Développement Mobile Multiplateforme

Pr. Mohamed BOUDCHICHE

École Nationale de l'Intelligence Artificielle et du Digital (ENIAD)
Université Mohammed Premier

m.boudchiche@edu.ump.ma

Année universitaire 2025–2026

GINF2 - IA2 - ROC2

Chapitre 1 : Introduction au Développement Multiplateforme

Pr. Mohamed BOUDCHICHE

École Nationale de l'Intelligence Artificielle et du Digital (ENIAD)
Université Mohammed Premier

m.boudchiche@edu.ump.ma

Année universitaire 2025–2026

Définition du développement multiplateforme

- Le **développement multiplateforme** consiste à créer une seule application capable de fonctionner sur plusieurs systèmes d'exploitation : **Android**, **iOS**, **Windows**, **Web**, voire **macOS** ou **Linux**.
- L'objectif est de **mutualiser le code** : écrire la logique et l'interface une seule fois, puis la déployer sur plusieurs environnements.
  → Cela permet de **réduire le coût**, le temps de développement et la charge de maintenance.
- L'application repose sur un **même cœur de code** (logique, interface, communication réseau, stockage local) mais peut nécessiter quelques **adaptations spécifiques** selon les plateformes (taille d'écran, boutons système, permissions, etc.).
- Cette approche s'oppose au développement **natif**, où chaque version (Android, iOS, Web) doit être écrite dans un langage différent — par exemple : Java/Kotlin pour Android, Swift pour iOS, JavaScript pour le Web.

Intérêt du développement multiplateforme

- **Code unique et réutilisable :** Une seule base de code permet de générer des applications pour plusieurs plateformes (**Android**, **iOS**, **Web**, **Desktop**).
  → Cela évite de dupliquer la logique métier et facilite la synchronisation entre les versions.

- **Maintenance simplifiée :** Les corrections de bugs et les nouvelles fonctionnalités sont effectuées une seule fois.
  → Les mises à jour se propagent automatiquement à toutes les plateformes.

- **Cohérence visuelle et expérience utilisateur uniforme :** Les frameworks multiplateformes (Flutter, Xamarin, React Native...) assurent un design homogène, garantissant la même ergonomie et le même comportement sur tous les supports.

Intérêt du développement multiplateforme

- **Réduction des coûts de développement :** Une seule équipe suffit pour toutes les plateformes.
  → Moins de développeurs spécialisés, donc un budget de production et de maintenance réduit.

- **Déploiement et mise sur le marché rapides :** Le développement parallèle permet de publier simultanément sur le *Play Store*, l'*App Store* et le Web.
  → Diminution du *time-to-market* et meilleure réactivité face au marché.

- **Rentabilité et compétitivité accrues :** Les entreprises peuvent tester, ajuster et déployer plus rapidement leurs produits, tout en atteignant un public plus large sans effort supplémentaire.

Cas d'usage du développement multiplateforme

- **Startups et PME :** Ces structures recherchent des solutions **rapides à développer** et **économiques**. Le multiplateforme leur permet de lancer une application sur Android, iOS et Web en un temps réduit.

  → Exemples :
  - Application de *livraison locale* ou de *réservation de services*.
  - Plateforme *e-learning* ou de *formation en ligne*.
  - Application de *transport multimodal* (ex. projet « Smart Route Maroc »).

- **Institutions publiques et universités :** Besoin de **services unifiés** accessibles depuis un téléphone ou un navigateur.

  → Exemples :
  - Plateforme d'*inscription en ligne* (ex. « ENIAD Admissions »).
  - Application de *gestion des stages et projets PFE*.
  - Portail étudiant avec *emplois du temps*, *résultats*, *notifications push*.

Cas d'usage du développement multiplateforme

- **Grands groupes et entreprises internationales :** Recherchent la **cohérence de l'expérience utilisateur** sur tous les supports et à l'échelle mondiale.

  → Exemples :
  - *Google Ads*, *Google Classroom* (Flutter).
  - *Alibaba*, *eBay Motors*, *Reflectly*.
  - Banques et assurances : applications clients homogènes sur Android et iOS.

- **Entreprises du e-commerce et du secteur bancaire :** Un besoin fort d'unifier le design et les fonctionnalités (authentification, paiement, support client).

  → Exemples : *Jumia*, *CIH Mobile*, *BMCE Direct*, *Maroc Telecom*.

Cas d'usage du développement multiplateforme

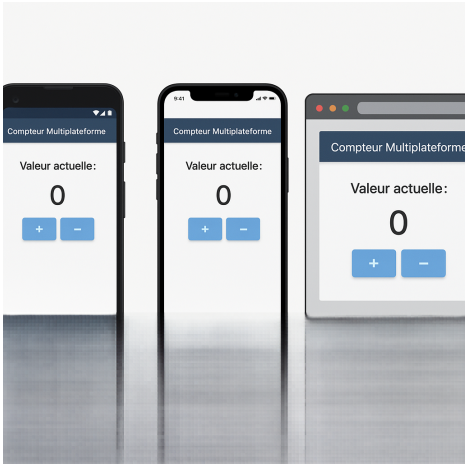

Illustration : une même application sur Android, iOS et Web

Les trois approches principales

1. **WebView / Hybride** – interface web encapsulée dans une application native.
2. **Code partagé natif** – logique commune, interface native.
3. **Rendu natif** – interface dessinée directement par le moteur graphique.

Approche 1 : WebView / Hybride

- Cette approche consiste à développer une application en utilisant des **technologies Web classiques** : **HTML**, **CSS** et **JavaScript**. L'application est ensuite embarquée dans un conteneur natif.

- L'interface utilisateur s'affiche dans une **WebView**, c'est-à-dire un mini-navigateur intégré dans l'application.
  → Cette WebView joue le rôle d'un moteur de rendu HTML/JS (comme Chrome ou Safari), mais à l'intérieur de l'app.

- Pour interagir avec les fonctionnalités du téléphone (**caméra**, **GPS**, **accéléromètre**, **fichiers**, etc.), on utilise des **plugins natifs** qui font le lien entre le code JavaScript et le système d'exploitation.

- **Exemples de frameworks :** *Apache Cordova*, *Ionic*, *PhoneGap*.
  → Ces outils emballent ton code web dans une application native.

Approche 2 : Code partagé natif

- Cette approche repose sur le principe de **partager la logique métier** de l'application (gestion des données, calculs, accès aux API, sécurité, authentification...) tout en conservant une **interface utilisateur native** propre à chaque plateforme.
- Le code métier est donc **mutualisé** entre Android, iOS (et parfois d'autres plateformes), tandis que chaque interface est développée avec les **composants graphiques natifs** du système.
- Cette approche offre un bon **équilibre entre performance et productivité** : l'application s'exécute en code natif, tout en évitant de dupliquer la logique.
- **Langages typiques : C#** avec le framework *.NET / Xamarin* ou **Kotlin Multiplatform (KMM)** pour le partage entre Android et iOS.
- **Exemples de frameworks :** *Xamarin.Forms*, *Xamarin.Android / iOS*, *Kotlin Multiplatform Mobile (KMM)*.

Approche 3 : Rendu natif

- Dans cette approche, l'application ne dépend plus des composants natifs du système (boutons, menus, listes, etc.). L'interface est entièrement **rendue par un moteur graphique intégré** (comme **Skia** ou **Impeller**).
- Le code source est écrit dans un langage unique (**Dart**, **JavaScript**, voire **C++**) puis **compilé en code machine natif** pour chaque plateforme cible (Android, iOS, Web, Desktop).
  $\rightarrow$ Cela assure des performances très proches du natif.
- L'application gère directement le rendu graphique à l'écran (textes, images, animations, transitions) sans passer par la WebView ni par les contrôles UI d'Android ou d'iOS.
- **Exemples :** *Flutter (Google)* — moteur **Skia**, rendu pixel-par-pixel. *React Native (Meta)* — traduit les composants React en éléments natifs via un bridge performant.

Présentation de Cordova

- **Apache Cordova** (anciennement *PhoneGap*) est un framework **open-source** développé à l'origine par la société canadienne **Nitobi**, puis acquis par **Adobe** en 2011.
- Le projet a ensuite été confié à la **fondation Apache Software Foundation**, qui le maintient encore aujourd'hui.

- Cordova est considéré comme le **pionnier du développement mobile multiplateforme**. Il a ouvert la voie aux frameworks hybrides modernes tels que *Ionic* et *Capacitor*.

- Il permet de créer des **applications hybrides** à partir de technologies Web classiques : **HTML** pour la structure, **CSS** pour le style, et **JavaScript** pour la logique applicative.

Présentation de Cordova

- Le code web est embarqué dans une application native via une **WebView**, c'est-à-dire un **navigateur intégré** à l'intérieur de l'application.
  → L'interface est rendue comme une page web, mais exécutée localement sur le téléphone.

- Cordova ajoute une **couche d'abstraction** entre le code JavaScript et les fonctionnalités du système d'exploitation. Cette couche est constituée de **plugins natifs**.

- Ces plugins agissent comme des **ponts ("bridges")** permettant au code JavaScript d'accéder au matériel du téléphone : *caméra, GPS, stockage, capteurs, notifications, etc.*

- Cette approche simplifie le développement, mais les performances dépendent du moteur de rendu de la WebView.

Historique et rôle de Cordova

- **2009 :** création de *PhoneGap* par la société canadienne **Nitobi**.
- **2011 :** rachat par **Adobe**, qui confie ensuite le projet à la **fondation Apache**.
- **2013–2016 :** émergence de frameworks construits sur Cordova (ex. *Ionic*).
- **Aujourd'hui :** Cordova reste une solution simple et stable pour les applications légères ou institutionnelles.
- Il a servi de **base technologique** aux solutions hybrides modernes.

Architecture de Cordova

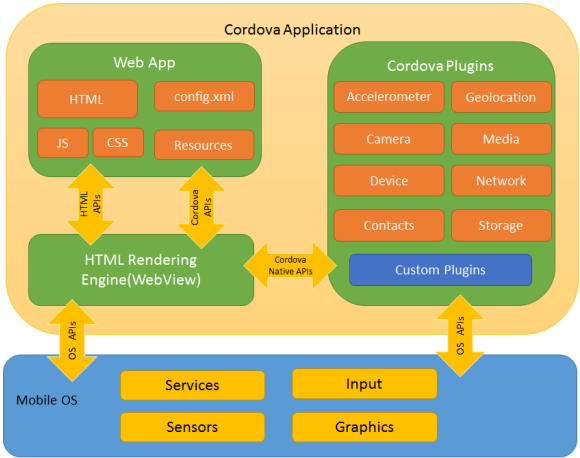

Schéma simplifié de l'architecture Cordova

Avantages de Cordova

- **Simplicité d'apprentissage :** Cordova utilise des langages web standards (**HTML, CSS, JavaScript**) — accessibles à tout développeur web.
- **Multiplateforme :** Un seul code source peut être déployé sur **Android**, **iOS**, et même **Windows**.
  $\rightarrow$ Cela permet de gagner du temps sur la maintenance et la publication.
- **Large écosystème de plugins :** Des centaines de plugins open-source existent pour accéder à la caméra, au GPS, aux fichiers, etc.
- **Idéal pour les prototypes et applications légères :** Permet de créer rapidement des MVPs, des formulaires interactifs ou des applications informatives.

Limites de Cordova

- **Performances limitées :** L'application s'exécute dans une **WebView** (navigateur intégré).
  $\rightarrow$ Le rendu est plus lent que les applications natives.
- **Expérience utilisateur moins fluide :** Les animations, transitions et interactions peuvent manquer de réactivité sur certains appareils.
- **Dépendance aux plugins :** Certaines fonctionnalités (appareil photo, Bluetooth, NFC) nécessitent des plugins externes parfois non maintenus.
- **Difficulté à gérer des interfaces complexes :** Les applications avec beaucoup de graphiques, de cartes ou d'animations lourdes (jeux, dashboards, etc.) sont peu adaptées à Cordova.

Présentation de Xamarin

- **Xamarin** est un framework de développement mobile multiplateforme créé en 2011 par la société **Xamarin Inc.**, puis racheté par **Microsoft** en 2016. Il fait désormais partie intégrante de l'écosystème **.NET**.

- Xamarin permet de développer des applications pour **Android**, **iOS**, et **Windows** en utilisant un seul langage : **C#**.
  $\rightarrow$ Tous les projets partagent la même logique métier, les mêmes bibliothèques et la même architecture logicielle.

- Les applications générées sont **compilées en code natif**, offrant des performances proches des applications développées en Java (Android) ou Swift (iOS).

Présentation de Xamarin

- Xamarin offre deux modes de développement selon le degré de mutualisation souhaité :

- **1. Xamarin.Android / Xamarin.iOS :**
  - Le développeur écrit du code **C#** spécifique à chaque plateforme.
  - L'accès aux API Android et iOS se fait via des **bindings .NET**.
  - Idéal quand on veut un contrôle fin sur l'interface native.

- **2. Xamarin.Forms :**
  - Introduit une couche d'abstraction basée sur **XAML** (langage de description d'UI).
  - Permet de créer une seule interface partagée pour Android, iOS et Windows.
  - Chaque élément XAML est traduit automatiquement en composant natif (bouton, texte, liste…).

Historique de Xamarin

- L'histoire de Xamarin commence avec le projet **Mono** (2001), initié par **Miguel de Icaza** et **Ximian**, visant à créer une version open-source du framework **.NET** pour Linux.
- Après plusieurs années d'évolution, l'équipe fonde en 2011 la société **Xamarin Inc.**, avec pour objectif de permettre le développement d'applications mobiles **Android** et **iOS** en **C#**.
- En **2016**, **Microsoft** rachète Xamarin et l'intègre dans son écosystème **.NET / Visual Studio**, offrant ainsi un environnement complet pour le développement multiplateforme.
- Depuis **2021**, Xamarin évolue vers une nouvelle plateforme appelée **.NET MAUI (Multi-platform App UI)**, qui unifie le développement mobile et desktop sous le même framework .NET.

Architecture de Xamarin

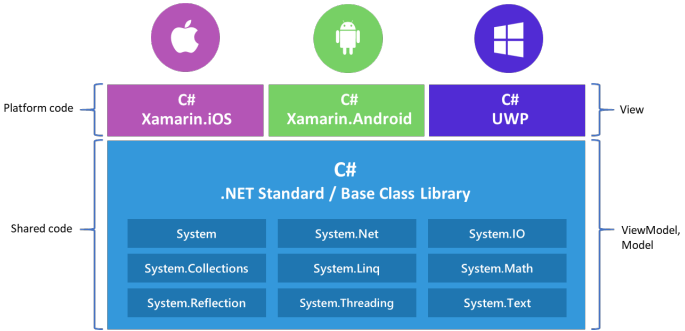

Schéma simplifié de l'architecture Xamarin

Architecture de Xamarin

- **.NET Standard** : bibliothèque commune partagée entre plateformes.
- **Mono Runtime** : environnement d'exécution compatible iOS et Android.
- **Bindings natifs** : accès direct aux API de chaque système via C#.

Avantages de Xamarin

- **Performances proches du natif :** Les applications Xamarin sont compilées en **code machine natif** (AOT ou JIT), assurant une exécution fluide comparable à celle des apps Swift ou Java.
  → Pas de WebView, tout est rendu via les composants UI natifs.

- **Réutilisation du code métier :** Jusqu'à **90 % du code** peut être partagé entre Android, iOS et Windows.
  → Idéal pour les projets de grande envergure (ex : ERP, apps d'entreprise, outils internes).

- **Intégration complète avec Visual Studio et Azure :** Débogage, tests, publication et CI/CD peuvent être gérés dans le même environnement.
  → Forte productivité pour les équipes .NET.

- **Accès direct aux API des plateformes :** Xamarin expose toutes les API Android et iOS via des *bindings .NET*, permettant d'utiliser les fonctionnalités natives sans quitter C#.

Limites de Xamarin

- **Environnement de développement lourd :** L'installation de **Visual Studio**, du SDK Android, et des outils iOS rend la configuration initiale complexe. → Nécessite une machine performante et une bonne connexion.
- **Taille importante des applications :** Les apps compilées avec Xamarin incluent le runtime .NET, ce qui augmente la taille du fichier final (*APK/IPA*).
- **Moins adapté aux petits projets ou prototypes :** Pour un simple formulaire ou une app rapide, Xamarin peut être trop complexe à mettre en place.
- **Courbe d'apprentissage pour les non-.NET :** Les développeurs web ou mobiles venant de JavaScript ou Java doivent s'adapter à l'environnement C#/XAML.

Présentation de Flutter (1/2)

- **Flutter** est un framework de développement multiplateforme créé par **Google** en **2017**. Il permet de développer des applications pour **Android**, **iOS**, **Web**, et **Desktop** à partir d'un **seul code source**.

- Flutter repose sur le langage **Dart**, un langage **orienté objet**, **fortement typé** et **compilé** (en code machine ou en JavaScript pour le web).
  → Dart est conçu pour la rapidité et la productivité des développeurs.

- Contrairement à d'autres frameworks, Flutter ne s'appuie pas sur les composants UI natifs des plateformes. Il redessine chaque élément de l'interface à l'aide de son moteur graphique interne, **Skia**.

- Cette approche garantit une **cohérence visuelle** parfaite et des **performances quasi natives**, quel que soit l'appareil.

Présentation de Flutter (2/2) : Architecture simplifiée

- Flutter est composé de plusieurs couches :
- **1. Framework (en Dart)** : Contient les bibliothèques de haut niveau (widgets, gestion d'état, animations, navigation, etc.).
  → C'est la partie que le développeur manipule directement.
- **2. Engine (en C++)** : Moteur graphique basé sur **Skia**, responsable du rendu, de la gestion du texte et des entrées utilisateur.
  → C'est lui qui transforme les widgets en pixels affichés à l'écran.
- **3. Embedder** : Interface spécifique à chaque OS (Android, iOS, Web, Windows...). Gère la fenêtre, les entrées clavier/tactile et la communication avec le système.

Architecture de Flutter

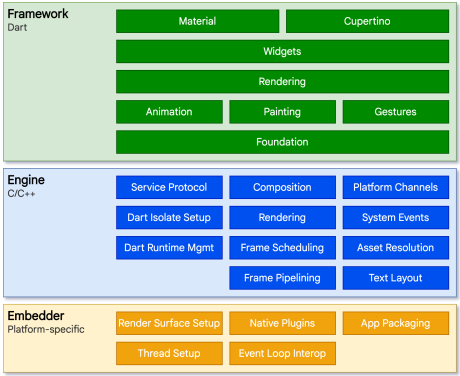

Schéma simplifié de l'architecture Flutter

Architecture de Flutter

- **Framework (Dart)** : gère les widgets, layout, gestion d'état, etc.
- **Engine (C++)** : rendu graphique via Skia.
- **Embedder** : interface entre Flutter et le système (Android/iOS).

Avantages de Flutter

- **Performances quasi natives :** Flutter compile le code Dart en **code machine natif** (AOT), et dessine directement via le moteur **Skia**. → Les animations, transitions et effets visuels sont fluides, sans passer par des ponts natifs.

- **Interface riche et moderne :** Flutter propose une large gamme de **widgets personnalisables** (Material Design, Cupertino, etc.) → Les développeurs peuvent créer des interfaces élégantes, dynamiques et identiques sur Android, iOS et Web.

- **Productivité élevée :** Grâce au **Hot Reload**, les modifications de code sont visibles instantanément sans redémarrer l'application. → Permet d'itérer très rapidement pendant le développement.

- **Code unique pour plusieurs plateformes :** Un seul projet Dart permet de générer des applications pour **mobile**, **web** et **desktop**. → Gain considérable en maintenance et en temps de déploiement.

Limites de Flutter

- **Taille des applications importante :** Les fichiers APK/IPA incluent le moteur Flutter et les dépendances Dart,
  $\rightarrow$ ce qui augmente la taille des applications (souvent $>$ 20–30 Mo).

- **Langage Dart encore jeune :** Bien que performant, **Dart** est moins connu que JavaScript, Python ou C#.
  $\rightarrow$ La communauté reste plus restreinte et certaines entreprises hésitent encore à l'adopter.

- **Écosystème de packages plus limité :** Malgré une croissance rapide, le nombre de bibliothèques et plugins disponibles reste inférieur à celui de l'écosystème **JavaScript (React Native)**.

- **Compatibilité web et desktop encore perfectible :** Certaines fonctionnalités ne sont pas encore totalement optimisées selon les plateformes (notamment sur Safari et macOS).

Flutter dans l'industrie

- Depuis son lancement en **2017**, Flutter s'est imposé comme l'un des frameworks multiplateformes les plus utilisés au monde.
- Il est adopté par de grandes entreprises internationales pour des applications de production :
  - **Google Ads** — gestion publicitaire mobile officielle de Google.
  - **eBay Motors** — application automobile développée entièrement avec Flutter.
  - **Alibaba Group** — plusieurs apps e-commerce et services internes.
  - **BMW**, **Toyota** — interfaces embarquées et apps de connectivité.
  - **Reflectly**, **Nubank**, **Philips Hue** — apps grand public multiplateformes.
- Flutter est soutenu par une communauté très active : plus de **30 000 packages open-source** et une forte présence sur GitHub, Stack Overflow et Medium.
- Aujourd'hui, Flutter est perçu comme le **standard du développement multiplateforme moderne**.

Comparaison des principaux frameworks multiplateformes

Framework	Langage	Approche	Performance	Interface (UI)
Cordova	HTML, CSS, JS	WebView (hybride)	Moyenne	UI Web (non native)
Xamarin	C#, .NET	Code métier partagé	Bonne	UI native (Android/iOS)
Flutter	Dart	Rendu graphique natif	Excellente	UI personnalisée riche

Chaque framework se distingue par son approche du rendu et du partage de code.

Analyse comparative

- **Cordova :** + Facile à apprendre, idéal pour les prototypes ou applications simples. − Limité par les performances de la WebView et une UI non native. → Convient aux projets légers ou formulaires interactifs.

- **Xamarin :** + Performance proche du natif, forte intégration à l'écosystème Microsoft. − Configuration lourde, moins adapté aux petits projets. → Excellent choix pour les entreprises déjà orientées .NET / Azure.

- **Flutter :** + Performance équivalente au natif, UI cohérente sur toutes les plateformes. + Hot Reload, riche bibliothèque de widgets, croissance rapide. − Taille d'app plus grande et communauté Dart encore en expansion. → Aujourd'hui considéré comme la **solution la plus complète et moderne**.

Chapitre 2 : Installation et Configuration de Flutter

Pr. Mohamed BOUDCHICHE

École Nationale de l'Intelligence Artificielle et du Digital (ENIAD)
Université Mohammed Premier

m.boudchiche@edu.ump.ma

Année universitaire 2025–2026

Objectifs du chapitre

- Comprendre les prérequis pour installer Flutter.
- Installer le SDK Flutter sur différents systèmes (Windows, macOS, Linux).
- Configurer l'environnement de développement.
- Vérifier l'installation avec `flutter doctor`.
- Créer et exécuter une première application Flutter.

Étape 1 : Télécharger Flutter

- Rendez-vous sur le site officiel : **`https://flutter.dev/docs/get-started/install`**
- Sélectionnez votre système d'exploitation :
  - **Windows** : téléchargement d'un fichier ZIP.
  - **macOS / Linux** : téléchargement d'une archive `.tar.xz`.
- Décompressez le dossier dans un répertoire stable (éviter le Bureau). **Exemples :**
  - Windows → `C:\src\flutter`
  - macOS / Linux → `/usr/local/flutter`

Étape 2 : Ajouter Flutter au PATH

- Pour exécuter Flutter depuis n'importe quel dossier, il faut l'ajouter au **PATH** (variable d'environnement du système).
- **Sous Windows :**
  1. Ouvrir "Modifier les variables d'environnement système".
  2. Ajouter le chemin suivant : `C:\src\flutter\bin`
- **Sous macOS / Linux :**

Commande à insérer dans `.bashrc` ou `.zshrc`

```
export PATH="$PATH:/usr/local/flutter/bin"
```

- Redémarrer le terminal puis tester : `flutter --version`

Étape 3 : Vérification avec `flutter doctor`

- Flutter intègre un outil de diagnostic appelé **flutter doctor**.
- Commande :

```
flutter doctor
```

- Il vérifie :
  - Le SDK Flutter et Dart.
  - Le SDK Android et les licences associées.
  - L'installation d'un IDE compatible (VS Code, Android Studio).
- Pour accepter les licences Android :

```
flutter doctor --android-licenses
```

Étape 4 : Installation des outils de développement

- Flutter nécessite un environnement de développement complet :
  - **Android Studio** → pour un IDE complet (émulateur intégré).
  - **Visual Studio Code** → plus léger, recommandé pour débutants.
- Installer les plugins :
  - Dans Android Studio → “Flutter” et “Dart”.
  - Dans VS Code → extensions “Flutter” et “Dart”.
- Configurer un émulateur Android (AVD) ou connecter un smartphone physique (Mode développeur + USB Debugging).

Étape 5 : Créer et exécuter un projet Flutter

- Créer un projet :

```
flutter create hello_flutter
```

- Se déplacer dans le dossier : `cd hello_flutter`
- Lancer l'application :

```
flutter run
```

- Flutter compile le projet et affiche la "*Counter App*", un exemple interactif avec un bouton + compteur.

Structure d'un projet Flutter

- **lib/** : contient le code source Dart, notamment `main.dart`.
- **android/**, **ios/**, **web/**, **windows/**, etc. : configurations propres à chaque OS.
- **pubspec.yaml** : fichier clé listant les dépendances, ressources et paramètres du projet.
- **test/** : tests unitaires du projet.
- **build/** : fichiers générés automatiquement après compilation.

Résumé du Chapitre 2

- Installation et configuration du SDK Flutter.
- Ajout du dossier `/bin` au PATH système.
- Vérification avec `flutter doctor`.
- Installation d'un IDE (VS Code ou Android Studio).
- Création et exécution d'une première application Flutter.