

Sommaire

- Utilité de la qualité de code
- Spécificités du front
- TP 1 (sans clean code)
- Présentation des bonnes pratiques JS/TS/Angular
- TP 2
- Refactoring
- Test Unitaire et TDD dans le clean code
- Intégration Lint, Sonar, CI/CD
- Tests e2e
- Performance et optimisation

Utilité de la qualité de code

Importance de la qualité du code : La qualité du code est cruciale pour plusieurs raisons :

- **Lisibilité** : Un code propre est plus facile à lire et à comprendre, non seulement pour le développeur qui l'écrit, mais aussi pour les autres membres de l'équipe qui pourraient devoir le maintenir ou le modifier à l'avenir.
- **Maintenance** : Un code bien structuré et sans redondance est plus facile à mettre à jour, ce qui est essentiel pour les projets à long terme.
- **Débogage** : Un code de qualité est généralement moins sujet aux erreurs et plus facile à déboguer, car les problèmes potentiels sont mieux isolés.
- **Performance** : Bien que la performance ne soit pas toujours directement liée à la propreté du code, un code bien écrit tend à être plus efficace en termes de ressources.
- **Collaboration** : Lorsque plusieurs développeurs travaillent sur le même projet, avoir un code propre permet une meilleure collaboration et réduit les frictions au sein de l'équipe.

Spécificités du front

Contexte du développement frontend avec Angular :

- **Modularité** : Angular favorise une approche modulaire, où chaque composant, service ou module a une responsabilité claire. Cela permet de réutiliser et de maintenir facilement le code.
- **Structure des composants** : Dans Angular, la séparation des composants selon leur rôle dans l'interface utilisateur est primordiale. Cela implique de découper l'application en composants logiques et de les organiser de manière cohérente.
- **Réactivité et performance** : Angular intègre des mécanismes pour gérer la réactivité des interfaces, via l'utilisation d'observables et de la gestion de l'état. Le clean code dans ce contexte signifie que chaque partie de l'application réagit de manière efficace aux changements de données.
- **Testing** : Angular inclut des outils comme Jasmine et Karma pour faciliter les tests unitaires et d'intégration, ce qui fait partie intégrante d'une stratégie de code propre.

Exercice : Développement d'une application de Bowling en Angular

Vous devez développer une application de bowling en Angular qui gère les scores de plusieurs joueurs.

1. **Créer un composant unique pour l'application de bowling.**

2. **Implémenter les fonctionnalités suivantes :**

- **Gestion des joueurs** : L'application doit permettre de gérer plusieurs joueurs.
- **Lancers aléatoires** : Pour chaque joueur, implémentez une fonctionnalité qui génère des lancers aléatoires.
- **Calcul du score** : Le score doit être calculé après chaque lancer, en prenant en compte les règles de base du bowling (strikes, spares).
- **Affichage du score** : Affichez le score de chaque joueur à l'écran.

Présentation des bonnes pratiques JS/TS/Angular

1. Nommage clair et significatif

Le nom des variables, fonctions, et classes doit être suffisamment descriptif pour indiquer clairement leur rôle. Un bon nom est explicite et permet de comprendre immédiatement l'intention du code sans nécessiter de commentaires ou de deviner le contexte.

Exemple :

- Mauvais : `let d: number;`
 - **Problème** : Le nom `d` est ambigu. On ne sait pas ce que cette variable représente.
- Bon : `let totalCost: number;`
 - **Pourquoi c'est mieux** : `totalCost` est explicite et décrit clairement que la variable stocke le coût total.

Recommandations pour Angular :

- **Variables** : Utilisez des noms de variables qui reflètent leur contenu ou leur rôle.
 - Ex : `let userList = [];` au lieu de `let list = [];`
- **Fonctions** : Le nom de la fonction doit refléter l'action ou le calcul effectué.
 - Ex : `calculateTotalCost()` au lieu de `doCalculation()`.
- **Classes et Services** : Les noms doivent refléter le rôle ou l'entité qu'ils représentent.
 - Ex : `UserService` pour un service qui gère les utilisateurs.

Présentation des bonnes pratiques JS/TS/Angular

Démonstration :

```
calculate() {  
  let t = this.totalAmount * 0.1;  
  return t;  
}
```

```
calculateTax() {  
  let taxAmount = this.totalAmount * 0.1;  
  return taxAmount;  
}
```

Présentation des bonnes pratiques JS/TS/Angular

2. Fonctions et méthodes courtes

Les fonctions doivent être aussi courtes que possible tout en accomplissant une tâche spécifique. L'objectif est de rendre chaque fonction compréhensible d'un coup d'œil. Une fonction courte est plus facile à tester et à maintenir. Si une fonction devient trop longue, elle devrait probablement être divisée en plusieurs fonctions plus petites.

Exemple :

- Mauvais :

```
calculateTotal() {  
  let subtotal = 0;  
  for (let i = 0; i < this.items.length; i++) {  
    subtotal += this.items[i].price * this.items[i].quantity;  
  }  
  let tax = subtotal * 0.1;  
  let discount = this.getDiscount();  
  let total = subtotal + tax - discount;  
  return total;  
}
```

Présentation des bonnes pratiques JS/TS/Angular

- Bon :

```
calculateTotal() {  
  const subtotal = this.calculateSubtotal();  
  const tax = this.calculateTax(subtotal);  
  const discount = this.getDiscount();  
  return subtotal + tax - discount;  
}  
  
private calculateSubtotal(): number {  
  return this.items.reduce((sum, item) => sum + item.price * item.quantity, 0);  
}  
  
private calculateTax(subtotal: number): number {  
  return subtotal * 0.1;  
}
```


Présentation des bonnes pratiques JS/TS/Angular

3. Un Seul Niveau d'Abstraction par Fonction

Une fonction ne doit pas mélanger différents niveaux d'abstraction. Par exemple, une fonction qui orchestre des actions de haut niveau ne devrait pas s'encombrer de détails de bas niveau. Cela aide à maintenir la clarté de la logique de la fonction et facilite sa compréhension.

Exemple :

- Mauvais :

```
renderPage() {  
  this.fetchData();  
  this.renderHeader();  
  this.showSpinner();  
}
```

- Bon :

```
renderPage() {  
  this.showLoadingState();  
  this.fetchData().then(() => {  
    this.hideLoadingState();  
    this.renderContent();  
  });  
}  
  
private showLoadingState() {  
  this.renderHeader();  
  this.showSpinner();  
}
```

Présentation des bonnes pratiques JS/TS/Angular

4. Code DRY (Don't Repeat Yourself)

Le principe DRY consiste à éviter les duplications de code. Le code dupliqué augmente la charge de maintenance car il peut entraîner des incohérences et nécessite des modifications multiples en cas de changement. L'objectif est de centraliser les logiques communes dans des fonctions ou des services réutilisables.

Exemple :

- Mauvais :

```
this.saveUser(user);  
this.logger.log('User saved successfully.');
```

```
this.notificationService.notify('User has been saved.');
```

- Bon :

```
this.saveUser(user);  
this.handlePostSave('User saved successfully.', 'User has been saved.');
```

```
private handlePostSave(logMessage: string, notificationMessage: string) {  
  this.logger.log(logMessage);  
  this.notificationService.notify(notificationMessage);  
}
```

Présentation des bonnes pratiques JS/TS/Angular

5. Utilisation de l'Égalité faible vs. Égalité stricte

- **Égalité Faible (==)** : L'égalité faible compare deux valeurs en effectuant une conversion implicite de type, ce qui peut entraîner des résultats inattendus. Par exemple, `0 == false` retourne `true`, car JavaScript convertit `false` en `0` avant de faire la comparaison.
- **Égalité Stricte (===)** : L'égalité stricte compare les valeurs sans conversion de type. Si les deux valeurs sont de types différents, la comparaison retourne `false`. Par exemple, `0 === false` retourne `false` parce que `0` est un nombre et `false` est un booléen.
- **Prévisibilité** : L'égalité stricte garantit que seules des valeurs du même type et de la même valeur seront considérées comme égales. Cela évite les comportements inattendus qui peuvent survenir avec l'égalité faible.
- **Clarté** : En utilisant l'égalité stricte, le code devient plus lisible et compréhensible. Les développeurs peuvent être sûrs que la comparaison est faite sans conversion implicite de type, ce qui rend le code plus clair et plus facile à comprendre.
- **Réduction des bugs** : Les conversions implicites peuvent introduire des bugs subtils difficiles à détecter. En utilisant l'égalité stricte, vous réduisez le risque de telles erreurs, ce qui améliore la qualité globale du code.

Présentation des bonnes pratiques JS/TS/Angular

Mauvais Exemple :

```
const userAge = '18'; // string

if (userAge == 18) {
  console.log('User is an adult.');
```

Bon Exemple :

```
const userAge = '18'; // string

if (userAge === 18) {
  console.log('User is an adult.');
```

```
} else {
  console.log('User age is not a number.');
```

```
}
```

Présentation des bonnes pratiques JS/TS/Angular

6. Gestion des Erreurs

La gestion des erreurs dans Angular (et JavaScript en général) consiste à capturer, traiter et éventuellement signaler les erreurs qui peuvent survenir lors de l'exécution de l'application. Une bonne gestion des erreurs est essentielle pour créer des applications robustes et résilientes, conformes aux principes de Clean Code.

- **Anticipation des erreurs** : Un code propre anticipe les erreurs possibles et les gère de manière appropriée. Cela signifie prévoir des cas où les entrées peuvent être invalides, où des requêtes réseau peuvent échouer, etc.
- **Gérer les erreurs localement** : Les erreurs doivent être capturées et gérées aussi près que possible de l'endroit où elles se produisent, ce qui rend le code plus robuste et maintenable.
- **Clarté dans le traitement des erreurs** : Les blocs de gestion des erreurs (`try...catch`) doivent être clairs et ne pas cacher les erreurs. Il est essentiel d'avoir une stratégie claire pour gérer chaque type d'erreur.
- **Ne pas masquer les erreurs** : Évitez d'ignorer les erreurs ou de les masquer sans raison. Chaque erreur doit être logiquement traitée ou, si elle ne peut pas être gérée localement, propagée de manière appropriée pour être traitée ailleurs.

Présentation des bonnes pratiques JS/TS/Angular

Mauvais Exemple :

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

Bon Exemple :

```
getData() {  
  this.http.get('/api/data')  
    .subscribe({  
      next: (data) => {  
        console.log(data);  
      },  
      error: (error) => {  
        console.error('An error occurred:', error);  
        this.handleError(error);  
      }  
    });  
}  
  
handleError(error: any) {  
  // Logique pour traiter l'erreur (affichage d'un message, etc.)  
  alert('Failed to load data. Please try again later.');
```

Présentation des bonnes pratiques JS/TS/Angular

7. Commentaires

- **Le Principe de Base : Un Code Auto-Explicatif**

Le principe fondamental de Clean Code est que le code lui-même doit être suffisamment clair pour que les commentaires ne soient pas nécessaires pour comprendre ce qu'il fait. Cela signifie que les noms des variables, fonctions, classes, et la structure du code doivent être explicites et bien choisis.

- **Lisibilité améliorée** : Un code bien nommé et structuré est plus facile à lire et à comprendre, ce qui réduit le besoin de commentaires explicatifs.
- **Éviter la redondance** : Les commentaires qui décrivent ce que fait le code plutôt que pourquoi il le fait sont souvent redondants et peuvent devenir obsolètes si le code change mais pas les commentaires.
- **Réduction des erreurs** : Les commentaires peuvent devenir des sources de confusion s'ils ne sont pas maintenus à jour. Un commentaire incorrect peut induire en erreur, contrairement à un code clair et précis.

Mauvais Exemple :

```
// Calculer le coût total en ajoutant la taxe au sous-total  
let totalCost = subtotal + (subtotal * taxRate);
```

Bon Exemple :

```
let totalCost = calculateTotalCost(subtotal, taxRate);
```

Présentation des bonnes pratiques JS/TS/Angular

Quand utiliser des commentaires

Bien que l'objectif soit de minimiser les commentaires, il existe des situations où les commentaires sont nécessaires et bénéfiques. Ces commentaires doivent être utilisés pour expliquer le **pourquoi** d'une décision de conception ou pour fournir des informations contextuelles qui ne sont pas évidentes dans le code lui-même.

Cas où les commentaires sont justifiés :

- **Explication des décisions complexes** : Lorsque vous avez pris une décision de conception non triviale qui pourrait ne pas être immédiatement claire pour un autre développeur.
- **Avertissements** : Indiquer les conséquences potentielles de modifier une certaine partie du code.
- **Documentation des API publiques** : Si vous développez des API ou des services publics, il est important de documenter leur utilisation, leurs paramètres, et leurs comportements.
- **TODOs et FIXMEs** : Utilisez des commentaires pour indiquer des tâches inachevées ou des parties du code qui nécessitent une attention future.

Présentation des bonnes pratiques JS/TS/Angular

Bon Exemple de Commentaire :

```
// Contournement du bug #1234 dans la librairie X. Ce code doit être supprimé après la mise à jour de la librairie X à la version 2.0.  
fixLibraryBug() {  
  // Code de contournement ici  
}
```

Présentation des bonnes pratiques JS/TS/Angular

3. Types de commentaires à éviter

Types de commentaires à éviter :

- **Commentaires évidents** : Des commentaires qui répètent simplement ce que fait le code.
- **Commentaires obsolètes** : Des commentaires qui ne sont pas mis à jour lorsque le code change, créant une source potentielle de confusion.
- **Commentaires de "remplissage"** : Des commentaires ajoutés pour respecter une convention de style ou de documentation sans réelle valeur ajoutée.

Mauvais Exemple :

```
// Incrémenter le compteur  
counter += 1;
```

Présentation des bonnes pratiques JS/TS/Angular

4. Documentation des API et Services

La documentation des API et des services est essentielle lorsque vous exposez des fonctionnalités à d'autres parties de votre application ou à des développeurs externes. Ces commentaires doivent être clairs, concis, et fournir toutes les informations nécessaires pour utiliser les API correctement.

- **Utiliser JSDoc ou TypeDoc** : Pour documenter les méthodes publiques, les paramètres, et les retours de fonction.
- **Expliquer les comportements inattendus ou les effets secondaires** : Informer les utilisateurs de l'API sur tout comportement non trivial ou effets secondaires.

Présentation des bonnes pratiques JS/TS/Angular

Bon Exemple :

```
/**
 * Crée un nouvel utilisateur dans le système.
 * @param user - Les informations de l'utilisateur à créer.
 * @returns Observable<User> - Un observable qui émet l'utilisateur créé.
 * @throws UserAlreadyExistsError - Si un utilisateur avec le même email existe déjà.
 */
createUser(user: User): Observable<User> {
  // Code pour créer un utilisateur
}
```

Présentation des bonnes pratiques JS/TS/Angular

8. Typage en TypeScript

TypeScript est un sur-ensemble de JavaScript qui introduit le typage statique. Le typage statique signifie que les types des variables, des paramètres de fonction, et des valeurs de retour sont connus et vérifiés au moment de la compilation. Cela permet de détecter les erreurs potentielles plus tôt dans le cycle de développement et de rendre le code plus explicite et sûr.

- **Détection précoce des erreurs** : Le typage statique aide à identifier les erreurs de type avant même que le code ne soit exécuté, réduisant ainsi les bugs en production.
- **Clarté et lisibilité** : Le typage explicite améliore la lisibilité du code en rendant les intentions du développeur claires. Un développeur qui lit le code peut comprendre immédiatement quels types de données sont attendus, sans avoir à deviner ou à exécuter le code.
- **Auto-documentation** : Le typage en TypeScript sert de documentation intégrée, rendant souvent les commentaires superflus pour expliquer les types de données manipulés par les fonctions et les méthodes.
- **Sécurité** : En définissant explicitement les types, vous réduisez les risques de comportements imprévisibles causés par des valeurs inattendues.

Mauvais Exemple (JavaScript pur) :

```
function calculateTotal(price, quantity) {  
  return price * quantity;  
}  
const total = calculateTotal(10, '5'); // total sera "105", ce qui est incorrect
```

Présentation des bonnes pratiques JS/TS/Angular

Bon Exemple (TypeScript avec Typage) :

```
function calculateTotal(price: number, quantity: number): number {  
    return price * quantity;  
}  
  
const total = calculateTotal(10, 5); // total sera 50, ce qui est correct
```

9. Utilisation des Interfaces et des Types

TypeScript permet de définir des interfaces et des types pour structurer les objets et les fonctions. Les interfaces définissent la forme d'un objet, tandis que les types peuvent combiner plusieurs types existants ou être utilisés pour des unions de types.

Pourquoi utiliser des interfaces et des types :

- **Clarification des attentes** : Les interfaces et les types rendent explicite la structure des données manipulées, ce qui améliore la compréhension du code et réduit les erreurs.
- **Réutilisabilité** : En définissant des interfaces réutilisables, vous pouvez appliquer une structure cohérente à travers l'ensemble de l'application, améliorant ainsi la cohérence et la maintenabilité.
- **Encapsulation** : Les interfaces permettent de masquer les détails de l'implémentation tout en exposant seulement ce qui est nécessaire, ce qui est un principe clé de Clean Code.

Présentation des bonnes pratiques JS/TS/Angular

Mauvais Exemple :

```
function getUserInfo(user) {  
  console.log(user.name);  
  console.log(user.email);  
}
```

Bon Exemple :

```
interface User {  
  name: string;  
  email: string;  
}  
  
function getUserInfo(user: User) {  
  console.log(user.name);  
  console.log(user.email);  
}
```

Présentation des bonnes pratiques JS/TS/Angular

10. Modules TypeScript

Les modules TypeScript sont des fichiers qui encapsulent du code (classes, fonctions, constantes, etc.) et les exportent pour qu'ils puissent être utilisés dans d'autres parties de l'application. L'utilisation de modules favorise une organisation claire du code, améliore la réutilisabilité, et permet de maintenir un code propre et bien structuré.

- **Organisation** : Les modules permettent de diviser l'application en parties logiques et cohérentes, ce qui rend le code plus facile à gérer et à naviguer.
- **Encapsulation** : Les modules limitent la portée des variables et des fonctions à l'intérieur du module, réduisant ainsi les risques de conflits de nom et rendant le code plus sûr.
- **Réutilisabilité** : Le code encapsulé dans un module peut être facilement réutilisé dans d'autres modules ou applications, suivant le principe DRY (Don't Repeat Yourself) du Clean Code.

Présentation des bonnes pratiques JS/TS/Angular

11. Respect des Principes SOLID avec TypeScript et Angular

Les principes SOLID sont des principes de conception orientés objet qui sont essentiels pour écrire un code propre et maintenable. TypeScript, avec son système de typage et ses modules, facilite l'application de ces principes dans Angular.

- **Single Responsibility Principle (SRP)** : Chaque module ou classe doit avoir une seule responsabilité.
- **Open/Closed Principle (OCP)** : Les modules doivent être ouverts à l'extension mais fermés à la modification.
- **Liskov Substitution Principle (LSP)** : Les objets d'une classe dérivée doivent pouvoir remplacer les objets de la classe mère sans affecter le fonctionnement du programme.
- **Interface Segregation Principle (ISP)** : Les clients ne doivent pas être obligés de dépendre d'interfaces qu'ils n'utilisent pas.
- **Dependency Inversion Principle (DIP)** : Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.

Sujet du TP

En utilisant l'application de Gestion des commandes. Adopter les Principes de Clean Code et SOLID

Contexte :

Vous êtes chargé de refactorer une application de gestion des commandes pour un petit commerce. Le code de cette application a été écrit sans suivre les bonnes pratiques de Clean Code et SOLID. Votre tâche est de corriger ces problèmes en appliquant les principes suivants :

1. Nommage clair et significatif
2. Fonctions et méthodes courtes
3. Un seul niveau d'abstraction par fonction
4. Application du principe DRY (Don't Repeat Yourself)
5. Utilisation de l'égalité stricte (===)
6. Gestion appropriée des erreurs
7. Utilisation appropriée des commentaires
8. Utilisation du typage en TypeScript
9. Organisation modulaire avec TypeScript
10. Application des principes SOLID

Refactoring

Le **refactoring** est le processus de modification du code source d'un programme pour en améliorer la structure interne sans en changer le comportement externe. Autrement dit, le but du refactoring est de rendre le code plus propre, plus lisible, plus maintenable et plus évolutif, tout en s'assurant que l'application fonctionne exactement de la même manière avant et après les modifications.

1. **Amélioration de la lisibilité** : Un code bien organisé et bien structuré est plus facile à lire et à comprendre, ce qui permet aux développeurs de travailler plus efficacement.
2. **Réduction de la complexité** : En simplifiant les structures complexes et en décomposant les fonctions longues, le refactoring aide à réduire la complexité cognitive, ce qui facilite la compréhension et la maintenance du code.
3. **Facilitation de la maintenance** : Le code propre est plus facile à corriger, à améliorer et à étendre. Le refactoring aide à identifier et à éliminer les mauvaises pratiques de codage, rendant le code plus robuste et moins sujet aux erreurs.
4. **Préparation pour de nouvelles fonctionnalités** : Avant d'ajouter de nouvelles fonctionnalités à une base de code, le refactoring peut être nécessaire pour rendre la structure du code plus apte à intégrer ces nouvelles fonctionnalités sans introduire de bugs ou de dettes techniques.
5. **Réduction de la duplication de code** : Le refactoring permet de supprimer le code redondant en centralisant les logiques communes, ce qui suit le principe DRY ("Don't Repeat Yourself").

Refactoring

1. **Identification des zones à améliorer** : Analyser le code pour repérer les "odeurs de code" ou "code smells", qui sont des signes de code nécessitant une amélioration. Cela peut inclure des fonctions trop longues, des noms de variables ambigus, de la duplication de code, une mauvaise gestion des erreurs, etc.
2. **Application des techniques de refactoring** : Utiliser des techniques spécifiques pour améliorer la structure du code. Parmi ces techniques, on peut citer :
 - **Extraction de fonction/méthode** : Décomposer une fonction complexe en plusieurs fonctions plus simples et plus courtes.
 - **Renommage** : Renommer les variables, les fonctions ou les classes pour qu'elles soient plus descriptives et reflètent mieux leur rôle.
 - **Encapsulation** : Regrouper des données et des comportements dans une classe ou un module pour améliorer la modularité et l'encapsulation.
 - **Décomposition des conditions complexes** : Simplifier les conditions complexes en les décomposant en sous-conditions plus simples.
3. **Test continu** : S'assurer que les tests unitaires et fonctionnels sont toujours réussis après chaque étape de refactoring. Cela garantit que le comportement de l'application reste inchangé.

Refactoring

1. Comprendre le code existant

- **Analyse du Code** : Avant de commencer le refactoring, il est crucial de bien comprendre le code existant. Cela implique de lire le code, de comprendre ses fonctionnalités, ses dépendances, et de s'assurer que vous connaissez le comportement attendu.
- **Identification des "Code Smells"** : Recherchez les signes de mauvaise qualité du code, comme des fonctions trop longues, des variables mal nommées, des duplications, des structures complexes, etc. Ce sont des indicateurs qu'un refactoring est nécessaire.

2. Assurer une couverture de tests adéquate

- **Écriture de Tests** : Avant de modifier le code, assurez-vous qu'il existe des tests unitaires et fonctionnels qui couvrent les fonctionnalités critiques. Si nécessaire, écrivez des tests pour les parties non couvertes. Ces tests serviront de garde-fou pour garantir que le comportement du code reste inchangé après le refactoring.
- **Exécution des Tests** : Exécutez les tests existants pour vous assurer qu'ils passent tous avant de commencer le refactoring. Cela garantit que vous partez d'un code fonctionnel.

Refactoring

3. Appliquer des Techniques de Refactoring

- **Travailler Par Petites Étapes** : Le refactoring doit être effectué par petites étapes incrémentales. Cela permet de minimiser les risques d'introduire des bugs et de faciliter la gestion des changements.
- **Techniques de Refactoring Communes** :
 - **Extraction de Méthode/Fonction** : Si une fonction est trop longue ou fait plusieurs choses, extrayez certaines parties de la fonction dans de nouvelles fonctions avec des noms explicites.
 - **Renommage** : Changez les noms des variables, des fonctions, ou des classes pour qu'ils soient plus descriptifs et reflètent mieux leur rôle.
 - **Encapsulation** : Déplacez les données et les comportements associés dans des classes ou des modules appropriés, réduisant ainsi les dépendances globales.
 - **Décomposition de Conditions** : Si une condition est complexe, divisez-la en plusieurs sous-conditions plus simples et plus lisibles.
 - **Réduction des Duplications** : Identifiez et centralisez les parties du code qui sont dupliquées pour réduire la répétition et faciliter la maintenance.
 - **Simplification des Structures** : Si une structure de contrôle (comme une boucle ou une condition) est trop complexe, essayez de la simplifier ou de la restructurer.

Refactoring

Le concept de "**Ne porter qu'une casquette à la fois**" dans le contexte du refactoring fait référence à l'idée de se concentrer sur une seule tâche ou un seul type de modification à la fois lorsque vous travaillez sur du code. Cette approche est cruciale pour éviter la confusion, réduire les risques d'introduire des bugs et faciliter la compréhension du processus de refactoring.

- **Refactoring** : Modifier la structure du code sans changer son comportement.
 - **Ajout de fonctionnalité** : Introduire de nouvelles fonctionnalités sans nécessairement améliorer ou modifier le code existant.
 - **Correction de bug** : Résoudre un problème spécifique sans toucher à d'autres parties du code.
1. **Clarté** : En se concentrant sur une seule tâche à la fois, il devient plus facile de comprendre et de suivre les modifications apportées au code. Cela facilite la révision du code par d'autres développeurs et réduit les risques d'introduire des erreurs.
 2. **Simplicité des Tests** : Si vous modifiez plusieurs aspects du code en même temps, il devient difficile de tester chaque modification individuellement. En refactorant d'abord le code avant d'ajouter de nouvelles fonctionnalités ou de corriger des bugs, vous pouvez être sûr que chaque modification est bien couverte par des tests.

Refactoring

3. **Minimisation des Risques** : Le fait de séparer les différentes tâches permet de minimiser les risques d'introduire des bugs. Si vous changez la structure du code (refactoring) et ajoutez une nouvelle fonctionnalité en même temps, il devient difficile de savoir quelle modification est responsable en cas de problème.
4. **Amélioration de la Maintenabilité** : En appliquant ce principe, le code reste propre et organisé, ce qui facilite sa maintenabilité à long terme. Cela évite également l'accumulation de "dette technique".

Refactoring

1. Identifier la tâche à accomplir :

- Avant de commencer à travailler, déterminez clairement ce que vous allez faire : refactoring, ajout de fonctionnalité, ou correction de bug. Ne mélangez pas ces activités.

2. Diviser les étapes :

- Si le code nécessite à la fois un refactoring et une nouvelle fonctionnalité, commencez par refactorer le code existant pour le rendre plus propre et plus facile à étendre. Ensuite, ajoutez la nouvelle fonctionnalité une fois que la base est propre.

3. Utiliser le contrôle de version :

- Utilisez des commits séparés pour chaque type de modification. Par exemple, faites un commit pour le refactoring, puis un autre pour l'ajout d'une fonctionnalité. Cela permet de revenir facilement en arrière si une modification introduit un problème.

4. Tester après chaque modification :

- Testez votre code après chaque changement significatif. Si vous refactorisez du code, exécutez vos tests pour vous assurer que le comportement reste inchangé. Puis, testez à nouveau après avoir ajouté une nouvelle fonctionnalité.

Refactoring

5. Réviser et itérer :

- Une fois que vous avez effectué une série de modifications, révisez votre travail pour vous assurer que vous n'avez pas introduit de bugs ou de régressions. Si nécessaire, faites des ajustements supplémentaires.

Architecture Angular

- Une architecture évolutive est une manière de concevoir une application de manière à ce qu'elle puisse grandir et évoluer sans rencontrer de problèmes majeurs. Cela signifie qu'au fur et à mesure que l'application devient plus complexe, qu'elle traite plus de données ou que les besoins des utilisateurs changent, elle continue de fonctionner efficacement.
- Lorsque nous parlons d'une application évolutive, nous parlons de la capacité à gérer la croissance. Si l'architecture est mal conçue, vous pourriez vous retrouver avec du code difficile à maintenir, des délais de développement plus longs et des coûts qui explosent. Une bonne architecture aide à éviter ces pièges et permet de maintenir une haute qualité dans l'application, même à long terme.

Architecture Angular - Scalabilité

Contraintes spécifiques :

Contrairement aux applications backend, où le défi est souvent de gérer un grand nombre d'utilisateurs simultanément, les applications avec une interface graphique (GUI) doivent gérer des aspects différents de la scalabilité :

- **Taille des données** : À mesure que plus de données sont chargées dans l'application, celle-ci doit continuer de fonctionner sans ralentissements.
- **Complexité croissante** : Plus l'application devient complexe (avec de nouvelles fonctionnalités et plus de code), plus il devient difficile de la maintenir. Il est crucial d'avoir une architecture qui peut supporter cette complexité sans se dégrader.
- **Temps de chargement** : Lorsque le projet grandit, les temps de chargement peuvent augmenter. Une bonne architecture aide à minimiser cet impact.

Objectif :

Le but d'une architecture évolutive est de s'assurer que, peu importe la taille de l'application ou la quantité de données qu'elle traite, l'expérience utilisateur reste fluide et agréable. Cela signifie des temps de réponse rapides, peu de bugs, et une interface qui reste réactive.

Architecture Angular

Architecture non évolutive :

Si l'architecture d'une application n'est pas pensée pour évoluer, plusieurs problèmes peuvent survenir :

- **Difficulté accrue de développement** : Le code peut devenir si complexe que chaque nouvelle fonctionnalité prend beaucoup plus de temps à implémenter.
- **Dette technique** : C'est lorsqu'on accumule des "solutions temporaires" ou du code de mauvaise qualité pour aller plus vite, ce qui finit par ralentir considérablement le développement à long terme.
- **Coût élevé** : Tous ces problèmes se traduisent par des coûts plus élevés, car il faut plus de temps et d'efforts pour maintenir et faire évoluer l'application.

Architecture Angular

Équilibre entre simplicité et efficacité :

Une bonne architecture doit être flexible et capable de s'adapter, que l'application soit petite ou grande. Elle doit permettre une excellente expérience utilisateur, même si l'application traite une grande quantité de données.

Règles claires pour les développeurs :

Pour que le code reste propre et facile à maintenir, il est important d'établir des règles claires que tous les développeurs peuvent suivre. Cela aide à maintenir la cohérence dans le code et à réduire les erreurs.

Réduction de la courbe d'apprentissage :

L'architecture doit être suffisamment simple pour que les nouveaux développeurs puissent rapidement comprendre comment l'application est construite et se mettre au travail sans avoir besoin d'un long temps d'adaptation.

Architecture Angular

Structure de Projet Modulaire

Organisation claire :

Dans une application bien structurée, chaque fonctionnalité ou module doit être clairement séparé dans l'arborescence des fichiers. Cela permet de savoir où trouver et modifier une fonctionnalité spécifique.

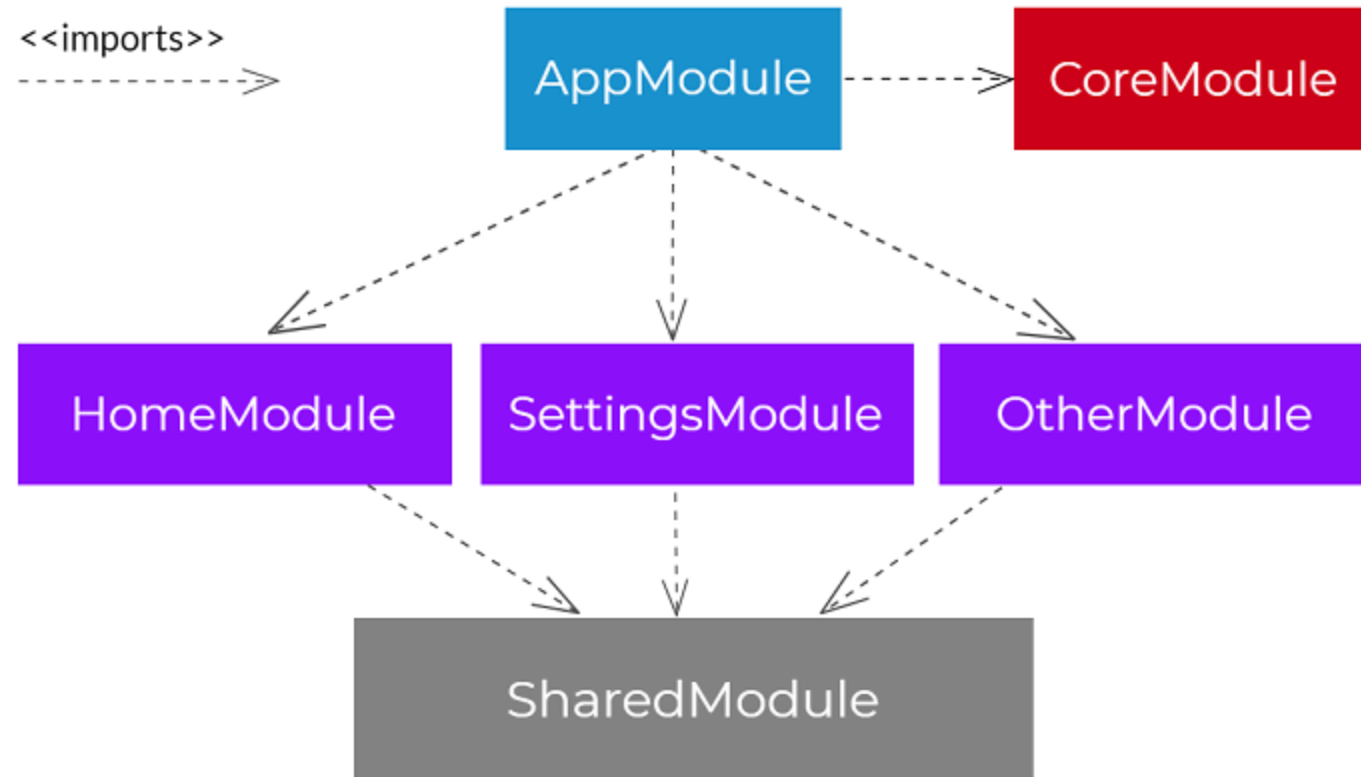
Isolation des modules :

Chaque module doit être autonome, c'est-à-dire qu'il ne doit pas dépendre des autres modules pour fonctionner. Par exemple, si vous supprimez un module, l'application devrait continuer à fonctionner sans problème. Cette isolation simplifie la maintenance et réduit les risques d'erreurs lorsqu'une modification est apportée.

Avantages :

Cette structure modulaire facilite la gestion du code et permet à plusieurs développeurs de travailler sur différents modules sans se gêner. Cela rend également le code plus propre et plus facile à comprendre, ce qui est essentiel pour la scalabilité.

Architecture Angular



Test avec Angular - TDD

Le TDD est une méthode de développement logiciel qui préconise d'écrire les tests avant même de coder la fonctionnalité elle-même. Cette approche se décompose en trois étapes principales :

1. **Écrire un test qui échoue** : Avant d'ajouter une nouvelle fonctionnalité, vous écrivez un test qui décrit le comportement attendu de cette fonctionnalité. Au départ, le test échouera car la fonctionnalité n'existe pas encore.
2. **Faire passer le test** : Ensuite, vous écrivez le code minimal nécessaire pour faire passer le test. Cette étape oblige le développeur à se concentrer uniquement sur les exigences de la fonctionnalité.
3. **Refactoriser le code** : Après que le test passe, vous pouvez refactoriser votre code avec l'assurance que les tests existants vous aideront à détecter toute régression introduite par le refactoring.

Test avec Angular - Rappel outils test

- **Jasmine** : Jasmine est un framework de test pour JavaScript qui est utilisé pour tester le code de manière asynchrone. C'est un framework de développement piloté par le comportement (BDD pour Behavior-Driven Development), ce qui signifie qu'il permet de décrire les tests d'une manière qui se rapproche du comportement attendu de l'application.
 - **Syntaxe claire et lisible** : Jasmine a une syntaxe simple qui permet de décrire les cas de test et les attentes de manière compréhensible. Cela rend les tests plus lisibles non seulement pour les développeurs mais aussi pour les non-développeurs.
 - **Tests isolés** : Jasmine fournit des moyens d'isoler les tests en utilisant des `spies` pour surveiller et contrôler l'accès aux méthodes spécifiques, permettant ainsi de tester les composants en isolation sans dépendre d'autres parties du système.
 - **Support asynchrone** : Jasmine supporte les tests asynchrones, ce qui est crucial pour tester les opérations basées sur les promesses ou les observables dans Angular.
 - **Pas besoin de DOM** : Bien que Jasmine puisse être utilisé pour tester du code qui interagit avec le DOM, il n'a pas besoin d'un navigateur pour exécuter les tests. Cela dit, dans le contexte d'Angular, Karma est souvent utilisé pour exécuter les tests de Jasmine dans un environnement de navigateur.

Test avec Angular - Rappel outils test

- **Karma:** Karma est un lanceur de test pour les projets JavaScript. Il est conçu pour travailler avec n'importe quel framework de test (comme Jasmine, Mocha, QUnit) et est donc très flexible. Dans le contexte d'Angular, Karma est utilisé pour exécuter les tests écrits avec Jasmine dans un navigateur réel, un navigateur sans tête (Headless Chrome, par exemple), ou même plusieurs navigateurs.
 - **Intégration facile avec les projets Angular :** Karma est configuré par défaut lorsque vous créez un nouveau projet Angular avec Angular CLI, ce qui rend le démarrage des tests unitaires et d'intégration rapide et sans douleur.
 - **Feedback en temps réel :** Karma permet de surveiller les fichiers de projet et de réexécuter automatiquement les tests à chaque modification, offrant un retour rapide sur les changements apportés.
 - **Prise en charge des navigateurs multiples :** Karma peut lancer les tests dans plusieurs navigateurs simultanément, ce qui est utile pour tester la compatibilité entre navigateurs.
- **Utilisation de Jasmine avec Karma dans Angular:** Lorsque vous écrivez des tests dans un projet Angular, vous utilisez la syntaxe de Jasmine pour définir des `describe` blocks (qui regroupent les tests) et des `it` blocks (qui définissent les tests individuels). Vous utilisez des assertions fournies par Jasmine, comme `expect`, pour vérifier que le code se comporte comme prévu.

TestBed

TestBed est un module de test fourni par le package `@angular/core/testing` qui permet de simuler l'environnement d'une application Angular. Il crée un module Angular dynamiquement que vous pouvez configurer en fonction des besoins de vos tests, en imitant le fonctionnement de votre application réelle. Cela vous permet de tester des composants, des directives, des services, etc., dans un contexte qui ressemble de près à leur utilisation dans une application Angular.

1. Caractéristiques Principales de TestBed

- **Configuration de Module Dynamique** : Avec TestBed, vous pouvez configurer dynamiquement le module de test en spécifiant les composants, les directives, les pipes, les services, etc., que vous souhaitez inclure dans le test. Cela inclut la déclaration de composants, l'importation de modules, la fourniture de services, et plus encore.
- **Création de Composants** : TestBed facilite la création d'instances de composants pour les tests, permettant de tester leur comportement en isolation ou en interaction avec d'autres composants et services.
- **Injection de Dépendances** : TestBed prend en charge l'injection de dépendances dans les tests, ce qui vous permet d'injecter des services dans vos composants de test, de fournir des mocks ou des stubs pour les dépendances externes, et d'accéder aux instances de services configurés pour le module de test.
- **Accès au DOM** : Lorsque vous créez des composants avec TestBed, vous pouvez accéder à leur DOM pour tester leur rendu et leurs interactions avec l'utilisateur. Cela est particulièrement utile pour tester les bindings, les événements et le rendu conditionnel.

Test - spy

Un "spy" est une fonction qui vous permet de surveiller l'utilisation d'une fonction que cela soit sur des objets existants ou des objets simulés (mocks). L'utilisation d'un spy permet de vérifier si des fonctions ont été appelées, combien de fois elles l'ont été, avec quels arguments, et quelles valeurs elles ont retournées. Les spies sont particulièrement utiles pour les tests unitaires où vous avez besoin d'isoler les comportements des composants ou des services que vous testez, sans dépendre de l'exécution réelle d'autres parties de votre application.

- **Surveiller les appels à la fonction** : Vous pouvez vérifier si une fonction a été appelée et combien de fois.
- **Inspecter les arguments** : Les spies peuvent enregistrer les arguments passés aux fonctions lors de leurs appels, ce qui vous permet de vérifier si la fonction a été appelée avec les bons arguments.
- **Contrôler les valeurs de retour** : Un spy peut retourner une valeur spécifique lorsqu'il est appelé, vous permettant de simuler différentes réponses pour vos tests.
- **Simuler des erreurs** : Vous pouvez configurer un spy pour qu'il lance une erreur lorsqu'il est appelé, ce qui est utile pour tester le comportement de gestion des erreurs.
- **Créer des mocks** : Les spies peuvent être utilisés pour créer des versions simulées (mocks) de fonctions ou de services, isolant ainsi le code que vous testez de ses dépendances.

Test - spy

Pour surveiller si une méthode d'un objet a été appelée :

```
let myObj = { myMethod: () => {} };
spyOn(myObj, 'myMethod');

myObj.myMethod();

expect(myObj.myMethod).toHaveBeenCalled();
```

Pour faire en sorte qu'un spy retourne une valeur spécifique :

```
let myObj = { myMethod: () => {} };
spyOn(myObj, 'myMethod').and.returnValue(42);

expect(myObj.myMethod()).toBe(42);
```

Pour simuler une erreur lors de l'appel à une méthode :

```
let myObj = { myMethod: () => {} };
spyOn(myObj, 'myMethod').and.throwError('error');

expect(() => myObj.myMethod()).toThrowError('error');
```

Pour inspecter les arguments avec lesquels une méthode a été appelée :

```
let myObj = { myMethod: () => {} };
spyOn(myObj, 'myMethod');

myObj.myMethod('arg1', 'arg2');

expect(myObj.myMethod).toHaveBeenCalledWith('arg1', 'arg2');
```

Lazy Loading

- Le Lazy Loading, ou chargement paresseux, est une stratégie d'optimisation importante dans le développement d'applications Angular. Il consiste à charger les modules de l'application sur demande plutôt que lors du chargement initial de l'application.
- Cela permet de réduire significativement le temps de chargement initial de l'application, améliorant l'expérience utilisateur, surtout pour ceux avec des connexions Internet lentes, et optimisant l'utilisation de la bande passante en ne chargeant que les ressources nécessaires.
- Pour implémenter le Lazy Loading, vous devez configurer les routes de l'application pour charger les modules de manière paresseuse. Voici comment cela pourrait être fait dans le fichier de routing principal de l'application (app-routing.module.ts) :

```
const routes: Routes = [  
  { path: '', redirectTo: '/home', pathMatch: 'full' },  
  { path: 'home', loadChildren: () => import('./home/home.module').then(m => m.HomeModule) },  
  { path: 'products', loadChildren: () => import('./products/products.module').then(m => m.ProductsModule) },  
  { path: 'about', loadChildren: () => import('./about/about.module').then(m => m.AboutModule) }  
];  
  
@NgModule({  
  imports: [RouterModule.forRoot(routes)],  
  exports: [RouterModule]  
})  
export class AppRoutingModule { }
```

Stratégies de Préchargement

- Angular propose plusieurs stratégies de préchargement pour les modules.
- Ces stratégies peuvent être utilisées pour améliorer davantage l'expérience utilisateur en chargeant à l'avance les modules qui seront probablement nécessaires dans un futur proche, sans attendre que l'utilisateur navigue vers une route spécifique.
- **Stratégies Disponibles**
 - **PreloadAllModules**: Précharge tous les modules lazy loaded dès que possible, après le chargement initial de l'application.
 - **NoPreloading** (par défaut): Aucun module n'est préchargé à moins que cela ne soit explicitement demandé.
 - **Stratégie Personnalisée**: Permet de créer une logique de préchargement personnalisée, basée sur des critères spécifiques comme la connexion réseau de l'utilisateur, les préférences de l'utilisateur, etc.

Utilisation des Guardes

Les guardes dans Angular sont des services que l'on peut utiliser pour contrôler la navigation dans une application. Ils offrent plusieurs fonctions, comme décider si un utilisateur peut naviguer vers ou depuis une route donnée. Les types de guardes les plus courants sont :

- **CanActivate** : Détermine si une route peut être activée.
- **CanLoad** : Détermine si un module associé à une route peut être chargé.

1. CanActivate

- `CanActivate` est utilisé pour protéger une route. Avant que l'utilisateur accède à une route, Angular vérifie si le garde retourne `true` (accès autorisé) ou `false` (accès refusé). C'est utile pour des scénarios comme la vérification de l'authentification d'un utilisateur avant de lui permettre d'accéder à une partie de l'application réservée aux membres.

2. CanLoad

- `CanLoad` est similaire à `CanActivate`, mais est utilisé pour empêcher le chargement paresseux (lazy loading) d'un module si l'utilisateur ne remplit pas certaines conditions. Cela peut réduire la quantité de code téléchargée par des utilisateurs non autorisés, améliorant la sécurité et les performances de l'application.

Utilisation des Résolveurs

Les résolveurs permettent à une application Angular de précharger des données avant de finaliser la navigation vers une route. Cela signifie que la vue associée à la route ne s'affiche que lorsque toutes les données nécessaires sont chargées, ce qui peut améliorer l'expérience utilisateur en évitant d'afficher des vues incomplètes ou en attente de chargement de données.

1. **Création du Service Résolveur** : créez un service qui implémente l'interface `Resolve`. Ce service doit définir une méthode `resolve` qui récupère les données nécessaires.
2. **Configuration de la Route** : Associez ensuite ce résolveur à la route concernée dans le module de routage. Vous spécifiez dans la configuration de la route quel résolveur utiliser et quelles données il doit charger.

Eslint

ESLint est un outil d'analyse statique pour JavaScript et TypeScript qui contribue directement à l'amélioration de la **qualité du code**.

1. Uniformisation du style de code

ESLint impose des règles cohérentes dans tout le projet, garantissant que tous les développeurs suivent les mêmes conventions de codage. Cela inclut des aspects comme :

- **Indentation** : Assurer que tout le code est correctement indenté, facilitant ainsi la lisibilité.
- **Utilisation cohérente des guillemets** : Imposer des guillemets simples ou doubles pour les chaînes de caractères.
- **Espaces et retours à la ligne** : Imposer des règles pour les espaces entre les blocs, les retours à la ligne, etc.

2. Détection des erreurs potentielles

ESLint peut identifier des erreurs courantes avant même que le code ne soit exécuté. Par exemple :

- **Variables non utilisées** : ESLint signale les variables déclarées mais non utilisées, ce qui aide à nettoyer le code inutile.
- **Fonctions non appelées** : Détecte les fonctions qui ne sont jamais utilisées, réduisant ainsi le code mort.
- **Déclaration non définie** : Alerte sur les variables ou fonctions appelées sans avoir été définies auparavant.

Eslint

3. Optimisation des performances

Certaines règles ESLint aident à identifier les parties de code qui pourraient être optimisées. Par exemple :

- **Eviter l'utilisation de `var`** : Encourage l'utilisation de `let` et `const`, ce qui améliore la gestion des variables en fonction de leur portée.
- **Code inutile** : ESLint détecte les conditions, boucles ou fonctions qui ne sont jamais atteintes ou nécessaires.

4. Réduction de la complexité cyclomatique

ESLint peut limiter la complexité cyclomatique, c'est-à-dire le nombre de chemins d'exécution possibles dans une fonction. Cela permet de :

- **Simplifier les fonctions** : Encourage à diviser les fonctions trop longues en plusieurs petites fonctions.
- **Réduire les conditions imbriquées** : Encourage à simplifier les conditions et les boucles.

Eslint

5. Meilleures pratiques de sécurité

Certaines règles d'ESLint mettent en avant les bonnes pratiques pour éviter des vulnérabilités de sécurité courantes :

- **Éviter l'utilisation de `eval()`** : `eval()` peut exécuter du code malveillant et est souvent une source de failles de sécurité.
- **Validation des entrées** : ESLint peut signaler les endroits où les entrées des utilisateurs ne sont pas correctement validées.

6. Amélioration de la maintenabilité

En détectant les redondances, les noms de variables peu explicites, ou les fonctions trop complexes, ESLint incite les développeurs à :

- **Renommer les variables ou fonctions** : Encourage à utiliser des noms plus explicites et plus significatifs.
- **Simplifier le code** : Incite à réduire la duplication de code.

7. Support des équipes et des projets à grande échelle

Dans une équipe de développement, chaque développeur a ses propres habitudes de codage. ESLint permet de :

- **Normaliser les contributions** : Tous les membres de l'équipe suivent les mêmes règles, ce qui facilite la relecture du code.
- **Améliorer l'intégration continue** : En intégrant ESLint dans un pipeline CI/CD, vous pouvez bloquer le déploiement si des erreurs critiques sont détectées.

AOP

L'**Aspect-Oriented Programming (AOP)** est une approche de programmation qui vise à augmenter la modularité en permettant la séparation des préoccupations transversales. Les préoccupations transversales sont des aspects du code qui coupent plusieurs couches de votre application, comme la gestion des logs, la sécurité, la gestion des transactions, etc. Au lieu d'avoir ces préoccupations mélangées avec le code métier principal, AOP permet de les isoler dans des "aspects" qui peuvent être appliqués de manière transversale.

AOP

Implémentation de l'AOP dans Angular

Bien qu'Angular ne supporte pas directement l'AOP de manière native comme Spring en Java, vous pouvez implémenter des concepts similaires en utilisant des **decorators** et des services. Voici une méthode courante pour imiter l'AOP en Angular :

1. **Créer un Decorator** : Un decorator peut être utilisé pour appliquer des comportements transversaux sur des méthodes spécifiques.
2. **Utiliser les Interceptors HTTP** : Angular fournit des interceptors qui permettent de gérer des préoccupations transversales liées aux requêtes HTTP comme la gestion des erreurs, la journalisation, etc.
3. **Utiliser des Services** : Créer des services Angular pour des fonctionnalités transversales comme le logging ou la gestion des exceptions et les injecter là où ils sont nécessaires.

AOP

Exemple d'implémentation avec un Decorator

```
function LogExecutionTime(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const originalMethod = descriptor.value;

    descriptor.value = function (...args: any[]) {
        console.log(`Execution started: ${propertyKey}`);
        const start = performance.now();

        const result = originalMethod.apply(this, args);

        const end = performance.now();
        console.log(`Execution ended: ${propertyKey}. Time taken: ${end - start}ms`);
        return result;
    };

    return descriptor;
}

class ExampleService {
    @LogExecutionTime
    someMethod() {
        // Method logic
    }
}
```


AOP

Exemple d'utilisation d'un Interceptor HTTP

```
@Injectable()
export class LoggingInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    console.log(`Request made to: ${req.url}`);

    return next.handle(req).pipe(
      tap(event => {
        if (event instanceof HttpResponse) {
          console.log(`Response received from: ${req.url}`);
        }
      })
    );
  }
}
```

AOP

Importance de l'AOP dans le Clean Code

L'AOP aide à maintenir le **Clean Code** en séparant les préoccupations transversales du code métier principal. Voici quelques avantages :

1. **Modularité améliorée** : Les fonctionnalités transversales comme le logging, la sécurité ou la gestion des transactions peuvent être isolées dans des aspects, rendant le code métier plus lisible et plus maintenable.
2. **Réduction de la duplication** : Au lieu d'avoir du code de logging ou de gestion d'erreurs dispersé partout, vous pouvez centraliser ces préoccupations.
3. **Facilité de maintenance** : Il est plus facile de mettre à jour ou de changer une préoccupation transversale lorsqu'elle est centralisée dans un aspect plutôt que dispersée dans tout le code.
4. **Testabilité accrue** : Le code est plus facile à tester lorsqu'il n'est pas encombré par des préoccupations transversales.

Optimisations

1. Éviter les Subscriptions non gérées avec `takeUntil` ou `async pipe`

Les abonnements aux observables dans Angular, tels que les appels HTTP, doivent être gérés pour éviter les fuites de mémoire. Il est recommandé d'utiliser des techniques comme `takeUntil` pour détruire les abonnements lorsqu'un composant est détruit, ou l'`async pipe` dans les templates Angular pour gérer les abonnements automatiquement.

Optimisations

Exemple 1 : Utilisation de `takeUntil` pour nettoyer les abonnements

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subject } from 'rxjs';
import { takeUntil } from 'rxjs/operators';
import { MyService } from './my-service.service';

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
})
export class ExampleComponent implements OnInit, OnDestroy {
  private destroy$ = new Subject<void>(); // Sujet utilisé pour notifier la destruction du composant
  public data: any;

  constructor(private myService: MyService) {}

  ngOnInit(): void {
    // Abonnement à l'observable avec nettoyage via takeUntil
    this.myService.getData()
      .pipe(takeUntil(this.destroy$))
      .subscribe(data => {
        this.data = data;
      });
  }

  ngOnDestroy(): void {
    // Notification pour détruire les abonnements
    this.destroy$.next();
  }
}
```

Optimisations

Exemple 2 : Utilisation de l'`async` pipe dans le template

L'`async` pipe permet de gérer automatiquement les abonnements dans le template, sans avoir besoin de les nettoyer manuellement.

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
import { MyService } from './my-service.service';

@Component({
  selector: 'app-example',
  template: `
    <div *ngIf="data$ | async as data">
      {{ data }}
    </div>
  `,
})
export class ExampleComponent {
  public data$: Observable<any>;

  constructor(private myService: MyService) {
    // On assigne directement l'observable sans avoir besoin de s'abonner
    this.data$ = this.myService.getData();
  }
}
```

Optimisations

2. Utiliser la Stratégie de Détection des Changements `OnPush`

La stratégie de détection des changements par défaut d'Angular est assez gourmande, car elle vérifie constamment les changements dans le DOM. L'utilisation de la stratégie `OnPush` permet d'optimiser les performances en ne déclenchant la détection de changements que lorsque les **entrées** (`@Input`) du composant changent ou lorsqu'un événement asynchrone (comme un observable) est émis.

Optimisations

Exemple d'utilisation de OnPush

```
import { Component, ChangeDetectionStrategy, Input } from '@angular/core';

@Component({
  selector: 'app-onpush-example',
  template: `
    <div>
      <h2>{{ data }}</h2>
    </div>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class OnPushExampleComponent {
  @Input() data: string; // Seule cette entrée déclenchera la détection de changements
}
```

Rappel des bests practices

1. Meilleures pratiques ESLint avec Angular

Utiliser ESLint et des plugins Angular spécifiques

- Installe le plugin ESLint pour Angular (`@angular-eslint/eslint-plugin`).
- Configure des règles spécifiques pour Angular afin d'éviter les erreurs courantes et d'améliorer la lisibilité du code.

Règles communes

- **Pas de variables inutilisées** : Utiliser la règle `no-unused-vars`.
- **Pas d'`any`** : Eviter l'utilisation du type `any`, toujours préférer des types explicites.
- **CamelCase** : Adopter le nommage camelCase pour les variables et les méthodes.
- **Limiter la complexité** : Réduire les if/else imbriqués, limiter la profondeur des fonctions.
- **Respecter les conventions TypeScript/ES6** : Utiliser `let` et `const` correctement pour une bonne gestion des variables.

Rappel des bests practices

Pratiques Angular

- **Externalisation de la logique métier** : Minimiser la logique dans les composants, déplacer la logique dans des services.
- **Nettoyage des abonnements RxJS** : Toujours se désabonner des observables en utilisant `takeUntil` ou l'`async` pipe.
- **Stratégie OnPush** : Utiliser `ChangeDetectionStrategy.OnPush` pour les composants affichant des données statiques ou peu changeantes pour optimiser les performances.

Rappel des bests practices

3. Pratiques de tests et couverture

Viser une couverture de test élevée

- Utiliser `ng test --code-coverage` pour suivre la couverture des tests unitaires.
- Objectif : 80-90% de couverture minimale.

Tests unitaires, d'intégration, et E2E

- **Tests unitaires** : Testez les composants, services et pipes individuellement en moquant les dépendances.
- **Tests d'intégration** : Testez les interactions entre les composants ou les services.
- **Tests de bout en bout (E2E)** : Utiliser des outils comme **Cypress** ou **Protractor** pour tester les flux utilisateur réels.

Rappel des bests practices

Meilleures pratiques pour les tests Angular

- Mock les dépendances avec `TestBed`.
- Utiliser `HttpTestingController` pour simuler des appels HTTP dans les services.
- Tester les événements `@Output`, les méthodes publiques des composants et la logique interne des services.

5. Test-Driven Development (TDD)

Le développement piloté par les tests (TDD) garantit que chaque fonction de l'application est bien couverte dès le début.

Cycle TDD

1. **Écrire un test qui échoue** : Écrire un test avant d'implémenter une nouvelle fonctionnalité.
2. **Écrire le code minimal** pour passer le test.
3. **Réfacter le code** pour l'améliorer tout en passant les tests.

Avantages du TDD

- Amélioration de la qualité du code avec des tests couvrant toutes les fonctionnalités.
- Réduction des bugs et des régressions dans le code.

Rappel des bests practices

6. Autres bonnes pratiques Angular

Architecture modulaire

- Découper l'application en modules fonctionnels, un module par fonctionnalité, et utiliser le lazy loading pour charger les modules uniquement quand nécessaire.

Utilisation appropriée des services

- **Services partagés** : Utiliser les services pour partager la logique métier entre plusieurs composants.
- **Services scoped** : Limiter la portée des services à des modules spécifiques si leur utilisation est limitée.

Rappel des bests practices

Gérer les formulaires

- Préférer l'utilisation des **Reactive Forms** pour les formulaires complexes, car ils sont plus faciles à tester et à maintenir que les **Template-driven Forms**.

Nommage et structure du projet

- Respecter les conventions de nommage des fichiers Angular :
 - **composant** : `my-component.component.ts`
 - **service** : `my-service.service.ts`
 - **module** : `my-module.module.ts`
- Organiser les fichiers en fonction des fonctionnalités et non des types de fichiers (par exemple, tous les fichiers relatifs à une fonctionnalité dans un même dossier).

Gestion des erreurs

- Centraliser la gestion des erreurs HTTP avec un Intercepteur
- Gestion des erreurs au niveau du composant