Les Grands Principes de Dev

Les Grands Principes de Dev regroupe les fondamentaux qui permettent d'écrire du code clair, structuré et fiable dans n'importe quel langage.

On y aborde les principes essentiels du développement : simplicité, modularité, architecture, gestion de la concurrence et fiabilité des données.

Chaque chapitre donne une vue d'ensemble compréhensible, avec des exemples concrets en JavaScript ou en MySQL, pour comprendre non seulement le "quoi", mais surtout le "pourquoi".

L'objectif n'est pas de faire du code parfait, mais du code maîtrisé, évolutif et robuste face à la complexité réelle des projets.

1 ~ Simplicité du code (KISS)

? En un mot

KISS consiste à résoudre un problème avec la solution la plus simple possible.

? Ce que KISS est ?

Il garantit en pratique :

- □□ Un code facile à lire
- □□ Moins de bugs liés à la complexité
- □□ Une maintenance plus simple

? Ce que KISS n'est pas ?

- Du code bâclé
- L'absence de structure
- Ignorer les bonnes pratiques

? Mauvais

```
function getActiveUsers(users) {
  return users.filter(u => u.active === true).map(u => ({ id: u.id, name: u.name }));
}
```

? Bon

```
function getActiveUsers(users) {
  const activeUsers = [];
  for (const user of users) {
```

```
    if (user.active) {
      activeUsers.push({ id: user.id, name: user.name });
    }
  }
  return activeUsers;
}
```

? Pourquoi ?

- Moins de complexité cognitive
- Plus simple à debugger

2 ~ Éviter la duplication (DRY)

? En un mot

DRY consiste à éviter de répéter la même logique à plusieurs endroits.

? Ce que DRY est ?

Il garantit en pratique :

- □□ Une seule source de vérité
- □□ Moins d'erreurs lors des modifications
- □□ Un code cohérent

? Ce que DRY n'est pas ?

- Créer des abstractions trop tôt
- Factoriser sans raison

? Mauvais

```
function calculatePriceWithTax(price) {
  return price * 1.21;
}

function calculateInvoiceTotal(price) {
  return price * 1.21;
}
```

? Bon

```
function applyVAT(price) {
  return price * 1.21;
}
```

? Pourquoi ?

- Une modification du taux → un seul endroit à changer

3 ~ Structurer le code proprement (SOLID)

? En un mot

Solid est un outil pour maîtriser la complexité croissante.

? Ce que SOLID est ?

Il facilite:

- □□ Modularité
- □□ Testabilité
- □□ Extensibilité
- □□ Réduction de dette technique
- □□ Architecture durable
- □□ Travail en équipe

? Ce que SOLID n'est pas ?

- Une obligation dogmatique
- Un truc à appliquer partout
- Un truc adapté à tout (micro-scripts inutiles)

? Les 5 piliers de SOLID

? S - Single Responsibility Principle

Une classe ou un module ne doit avoir qu’une seule responsabilité métier.

? Mauvais

```
class UserService {
  createUser(data) { /* ... */ }
  sendWelcomeEmail(user) { /* ... */ }
  logUserCreation(user) { /* ... */ }
}
```

? Bon

```
class UserService {
  createUser(data) { /* ... */ }
}

class EmailService {
  sendWelcomeEmail(user) { /* ... */ }
}

class Logger {
  log(message) { /* ... */ }
}
```

? Pourquoi?

- Tests plus simple
- Evolution indépendante
- Moins d’effets de bord

? O - Open Closed Principle (OCP)

Ouvert à l’extension, fermé à la modification. On dois pouvoir ajouter un comportement sans modifier le code existant.

? Mauvais

```
function calculateDiscount(user) {
  if (user.type === "premium") return 0.2;
  if (user.type === "vip") return 0.3;
}
```

? Bon

```
class DiscountStrategy {
  getDiscount() {
    return 0;
  }
}

class PremiumDiscount extends DiscountStrategy {
  getDiscount() {
    return 0.2;
  }
}

class VipDiscount extends DiscountStrategy {
  getDiscount() {
    return 0.3;
  }
}
```

? Pourquoi?

- Scalabilité
- Architecture plugin
- Systèmes extensibles

? L - Liskov Substitution Principle (LSP)

Un sous-type doit pouvoir remplacer son type parent sans casser le comportement attendu. Si tu hérites, tu dois respecter le contrat.

? Mauvais

```
class Bird {
  fly() {}
}

class Penguin extends Bird {
  fly() {
    throw new Error("I can't fly");
  }
}
```

? Bon

```
class Bird {}

class FlyingBird extends Bird {
  fly() {}
}

class Penguin extends Bird {}
```

? Pourquoi?

- Hiérarchies cohérentes
- Pas de surprises

? I - Interface Segregation Principle (ISP)

Ne force pas une classe à implémenter ce qu’elle n’utilise pas.

? Mauvais

```
class Employee {
  work() {}
  eat() {}
  sleep() {}
}

class Robot extends Employee {
  work() {
    console.log("Working...");
  }

  eat() {
    throw new Error("Robot does not eat");
  }

  sleep() {
    throw new Error("Robot does not sleep");
  }
}
```

? Bon

```
class Workable {
  work() {}
}

class Human extends Workable {
  eat() {
    console.log("Eating...");
  }

  sleep() {
    console.log("Sleeping...");
  }
```

```
}

class Robot extends Workable {
  work() {
    console.log("Working...");
  }
}
```

? Pourquoi?

- Architecture plus flexible

? D - Dependency Inversion Principle (DIP)

Dépendre d’abstractions, pas de concrétions (implémentation concrète).

? Mauvais

```
class UserService {
  constructor() {
    this.database = new MySQLDatabase();
  }
}
```

? Bon

```
class UserService {
  constructor(database) {
    this.database = database;
  }
}

const db = new MySQLDatabase();
const service = new UserService(db);
```

? Pourquoi?

- Interchangeable (Ici, passer à Postgres, Mongo, une API)
- Scalable

4 ~ Maîtriser le couplage et la cohésion

? En un mot

Un bon design vise un couplage faible et une cohésion forte.

? Ce que cela garantit ?

- □□ Modules indépendants
- □□ Code évolutif
- □□ Tests plus simples

? Ce que cela n'est pas ?

- Multiplier les fichiers inutilement
- Complexifier l'architecture

? Mauvais (couplage fort)

```
class PaymentService {
  constructor() {
    this.gateway = new StripeGateway();
  }
}
```

? Bon (couplage faible)

```
class PaymentService {
  constructor(gateway) {
```

```
    this.gateway = gateway;
  }
}
```

? Pourquoi ?

- On peut remplacer Stripe facilement
- Meilleure flexibilité

5 ~ Séparer les responsabilités (Separation of Concerns)

? En un mot

Séparer les responsabilités évite qu'un module fasse tout.

? Ce que cela garantit ?

- □□ Code clair
- □□ Maintenance facilitée
- □□ Collaboration plus simple

? Ce que cela n'est pas ?

- Ajouter des couches inutiles
- Sur-architecturer

? Mauvais

```
app.post('/order', async (req, res) => {
  // validation
  // calcul prix
  // sauvegarde DB
  // envoi email
});
```

? Bon

```
app.post('/order', validateOrder, orderController);

function orderController(req, res) {
  orderService.create(req.body);
}
```

? Pourquoi ?

- Chaque couche a un rôle précis

6 ~ Structurer l’architecture d’une application (MVC, Clean Architecture)

? En un mot

Une architecture claire organise le code en couches distinctes.

? Ce que cela garantit ?

- □□ Organisation logique
- □□ Testabilité
- □□ Scalabilité

? Ce que cela n'est pas ?

- Une structure rigide obligatoire
- Une complexité systématique

Exemple simplifié MVC

```
// Controller
function createUser(req, res) {
  userService.create(req.body);
}

// Service
class UserService {
  create(data) {
    userRepository.save(data);
```

```
  }
}

// Repository
class UserRepository {
  save(data) {
    // accès MySQL
  }
}
```

? Pourquoi ?

- Séparation claire entre HTTP, logique métier et base

7 ~ Garantir des opérations sûres (Idempotence)

? En un mot

Une opération idempotente produit le même résultat si elle est exécutée plusieurs fois.

? Ce que cela garantit ?

- □□ Sécurité contre les doubles requêtes
- □□ Protection contre les doubles paiements

? Ce que cela n'est pas ?

- Une protection automatique contre tous les bugs
- Une solution miracle côté base

Exemple HTTP

```
app.put('/user/1', updateUser);
```

Appeler plusieurs fois la même requête garde le même état final.

? Pourquoi ?

- Important en cas de retry réseau

8 ~ Gérer la concurrence

? En un mot

Gérer la concurrence évite les incohérences quand plusieurs utilisateurs agissent en même temps.

? Ce que cela garantit ?

- □□♂□□♀ Pas de double dépense
- □□ Pas de stock négatif

? Ce que cela n'est pas ?

- Une optimisation de performance
- Une simple question de logique applicative

Exemple MySQL

```
START TRANSACTION;

SELECT stock FROM products WHERE id = 10 FOR UPDATE;

UPDATE products SET stock = stock - 1 WHERE id = 10;

COMMIT;
```

? Pourquoi ?

- La ligne est verrouillée pendant la transaction

9 ~ Transactions et verrous en base de données

? En un mot

Les transactions et verrous assurent l'intégrité des opérations en base de données.

? Ce que cela garantit ?

- □□ Cohérence des données
- □□ Pas d'état intermédiaire visible

? Ce que cela n'est pas ?

- Une solution magique contre les erreurs métier
- Une garantie de performance maximale

Exemple MySQL

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 50 WHERE id = 1;
UPDATE accounts SET balance = balance + 50 WHERE id = 2;

COMMIT;
```

? Pourquoi ?

- Les deux opérations sont validées ensemble

10 ~ Fiabilité des données (ACID)

? En un mot

ACID est un ensemble de garanties qui rendent tes transactions dans la base de donnée fiables, prévisibles et sûres en production.

? Ce que ACID est ?

Il garantit en pratique :

- □□ Pas d’opérations “à moitié faites”
- □□ Des données toujours valides après une transaction
- □□♂ Un comportement correct même avec plusieurs utilisateurs en même temps
- □□ Des modifications conservées même après un crash serveur

? Ce que ACID n'est pas ?

- Une garantie de performance maximale
- Une protection contre une mauvaise logique métier
- Une solution magique contre tous les problèmes de concurrence
- Un modèle obligatoire pour tous les cas (analytics, cache, etc.)

? Les 4 piliers d’ACID

? A - Atomicité

? Principe

Une transaction est indivisible : si une étape échoue, **tout est annulé**.

? Cas concret

Transfert d’argent entre deux comptes.

On veut :

- Débiter A
- Créditer B
- Ajouter une ligne dans l’historique

Si une seule étape échoue → rien ne doit être appliqué.

? Exemple MySQL

```
START TRANSACTION;

UPDATE accounts
SET balance = balance - 100
WHERE id = 1;

UPDATE accounts
SET balance = balance + 100
WHERE id = 2;

INSERT INTO ledger (from_account_id, to_account_id, amount)
VALUES (1, 2, 100);

COMMIT;
```

? Si problème :

```
ROLLBACK;
```

? Explication

Tant que `COMMIT` n’est pas exécuté :

- Les modifications ne sont pas définitives
- Si une requête échoue (ex: contrainte, crash, validation métier)
- `ROLLBACK` annule TOUT

Résultat : Pas de débit sans crédit. Pas d’état incohérent.

? C - Cohérence

? Principe

Après `COMMIT`, la base respecte toutes ses contraintes.

? Cas concret

Empêcher :

- Un solde négatif
- Une référence vers un compte inexistant

? Définition des contraintes

```
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  balance DECIMAL(10,2) NOT NULL,
  CHECK (balance >= 0)
) ENGINE=InnoDB;

CREATE TABLE ledger (
  id INT AUTO_INCREMENT PRIMARY KEY,
```

```
  from_account_id INT,
  to_account_id INT,
  amount DECIMAL(10,2) CHECK (amount > 0),
  FOREIGN KEY (from_account_id) REFERENCES accounts(id),
  FOREIGN KEY (to_account_id) REFERENCES accounts(id)
) ENGINE=InnoDB;
```

? Tentative invalide

```
START TRANSACTION;

INSERT INTO ledger (from_account_id, to_account_id, amount)
VALUES (1, 9999, 50);

COMMIT;
```

? Explication

- `9999` n’existe pas
- La contrainte FK bloque l’opération
- La transaction échoue

La base reste valide.

ACID empêche un état incohérent d’être validé.

? I - Isolation

? Principe

Les transactions concurrentes ne doivent pas créer d’anomalies.

? Cas concret

Deux utilisateurs tentent de dépenser le même solde en même temps.

? Exemple avec verrouillage

Transaction A :

```
START TRANSACTION;

SELECT balance
FROM accounts
WHERE id = 1
FOR UPDATE;

UPDATE accounts
SET balance = balance - 50
WHERE id = 1;

COMMIT;
```

Transaction B (en parallèle) :

```
START TRANSACTION;

SELECT balance
FROM accounts
WHERE id = 1
FOR UPDATE;
```

? Explication

- `FOR UPDATE` verrouille la ligne
- Transaction B attend la fin de A
- Pas de double débit simultané

Isolation = contrôle des conflits concurrents.

? D - Durabilité

? Principe

Une fois `COMMIT`, les données survivent à un crash.

? Cas concret

Paiement validé → serveur plante immédiatement après.

? Exemple

```
START TRANSACTION;


UPDATE accounts
SET balance = balance + 200
WHERE id = 2;


COMMIT;
```

Si le serveur tombe juste après :

```
SELECT balance FROM accounts WHERE id = 2;
```

La modification est toujours présente.

? Explication

Ici, sous InnoDB des logs (redo log) sont utilisés automatiquement pour garantir que :

- Une transaction validée sera restaurée après redémarrage
- L’état reste cohérent même après crash

? En pratique

ACID est crucial pour :

- □□ Paiements
- □□ Gestion de stock
- □□ Droits utilisateurs
- □□ Facturation
- □□ Systèmes financiers

Il garantit que ta base ne devient pas un chaos concurrent.