

🔶 1. @ManyToOne (la plus importante)

👉 Plusieurs entités pointent vers une seule

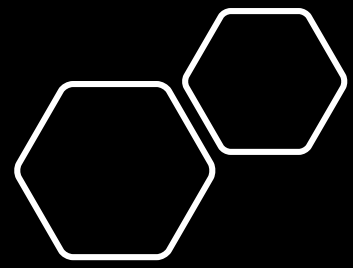
💡 Exemple

Plusieurs commandes → un seul utilisateur

```java
@Entity
public class Order {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id")
    private User user;
}
```

👉 En base :

```
orders.user_id → users.id
```



🔥 Points clés

- côté **N** (la table qui contient la clé étrangère)
- très fréquent
- généralement **LAZY recommandé**

🎯 À dire en entretien

> @ManyToOne est la relation la plus courante, car elle correspond directement à une clé étrangère en base.

🔶 2. @OneToMany

👉 Une entité possède plusieurs autres

💡 Exemple

Un utilisateur → plusieurs commandes

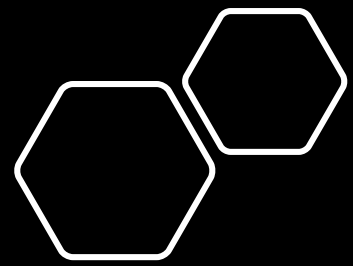
```java
@Entity
public class User {

    @OneToMany(mappedBy = "user")
    private List<Order> orders;
}
```

👉 Ici :

- `mappedBy = "user"` → relation inverse
- la vraie FK est dans `Order`

🔥 Points clés

- côté **1**
- souvent **inverse**
- ne contient PAS la clé étrangère



⚠️ Important

👉 Sans `mappedBy` → table de jointure ❌

🎯 À dire en entretien

> @OneToMany est généralement le côté inverse d'une relation, et doit être utilisé avec mappedBy pour éviter une table intermédiaire inutile.

🔶 3. @OneToOne

👉 Une entité correspond à une seule autre

💡 Exemple

Utilisateur → profil

```java
@Entity
public class User {

    @OneToOne
    @JoinColumn(name = "profile_id")
    private Profile profile;
}
```

🔥 Points clés

- relation 1 ↔ 1
- clé étrangère unique
- parfois coûteux (jointures inutiles)

🔷 Notion clé : côté propriétaire

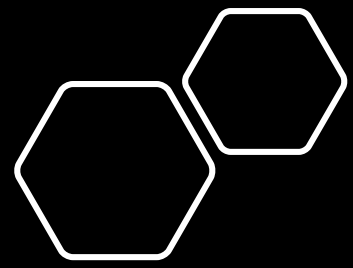
👉 Très important en entretien

- **Owning side** → contient la clé étrangère (`@JoinColumn`)
- **Inverse side** → utilise `mappedBy`

Exemple

Java

```java
// Owning side
@ManyToOne
@JoinColumn(name = "user_id")
private User user;

// Inverse side
@OneToMany(mappedBy = "user")
private List<Order> orders;
```



🔷 FetchType (important)

Relation	Fetch par défaut
ManyToOne	EAGER ⚠️
OneToMany	LAZY ✅

👉 Bonne pratique :

- privilégier **LAZY partout**
- contrôler explicitement le fetch

🔥 Résumé simple

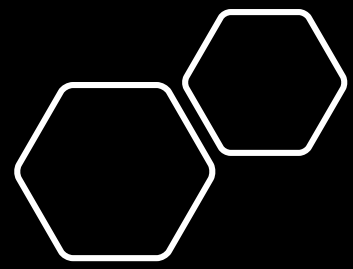
Annotation	Sens	FK
ManyToOne	N → 1	côté courant
OneToMany	1 → N	côté inverse
OneToOne	1 → 1	selon design

🔥 Réponse parfaite entretien

> Les annotations @OneToMany, @ManyToOne et @OneToOne permettent de modéliser les relations entre entités en JPA. @ManyToOne est la relation la plus courante et correspond à une clé étrangère côté base. @OneToMany est généralement le côté inverse de cette relation et doit être utilisé avec mappedBy pour éviter une table intermédiaire. @OneToOne représente une relation unique entre deux entités. Il est important de bien identifier le côté propriétaire de la relation et de gérer le fetch type pour éviter des problèmes de performance.

⚡ Version “senior”

> En pratique, je privilégie des relations simples avec @ManyToOne côté propriétaire et je limite l'usage des @OneToMany pour éviter les chargements coûteux. Je garde le lazy loading par défaut et je contrôle explicitement les fetch dans les requêtes critiques.



10. Différence entre une interface et une classe abstraite ?

Définition simple

La différence principale, c'est que :

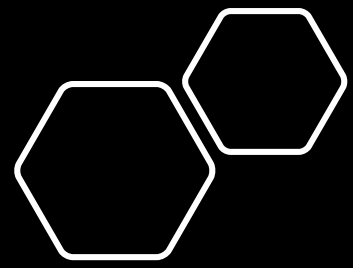
- une **interface** définit un **contrat**
- une **classe abstraite** définit une **base commune partiellement implémentée**

Autrement dit :

- l'interface dit "**voici ce que tu dois savoir faire**"
- la classe abstraite dit "**voici ce que tu dois savoir faire, et je te donne déjà une partie du comportement**"

1. L'interface

Une interface sert à décrire un ensemble de méthodes ou de propriétés qu'une classe devra respecter.



Idée principale

Elle représente un **engagement fonctionnel**.

Exemple Java

Java

```java
public interface PaymentService {
    void pay(double amount);
    void refund(double amount);
}
```

Toute classe qui implémente cette interface devra fournir ces méthodes.

Exemple TypeScript / Angular

TypeScript

```typescript
export interface User {
  id: number;
  name: string;
  email: string;
}
```

Ici, l'interface sert souvent à typer les objets manipulés par les composants, services ou appels HTTP.

À quoi sert une interface ?

- imposer une structure commune
- favoriser le découplage
- permettre plusieurs implémentations
- améliorer la lisibilité et la testabilité

Exemple concret

En Java :

- `StripePaymentService`
- `PaypalPaymentService`

En Angular :

on utilise souvent des interfaces pour typer :

- les DTO
- les réponses d'API
- les modèles de données
- les contrats de services

2. La classe abstraite

Une classe abstraite sert à factoriser du code commun entre plusieurs classes proches.

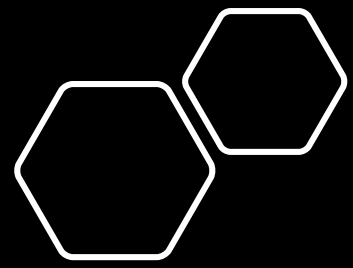
Elle peut contenir :

- des méthodes abstraites
- des méthodes déjà implémentées
- des attributs
- de la logique partagée

Exemple Java

Java

```java
public abstract class AbstractPaymentService {

    protected String currency = "EUR";

    public void logPayment(double amount) {
        System.out.println("Payment log: " + amount);
    }

    public abstract void pay(double amount);
}
```

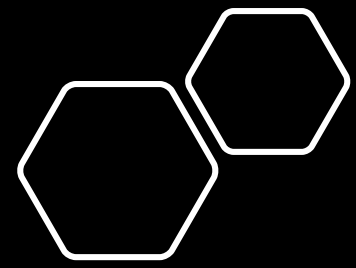


Java

```java
public class StripePaymentService extends AbstractPaymentService {
    @Override
    public void pay(double amount) {
        logPayment(amount);
        System.out.println("Stripe payment: " + amount + " " + curre
    }
}
```

À quoi sert une classe abstraite ?

- mutualiser du code
- imposer une architecture commune
- éviter la duplication
- centraliser des comportements partagés



---

3. Différence fondamentale

Interface

Elle décrit **ce qu'on doit faire.**

Classe abstraite

Elle décrit **ce qu'on doit faire + une partie de la manière de le faire.**

4. En Java : différences importantes

Interface

- pas d'état métier classique partagé comme une vraie base objet
- une classe peut implémenter **plusieurs interfaces**
- très adaptée au contrat, à l'injection, aux architectures propres

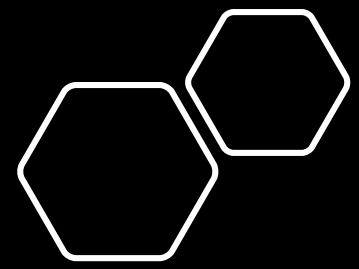
Classe abstraite

- peut contenir des attributs d'instance
- peut contenir du code métier commun
- une classe ne peut hériter que d'**une seule classe abstraite**

👉 C'est un point clé en entretien :

> En Java, on peut implémenter plusieurs interfaces, mais on ne peut hériter que d'une seule classe.

Ça, il faut le dire.



5. En Angular / TypeScript : nuance importante

Dans Angular, les interfaces sont **très utilisées** pour le typage, par exemple :

TypeScript

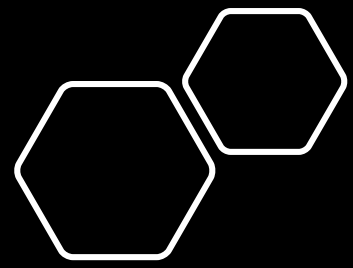
```typescript
export interface Product {
  id: number;
  label: string;
  price: number;
}
```

Elles servent surtout à :

- structurer les données
- typer les retours d'API
- fiabiliser le code

Les classes abstraites, elles, sont souvent utilisées quand on veut :

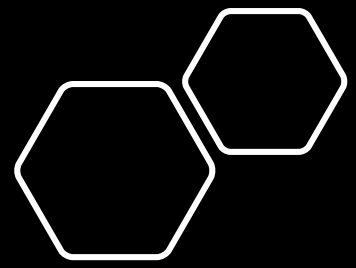
- partager du comportement entre composants ou services
- imposer une base commune
- créer des couches réutilisables



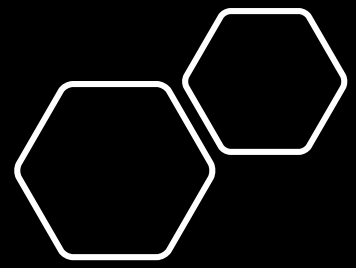
Exemple TypeScript

TypeScript

```typescript
export abstract class BaseComponent {
  loading = false;

  startLoading(): void {
    this.loading = true;
  }

  stopLoading(): void {
    this.loading = false;
  }

  abstract refresh(): void;
}
```



6. Quand utiliser l'un ou l'autre ?



Interface

On l'utilise quand :

- on veut définir un contrat
- on veut plusieurs implémentations possibles
- on veut découpler les couches
- on veut typer proprement les objets Angular/TypeScript

Classe abstraite

On l'utilise quand :

- plusieurs classes partagent déjà un vrai comportement commun
- on veut mutualiser du code
- on veut fournir une base technique ou métier

7. Réponse d’entretien prête à dire

> Une interface définit un contrat : elle dit quelles méthodes ou quelles propriétés une classe doit exposer, sans forcément fournir l’implémentation. Une classe abstraite, elle, sert de base commune : elle peut imposer certaines méthodes abstraites, mais aussi fournir du code déjà implémenté et des attributs partagés. En Java, une classe peut implémenter plusieurs interfaces mais n’hériter que d’une seule classe abstraite. Dans Angular et TypeScript, les interfaces sont très utilisées pour typer les données et les réponses d’API, tandis que les classes abstraites servent davantage à factoriser du comportement commun entre composants ou services.

---

8. Version plus senior

> Je vois l’interface comme un outil de contrat et de découplage, et la classe abstraite comme un outil de factorisation. Quand j’ai besoin de polymorphisme et de souplesse, je privilégie l’interface. Quand plusieurs classes partagent réellement de la logique ou un état commun, une classe abstraite devient pertinente.

9. Réponse courte si le recruteur veut aller vite

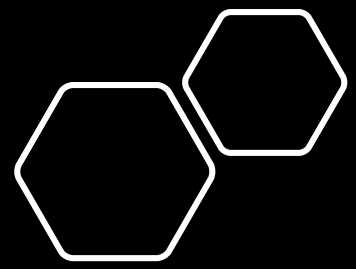
> Une interface définit ce qu'une classe doit faire ; une classe abstraite définit ce qu'elle doit faire et peut déjà lui fournir une partie du comportement. En Java, on peut implémenter plusieurs interfaces mais hériter d'une seule classe abstraite. En Angular, les interfaces servent surtout au typage, alors que les classes abstraites servent à partager du comportement.

---

10. Petit point intelligent à ajouter

Tu peux dire ça, c'est très bien vu :

> Je n'utilise pas une classe abstraite juste pour "faire comme de l'héritage". Je l'utilise seulement s'il existe un vrai comportement commun. Sinon, une interface est souvent plus propre.



11. Pourquoi préférer la composition à l’héritage ?

🔹 Définition simple

👉 Héritage = relation "est un" (*is-a*)

👉 Composition = relation "a un" (*has-a*)

💡 Exemple rapide

- Héritage :

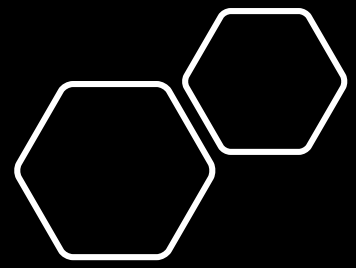
```java
class Dog extends Animal
```

👉 Dog **est un** Animal

- Composition :

```java
class Car {
    private Engine engine;
}
```

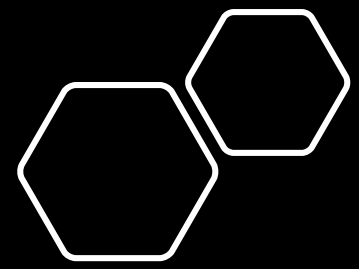
👉 Car **a un** moteur



🔥 Pourquoi préférer la composition ?

👉 Parce qu'elle rend le code :

- plus **flexible**
- plus **maintenable**
- moins **couplé**



🔸 1. Moins de couplage

👉 Avec l'héritage :

- la classe enfant dépend fortement de la classe parent
- tout changement dans le parent peut casser les enfants

👉 Avec la composition :

- les dépendances sont injectées
- plus facile à remplacer / modifier

💡 Exemple

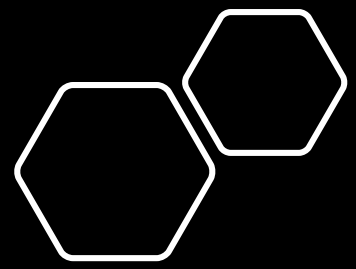
Héritage ❌

```java
class FileLogger extends Logger {
    // fortement dépendant
}
```

Composition ✅

```java
class Logger {
    private OutputStrategy strategy;
}
```

👉 Tu peux changer `strategy` sans casser le reste



🔸 2. Plus flexible (composition dynamique)

👉 Avec la composition, tu peux changer le comportement **à runtime**

```java
logger.setStrategy(new FileStrategy());
logger.setStrategy(new DatabaseStrategy());
```

👉 Impossible avec héritage (statique)

🔸 3. Évite les hiérarchies complexes

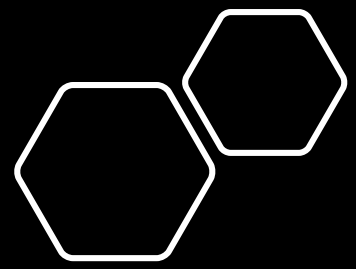
👉 L'héritage mène souvent à :

- classes profondes
- logique difficile à suivre
- rigidité

👉 Exemple classique :

```
Animal
  ├── Dog
      ├── HuntingDog
          ├── PoliceDog
```

👉 Ça devient vite ingérable



🔸 4. Favorise les bonnes pratiques modernes

👉 Composition = base de :

- SOLID (principe de composition over inheritance)
- Dependency Injection (Spring)
- Strategy pattern
- Clean Architecture

🔸 5. Plus testable

👉 Avec composition :

```java
class OrderService {
    private PaymentService paymentService;
}
```

👉 En test :

```java
mock(PaymentService.class)
```

👉 beaucoup plus simple

🔸 6. Évite les effets de bord

👉 En héritage :

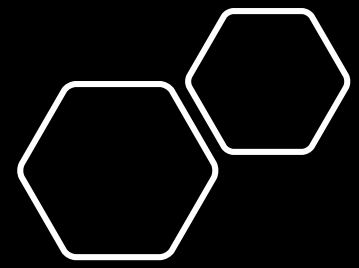
- tu hérites de comportements que tu ne veux pas
- override parfois dangereux

👉 Composition :

- tu contrôles exactement ce que tu utilises

🔥 Résumé clair

Héritage	Composition
rigide	flexible
couplé	découplé
statique	dynamique
difficile à maintenir	facile à faire évoluer

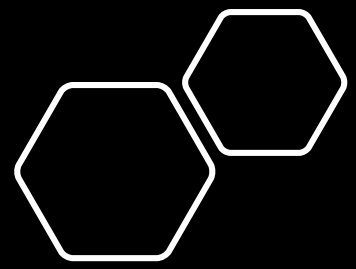


🔥 Réponse parfaite entretien

> On préfère généralement la composition à l'héritage car elle permet de réduire le couplage entre les classes et d'augmenter la flexibilité du code. Avec la composition, les comportements sont injectés et peuvent être modifiés dynamiquement, alors qu'avec l'héritage, les relations sont figées et peuvent rendre le système rigide et difficile à maintenir. C'est pour cela que les bonnes pratiques modernes, comme les principes SOLID ou les architectures Spring, favorisent la composition.

⚡ Version "senior"

> L'héritage est utile pour modéliser une vraie relation métier de type "is-a", mais il devient rapidement problématique dans les systèmes complexes. La composition permet de construire des objets plus modulaires, testables et évolutifs. C'est pour cela que je privilégie la composition, et j'utilise l'héritage uniquement lorsqu'il a un sens métier fort.



🚀 Bonus Angular / Java

👉 Angular :

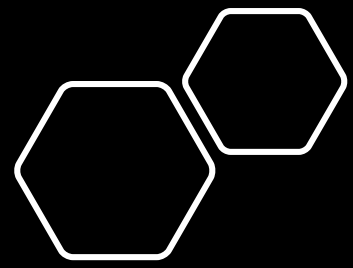
- services injectés → composition
- pas d'héritage massif

👉 Java / Spring :

- injection de dépendances → composition partout

12. Peux-tu expliquer le try-with-resources ?

Définition simple



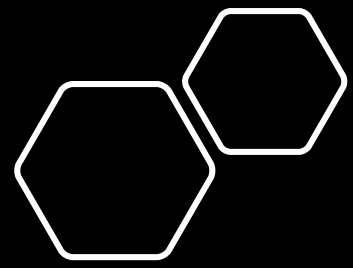
Le **try-with-resources** est une syntaxe introduite en Java qui permet de **fermer automatiquement les ressources** utilisées dans un bloc `try`.

Typiquement, cela concerne des objets comme :

- fichiers
- flux d'entrée/sortie
- connexions JDBC
- statements
- readers / writers

Autrement dit, au lieu de devoir fermer manuellement une ressource dans un `finally`, Java s'en charge automatiquement.

Pourquoi c’est utile ?



Quand on manipule une ressource système, il faut absolument la libérer :

- fermer un fichier
- fermer un stream
- fermer une connexion base de données

Sinon, on risque :

- des fuites mémoire
- des descripteurs de fichiers non libérés
- des connexions JDBC bloquées
- des bugs difficiles à diagnostiquer

Le `try-with-resources` a été créé pour rendre ce code **plus sûr**, **plus lisible** et **moins sujet aux erreurs**.

Syntaxe

```java
try (BufferedReader reader = new BufferedReader(new FileReader("test
    String line = reader.readLine();
    System.out.println(line);
} catch (IOException e) {
    e.printStackTrace();
}
```

Ici, `reader` sera fermé automatiquement à la fin du bloc, même s'il y a une exception.

Ce qu'il faisait avant

Avant `try-with-resources`, on écrivait souvent :

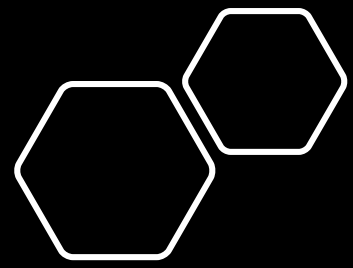
Java

```java
try {
    reader = new BufferedReader(new FileReader("test.txt"));
    String line = reader.readLine();
    System.out.println(line);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

C'est :

- plus verbeux
- plus lourd
- plus facile à mal coder

Le `try-with-resources` simplifie tout ça.



Condition pour l'utiliser

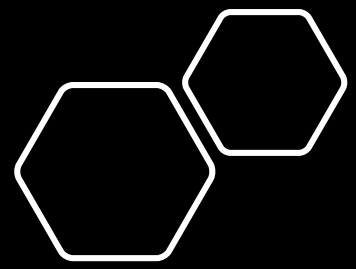
La ressource utilisée doit implémenter l'interface :

```java
AutoCloseable
```

ou une sous-interface comme :

```java
Closeable
```

Cela permet à Java d'appeler automatiquement la méthode `close()`.



Exemple avec JDBC

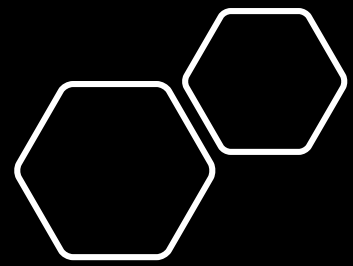
Très bon exemple d'entretien :

```java
try (
    Connection conn = dataSource.getConnection();
    PreparedStatement ps = conn.prepareStatement("SELECT * FROM user
    ResultSet rs = ps.executeQuery()
) {
    while (rs.next()) {
        System.out.println(rs.getString("name"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

Ici, Java fermera automatiquement :

- `ResultSet`
- `PreparedStatement`
- `Connection`

C'est exactement le genre de cas où cette syntaxe est très utile.



Avantages

1. Fermeture automatique

La ressource est libérée même si une exception se produit.

2. Code plus lisible

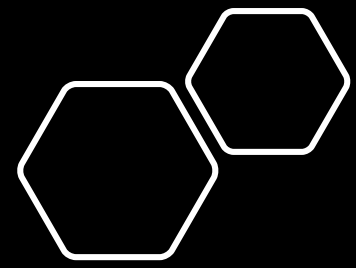
Beaucoup moins de code qu'avec `try / finally`.

3. Moins d'erreurs

On évite les oublis de fermeture.

4. Bonne pratique en production

Très important pour les accès fichiers, flux réseau, JDBC, etc.



Point un peu plus avancé

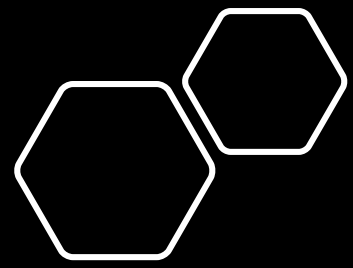
Si une exception est levée dans le bloc `try` et une autre pendant le `close()`, Java ne perd pas la seconde : elle devient une **suppressed exception**.

C'est un bon point à connaître, même si ce n'est pas obligatoire de le dire tout de suite en entretien.

Exemple d'idée :

> try-with-resources gère aussi proprement les exceptions levées au moment de la fermeture des ressources.

Ça fait sérieux.

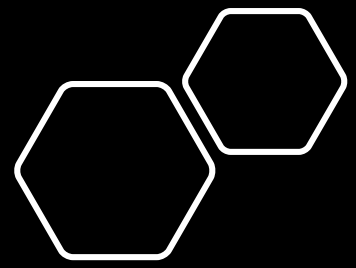


Réponse d'entretien prête à dire

> Le try-with-resources est une syntaxe Java qui permet de déclarer des ressources dans le try afin qu'elles soient fermées automatiquement à la fin du bloc. Il est utilisé pour les objets qui implémentent AutoCloseable, comme les fichiers, les streams ou les connexions JDBC. Cela permet d'éviter les oublis de fermeture, de réduire le code boilerplate par rapport à un finally, et de rendre le code plus sûr et plus lisible.

Version plus senior

> Le try-with-resources est une amélioration importante du modèle de gestion des ressources en Java. Il sécurise la libération des ressources critiques, réduit le risque de fuite et simplifie fortement le code. Je l'utilise systématiquement pour les flux I/O, JDBC et toute ressource AutoCloseable, plutôt que de gérer les fermetures manuellement dans des blocs finally.



Différence à bien dire si on te relance

Le recruteur peut demander :

“Quelle différence avec finally ?”

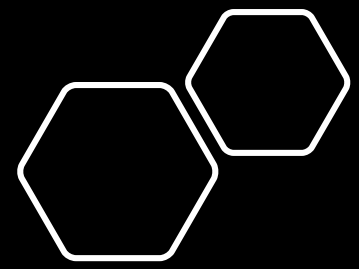
Tu peux répondre :

> Avec finally, on doit fermer manuellement la ressource, ce qui alourdit le code et augmente le risque d’erreur. Avec try-with-resources, la fermeture est automatique et standardisée.

Petit bonus intelligent

Tu peux ajouter :

> En pratique, try-with-resources est devenu la manière standard et propre de gérer les ressources en Java moderne.

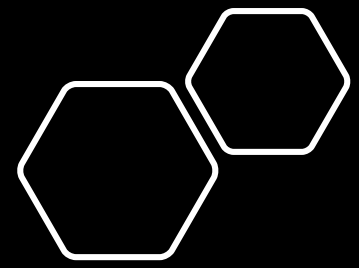


13. Différence entre InputStream et OutputStream ?

🔹 Définition simple

👉 La différence est **le sens du flux de données** :

- **InputStream** → lire des données (**entrée vers ton programme**)
- **OutputStream** → écrire des données (**sortie depuis ton programme**)



🔸 1. InputStream

👉 Sert à **lire des données depuis une source**

💡 Exemples de sources

- fichier
- réseau
- mémoire
- entrée utilisateur

Exemple

```java
InputStream is = new FileInputStream("file.txt");
int data = is.read();
```

👉 Ici :

- le programme **reçoit des données**
- on lit byte par byte

📌 Idée clé

> InputStream = **flux entrant**

🔸 2. OutputStream

👉 Sert à écrire des données vers une destination

💡 Exemples de destinations

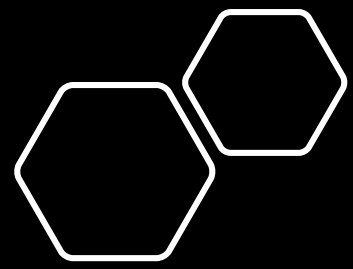
- fichier
- socket réseau
- buffer mémoire

Exemple

```java
OutputStream os = new FileOutputStream("file.txt");
os.write(65); // écrit 'A'
```

👉 Ici :

- le programme **envoie des données**



📌 Idée clé

> OutputStream = **flux sortant**

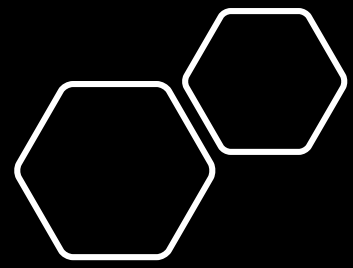
🔥 Schéma mental simple

👉 Pense comme ça :

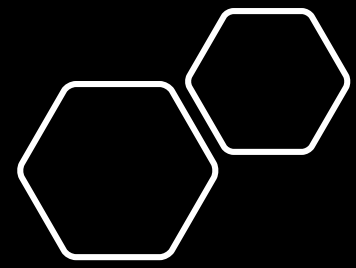
```
[Source] ---> InputStream ---> [Ton programme] ---> OutputStrea
```

Différences principales

InputStream	OutputStream
lecture	écriture
données entrantes	données sortantes
méthode `read()`	méthode `write()`
source → programme	programme → destination



Types concrets (important en entretien)



InputStream

- `FileInputStream`
- `BufferedInputStream`
- `ByteArrayInputStream`

OutputStream

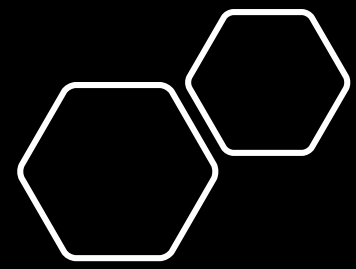
- `FileOutputStream`
- `BufferedOutputStream`
- `ByteArrayOutputStream`

Exemple complet (copie de fichier)

👉 Très bon exemple à donner

Java

```java
try (
    InputStream in = new FileInputStream("input.txt");
    OutputStream out = new FileOutputStream("output.txt")
) {
    byte[] buffer = new byte[1024];
    int length;

    while ((length = in.read(buffer)) != -1) {
        out.write(buffer, 0, length);
    }
}
```



👉 Ici :

- InputStream → lit
- OutputStream → écrit

🔸 Point important (niveau supérieur)

👉 Les streams travaillent en **bytes (binaire)**

👉 Pour du texte → on utilise :

- `Reader` / `Writer`

🔥 Réponse parfaite entretien

> InputStream et OutputStream sont des classes de base en Java pour gérer les flux de données. InputStream permet de lire des données depuis une source vers le programme, tandis que OutputStream permet d'écrire des données du programme vers une destination. La principale différence est donc le sens du flux. Ces classes sont utilisées pour manipuler des fichiers, des flux réseau ou des buffers en mémoire.

⚡ Version "senior"

> InputStream et OutputStream représentent des flux binaires unidirectionnels. InputStream est utilisé pour consommer des données, OutputStream pour en produire. Ils constituent la base de l'API I/O Java, souvent combinés avec des buffers pour optimiser les performances, et avec des Reader/Writer lorsqu'on travaille avec du texte.

🎯 Punchline qui fait la différence

👉

> "InputStream lit, OutputStream écrit — tout est une question de sens du flux."

💥 Simple et efficace.

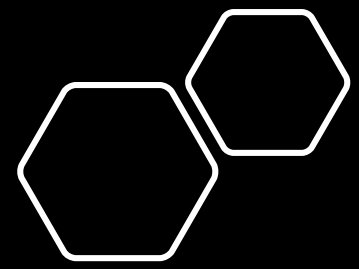
🚀 Petit bonus Angular / Java

👉 Backend Java :

- fichiers
- upload/download
- streaming HTTP

👉 Angular :

- côté client, on manipule plutôt des **streams RxJS**, mais le concept reste :
  👉 flux de données entrants/sortants



14. Pourquoi utilise-t-on async/await en JavaScript ?

🔹 Définition simple

👉 `async/await` sert à **gérer du code asynchrone de manière lisible et séquentielle**.

👉 En clair :

> écrire du code asynchrone **comme du code synchrone**, sans les callbacks imbriqués.

🔸 Pourquoi on en a besoin ?

JavaScript est **non bloquant** (single-thread avec event loop).

👉 Exemple classique :

- appel HTTP
- lecture fichier
- timer

👉 Ces opérations prennent du temps → on ne bloque pas le thread principal

---

🔥 Avant async/await (callbacks → problème)

JavaScript

```
getUser(function(user) {
  getOrders(user.id, function(orders) {
    getDetails(orders, function(details) {
      console.log(details);
    });
  });
});
```

👉😱 Callback hell :

- difficile à lire
- difficile à maintenir
- gestion d'erreurs compliquée

Avec Promises

JavaScript

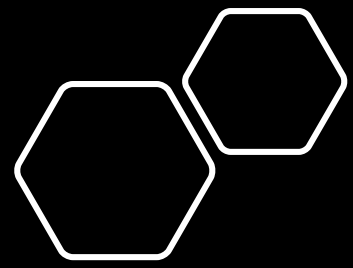
```
getUser()
  .then(user => getOrders(user.id))
  .then(orders => getDetails(orders))
  .then(details => console.log(details))
  .catch(error => console.error(error));
```

👉 mieux... mais encore verbeux

Avec async/await

JavaScript

```
async function loadData() {
  try {
    const user = await getUser();
    const orders = await getOrders(user.id);
    const details = await getDetails(orders);

    console.log(details);
  } catch (error) {
    console.error(error);
  }
}
```



🔷 Ce que fait réellement async/await

👉 Important à dire en entretien :

- `async` → une fonction retourne **toujours une Promise**
- `await` → attend la résolution d'une Promise

👉 Mais :

> ça ne rend pas le code synchrone — ça reste asynchrone sous le capot

🔸 Avantages

✅ 1. Lisibilité

Code proche du synchrone

✅ 2. Maintenance

Moins de complexité que `.then()`

✅ 3. Gestion d'erreurs propre

JavaScript

```
try {
  await call();
} catch (e) {
  // clair et propre
}
```

✅ 4. Moins de bugs

Moins d'imbrications → moins d'erreurs

🔸 Attention (point avancé)

👉 `await` est **bloquant dans la fonction async**

JavaScript

```
const a = await f1();
const b = await f2();
```

👉 exécuté **séquentiellement**

⚡ Optimisation

JavaScript

```
const [a, b] = await Promise.all([f1(), f2()]);
```

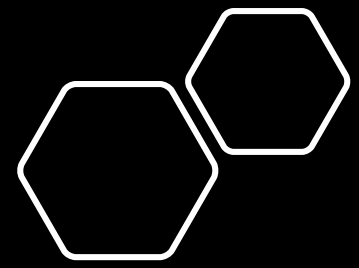
👉 exécution parallèle ✅

🔷 En Angular (très important)

👉 Angular utilise beaucoup RxJS (Observables)

Mais `async/await` est utilisé pour :

- appels HTTP simples
- logique séquentielle
- interop avec Promises

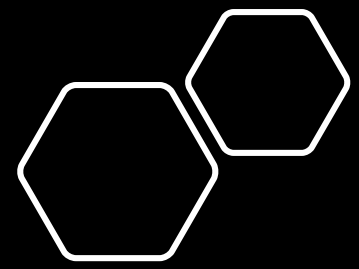


🔥 Réponse parfaite entretien

> async/await est utilisé en JavaScript pour simplifier la gestion du code asynchrone. Il permet d'écrire du code asynchrone de manière plus lisible et proche du code synchrone, en évitant les callbacks imbriqués ou les chaînes de Promises. Cela améliore la maintenabilité du code et facilite la gestion des erreurs avec des blocs try/catch.

⚡ Version "senior"

> async/await est une surcouche syntaxique des Promises qui améliore la lisibilité du code asynchrone. Il permet de structurer les flux asynchrones de manière séquentielle tout en restant non bloquant. En pratique, il faut toutefois faire attention à ne pas introduire des appels séquentiels inutiles et privilégier Promise.all pour les traitements parallèles.



JavaScript est-il monothread ? Pourquoi c’est important ?

🔹 Réponse courte

👉 Oui, JavaScript est monothread… mais avec un modèle asynchrone (event loop).

🔸 1. JavaScript est-il monothread ?

👉 Oui :

- un seul **thread principal d'exécution**
- une seule pile d'appel (**call stack**)
- une seule instruction exécutée à la fois

👉 Donc :

> JavaScript ne peut pas exécuter deux morceaux de code en même temps sur le thread principal.

🔸 2. Alors comment il gère l'asynchrone ?

👉 Grâce à :

- **Web APIs (navigateur)** ou APIs Node.js
- **Event Loop**
- **Task Queue / Microtask Queue**

💡 Schéma simple

```
Call Stack → Event Loop → Queue → Call Stack
```

Exemple

JavaScript

```javascript
console.log("A");

setTimeout(() => {
  console.log("B");
}, 0);

console.log("C");
```

👉 Résultat :

```
A
C
B
```

👉 Pourquoi ?

- `setTimeout` est délégué au navigateur
- callback exécuté plus tard via l'event loop

🔷 3. Pourquoi c'est important ?

👉 C'est LE point clé de la question.

✅ 1. Éviter de bloquer l'UI (Angular)

👉 Si tu fais :

```JavaScript
while(true) {}
```

👉 💥 freeze complet de l'application

✅ 2. Oblige à utiliser l'asynchrone

- Promises
- async/await
- Observables (Angular / RxJS)

✅ 3. Performance et UX

👉 Tout ce qui est long doit être :

- délégué (API, worker…)
- asynchrone

✅ 4. Pas de concurrence classique

👉 Contrairement à Java :

- pas de threads concurrents
- pas de `synchronized`
- moins de race conditions

👉 mais :

- attention aux **effets asynchrones**

🔶 4. Ce qu'il faut bien comprendre (niveau +)

👉 JavaScript est monothread...

👉 MAIS l'environnement ne l'est pas :

- navigateur → multi-thread interne
- Node.js → libuv (thread pool)

👉 JS délègue les tâches lourdes

🔷 5. Exemple Angular concret

👉 Appel HTTP :

TypeScript

```typescript
this.http.get('/api/users').subscribe(data => {
  console.log(data);
});
```

👉 Le thread principal :

- ne bloque pas
- continue de gérer l'UI

🔥 Réponse parfaite entretien

> JavaScript est monothread, ce qui signifie qu'il exécute une seule instruction à la fois sur un seul thread principal. Cependant, il gère l'asynchrone grâce à l'event loop et aux APIs de l'environnement, comme le navigateur ou Node.js. Cela permet d'éviter de bloquer le thread principal, notamment pour l'interface utilisateur. C'est important car toute opération longue doit être asynchrone, sinon elle bloque l'application. Ce modèle simplifie aussi la gestion de la concurrence, même s'il introduit d'autres complexités liées à l'asynchrone.

⚡ Version "senior"

> JavaScript est monothread au niveau de l'exécution du code, mais s'appuie sur un environnement multi-thread pour déléguer les opérations asynchrones. L'event loop orchestre le retour des callbacks dans la call stack. Cela impose une architecture non bloquante et explique l'usage massif de Promises, async/await et RxJS en Angular.