AIX ynov CAMPUS

CAPSULE

INTRODUCTION AUX BONNES PRATIQUES & DÉBOGAGE

Par Cyril RODRIGUES

LES OBJECTIFS

Cette capsule à pour objectif de vous familiariser avec les bonnes pratique de développement que cela soit durant les projet en solitaire mais également collaboratif. À l'issue de cette capsule, vous serez capable de :

- Expliquer pourquoi les bonnes pratiques comptent.
- Reconnaître les anti-patterns courants et leurs impacts.
- Appliquer des principes généraux lors de vos sessions de développement.
- Mettre en place des conventions d’équipe pour garder un code homogène.
- Déboguer votre code source grâce à une checklist en 7 étapes.

INTRODUCTION

LES BONNES PRATIQUES, C'EST QUOI ?!

Les bonnes pratiques sont un ensemble de règles simples et partagées par toute l'équipe au sein d'un projet. Elles posent un cadre commun qui rend le code lisible, fiable et durable, afin d'aller vite sans sacrifier la qualité.

Concrètement, elles facilitent la compréhension, la collaboration et l'évolution du code sur la durée...

Ne pas appliquer de bonnes pratiques, c'est accepter...

- une dette technique qui explose (évolutions lentes, régressions) ;
- une qualité aléatoire (bugs imprévisibles, incidents en prod) ;
- des coûts plus élevés à long terme (temps, image, stress).

LES BÉNÉFICES
DES BONNES PRATIQUES...

Adopter des bonnes pratiques apporte un cadre commun qui rend le code plus lisible, modulaire et testable, ce qui réduit les bugs et accélère les évolutions dans le temps. L'équipe parle le même langage (conventions, revues, documentation), la livraison devient plus fréquente et plus sûre, et les risques sécurité diminuent grâce à une meilleure gestion des erreurs, des dépendances et des secrets.

À retenir :

- Une base saine : vitesse durable (livrer souvent sans casser).
- Conventions + tests + CI : moins d'incidents et moins de stress.
- Code clair et structuré : collaboration facilitée et coûts réduits.

Bonnes pratiques = gains de temps sur le long terme

LES PRINCIPES GÉNÉRAUX

C'EST QUOI,

LES PRINCIPES GÉNÉRAUX !?

Les principes généraux guident des décisions quotidiennes simples et cohérentes. Ils aident à éviter la complexité inutile, réduire les bugs et rendre le code prévisible, testable et évolutif. L'objectif n'est pas d'être dogmatique, mais de disposer d'un cadre commun qui accélère l'équipe sur la durée.

Ces principes sont indépendants de tout langage : ce sont des concepts applicables partout (web, data, mobile, embarqué).

KEEP IT SIMPLE, STUPID (KISS)

KISS invite à choisir la solution la plus simple qui résout le problème actuel. La complexité a un coût dans la mesure où elle ralentit la lecture, augmente le risque de bugs mais rend également le débogage plus difficile.

```
// Trop complexe
function isEven(n):
    return ((n << 1) >> 1) == n

// Simple et clair
function isEven(n):
    return n % 2 == 0
```

Exemple avec la vérification d'un nombre paire

DON'T REPEAT YOURSELF, (DRY)

DRY consiste à supprimer la duplication et factoriser les règles métier pour avoir une source de vérité. Dans l'objectif d'éviter incohérences et des changements plus rapides. Factoriser le code permet d'avoir un seul point de changement pour garder une cohérences et éviter les erreurs.

```
// Duplication (risque d'incohérence)
priceWithTaxFR = price * 1.20
priceWithTaxDE = price * 1.19

// DRY: une fonction réutilisable
function addTax(price, rate):
    return price * (1 + rate)

priceWithTaxFR = addTax(price, 0.20)
priceWithTaxDE = addTax(price, 0.19)
```

Exemple avec le calcul d'une taxe

SÉPARATION
DES RESPONSABILITÉS

La séparation des responsabilités repose sur un principe simple, chaque fonction ou module doit assumer une responsabilité claire. On divise donc le code pour avoir une responsabilité par fonction. Par exemple : validation des données, règles métier, accès aux données. Pour obtenir un code plus lisible, testable et remplaçable par briques.

```
// Mélangé (dur à tester)
function processOrder(input):
    data = readFile(input)
    parsed = parseJSON(data)
    if parsed.total <= 0: return error("invalid")
    // règles métier + I/O tout entremêlé
    saveToDB(parsed)
    return ok()
```

VS

```
// Découpage par responsabilités
function processOrder(input):
    data = readInput(input)
    order = parseOrder(data)
    if not isValid(order): return error("invalid")
    result = applyBusinessRules(order)
    return persist(result)

function readInput(path): ...
function parseOrder(data): ...
function isValid(order): ...
function applyBusinessRules(order): ...
function persist(result): ...
```

YOU AREN'T GONNA NEED IT, (YAGNI)

YAGNI rappelle de ne pas développer à l'avance des fonctionnalités hypothétiques. Le besoin réel clarifie la solution alors que l'anticipation crée de la dette et de l'inutile. On code au moment du besoin !

GESTION D'ERREURS & OBSERVABILITÉ

Rendre les erreurs visibles (retours explicites, exceptions traitées) et diagnosticables (logs, métriques) évite les bugs silencieux. Des messages clairs et des niveaux de log adaptés accélèrent le débogage. Attention aux données, on ne logue jamais de données sensibles (mots de passe, tokens…).

```
// Fragile : plante silencieusement
user = fetchUser(id)
print(user.name)   // crash si fetch échoue

// Explicite + logs utiles
result = fetchUser(id)
if not result.success:
    logError("fetchUser failed", { "id": id, "err": result.error })
    return fallbackUser()

logInfo("user fetched", { "id": id })
print(result.data.name)
```

À retenir :

- Toujours gérer les erreurs (pas d'erreurs avalées).
- Logs : message + contexte utile + bon niveau.
- Aucune donnée sensible dans les erreurs ou les logs.

RÉSUMER...

Des règles simples guident un code clair, fiable et évolutif :

- KISS pour rester simple ; DRY pour éviter la duplication ; YAGNI pour ne pas anticiper inutilement.
- Séparation des responsabilités : une fonction/module = un rôle.
- Gestion d’erreurs & Observabilité : erreurs traitées, logs/métriques pour diagnostiquer vite.

LES CONVENTIONS

C'EST QUOI,
LES CONVENTIONS ?

Des conventions partagées donnent un langage commun : le code devient prévisible et cohérent, quelle que soit la personne qui l'écrit. Elles sont décidées en début de projet comme une convention d'équipe, puis appliquées au quotidien.

Elles couvrent le nommage, la structure du dépôt, le formatage, le workflow Git, les pull requests, les logs, les tests et la sécurité.

Il est important de notre deux points lors de l'utilisation de conventions :

- Des conventions simples pour avoir moins d'ambiguïtés.
- Écrites et visibles dans le repo par tous (Par exemple dans le readme.md)

NOMMAGE

La convention de nommage est l'une des plus importantes, elle garde le code d'un projet homogène et prévisible. Cela passe par des noms clairs, le choix d'une langue unique (français ou anglais) et d'un style (camelCase/snake_case), puis le respect de ces choix. Mieux vaut la clarté et la cohérence que la concision : on évite ainsi les malentendus et les interprétations.

On peut définir une convention de nommage pour :

- Variables & fonctions : noms explicites ; fonctions = verbe (ex. calculateTotal), booléens avec is/has/can.
- Classes/Types : PascalCase (ex. PaymentService, UserProfile).
- Constantes : UPPER_SNAKE_CASE (ex. MAX_RETRY, DEFAULT_TIMEOUT_MS).
- Fichiers & dossiers : style unique (ex. user_profile/, order-service/) et noms métier plutôt que techniques.
- Unités & pluriels : inclure l'unité (timeoutMs, priceEUR) et soigner singulier/pluriel (users, userId).

FORMATAGE & LINT

Uniformiser le style avec un formatter (formatage automatique) et un linter (règles de qualité). Exécuter ces outils en local et en CI pour des retours constants et automatiques. Le but : un code cohérent, des diffs lisibles et des revues qui se concentrent sur le fond (logique, design) plutôt que sur la forme.

- Règles d'équipe : longueur de ligne, indentations, guillemets, ordre des imports, interdits (TODO en prod, console.log, etc.).
- Hooks pre-commit : format et lint avant chaque commit puis la CI bloque si non conforme.
- Outils et config partagée dans le repo (fichiers de config du linter/formatter).

UTILISATION DE LOGS

Mettre en place des logs utiles et sobres aide autant au débogage qu'à la surveillance en production.
L'objectif est de rendre les événements observables, de diagnostiquer vite et de mesurer ce qui compte, sans noyer le signal.

Un bon log contient :

- Un niveau : DEBUG, INFO, WARN, ERROR, CRITICAL.
- Un message clair et concis, jamais de données sensibles (mots de passe, tokens).
- Une source du composant/service et timestamp normalisé.

TESTS

L’objectif n’est pas de tout tester mais de couvrir le risque là où il est. Écrire des tests unitaires pour le cœur métier (fonctions pures, règles), et des tests d’intégration pour les flux clés (vérifier la manipulation de la base de données). Important après chaque bug, ajouter un test de non-régression qui échouait avant et passe après le correctif.

Les élément importants :

- Couvrir le risque, pas viser un pourcentage magique.
- Unitaire pour la logique, intégration pour les flux.
- Chaque bug corrigé ajout test de non-régression ajouté.

SÉCURITÉ DE BASE

La sécurité se traite dès le départ et fait partie de la qualité de code. Valider les entrées, protéger les secrets (variables d'environnement / vault, jamais en dépôt), maintenir les dépendances à jour et appliquer le moindre privilège (comptes/roles/permissions minimales). Des limites (timeouts, rate-limit) et des logs non sensibles complètent le socle.

WORKFLOW GIT

Travailler par petites branches thématiques et ouvrir des PR courtes rend les changements plus faciles à relire, à tester et à livrer. Des PR ciblées limitent les conflits, accélèrent les retours et réduisent les risques de régression.

- Branches thématiques : feat/, fix/, chore/, refactor/ + nom explicite (feat/checkout-tax)
- Taille raisonnable : < 300–400 lignes modifiées si possible ; scinder sinon
- Description claire : contexte, ce qui change, comment tester (captures si utile)
- Commits propres : un changement logique par commit (rebase/squash si besoin)
- Demande de review ciblée : bons reviewers, attentes explicites

LE DÉBOGAGE

LE DÉBOGAGE

Le débogage efficace repose sur une méthode : rendre le bug reproductible, isoler la cause, mesurer l'effet d'un correctif et prévenir la réapparition. L'objectif n'est pas d'aller vite au hasard, mais de réduire l'incertitude pas à pas.

Il est important de retenir les éléments suivants :

- Une approche scientifique soit on part de l'hypothèse, correction puis enfin le test.
- Un seul paramètre change à la fois.

LE DÉBOGAGE

Un bon débogage n'est pas une série d'essais au hasard, mais une démarche structurée qui réduit l'incertitude à chaque étape. L'objectif : reproduire, isoler, observer, valider puis prévenir pour éviter que le problème ne revienne.

Les étapes pour résoudre un bug :

- Reproduire précisément l'erreur avec un ou plusieurs cas différents.
- Isoler la cause, l'objectif est de réduis le périmètre jusqu'au bloc à l'origine de l'erreur.
- Observer l'état (assertions, logs ciblés, métriques, traces / ID de corrélation).
- Formuler une hypothèse & la falsifier vite, l'objectif est de tester l'idée la plus probable.
- Corriger & prévenir : L'erreur est fixée puis vérifié pour empêche le retour du bug. L'objectif est éviter les régressions et effets de bord.

PRATIQUE

INTRODUCTION

Ce travail a pour but de mettre en pratique les principes et conventions vus en cours, sur la stack technique de votre choix (web, mobile, data, embarqué, etc.). Vous réaliserez un petit projet démontrant une implémentation soignée des éléments vues.

Objectifs du projet :

- Appliquer les principes généraux (KISS, DRY, YAGNI, séparation des responsabilités) à un cas concret.
- Définir et documenter des conventions d'équipe (nommage, messages de commit, PR).
- Automatiser la qualité avec un formatter et un linter
- Gérer les erreurs et observer l'application (retours explicites, logs utiles) sans données sensibles.
- Mettre en œuvre des bases de sécurité (validation d'entrées, gestion des secrets).
- Présenter une synthèse de recherche sur les outils/normes de votre stack et justifier vos choix.

SUJET (AU CHOIX)

Créez une petite application utile mais limitée sur la stack de votre choix. Quelques idées :

- Web
  - Checkout simplifié : calcul panier + taxes + remises + endpoints REST.
  - Application météo : recherche par ville, affichage des prévisions, gestion d'erreurs.
  - Application d'info : liste d'articles, filtres, favoris (en localhost).

- Logiciel
  - CLI : petit utilitaire (ex. conversion de fichiers, recherche/grep amélioré).
  - Mini logiciel desktop : viewer simple (fichiers, images, logs)

LA DEMANDE

Les attendus ci-dessous précisent exactement ce que vous devez livrer pour valider le travail :

1) Conventions & hygiène

- Nommage clair et cohérent (langue + style).
- Structure du dépôt lisible (README.md, src/, tests/, configs).
- Formatter et linter configurés.
- Workflow Git avec des petites branches, commits structuré, PR concises.

2) Principes de code

- KISS : solutions simples pas d'usine à gaz.
- DRY : factorisation pour obtenir une seule source de vérité.
- YAGNI : pas de features anticipées non utilisées.
- Séparation des responsabilités, des fonctions à rôle unique.

LA DEMANDE

Les attendus ci-dessous précisent exactement ce que vous devez livrer pour valider le travail :

3) Erreurs, logs, observabilité

- Chemins d'erreur explicites (retours/Exceptions traitées).
- Logs actionnables (niveau + message + contexte non sensible).

4) Débogage

- Produire un bug report et démontrer la méthode : repro, isolement, observation, hypothèse, fix, prévention.

LES LIVRABLES

Les livrables suivants décrivent ce que vous devez remettre pour évaluer votre travail de bout en bout :

- Code sur un repo public avec :
  - README.md (voir modèle)
  - Le code source du projet (avec une architecture propre)
  - Une PR exemplaire (ou screenshots) montrant la revue.
  - Bug report : Titre, étapes de repro, attendu vs observé, logs/trace, cause racine, correctif, test ajouté.
  - Note d'architecture : découpage en responsabilités, dépendances, choix KISS/DRY/YAGNI.

Date de rendu à voir sur Moodle

README.MD

```
# <Nom du projet>

## Objectif
Brève description métier (1-2 phrases).

## Démarrage
Prérequis, installation, variables d'environnement (sans secrets), lancement, tests.

## Structure
- src/<moduleA> : ...
- src/<moduleB> : ...
- tests/ : ...

## Conventions
Nommage (langue + style), formatter, linter, règles Git (branches, PR).

## Observabilité
Niveaux de logs utilisés, contexte logué, métriques (si présentes).

## Sécurité de base
Validation d'entrées, gestion secrets, dépendances.

## Limites & pistes
Ce qui manque (YAGNI), améliorations possibles.
```