Jenga2 – Core/Builders

Ce dossier contient les **builders spécifiques à chaque plateforme cible**.
Chaque builder hérite de la classe abstraite `Builder` (définie dans `Core/Builder.py`) et implémente les méthodes nécessaires à la compilation et à l'édition de liens pour une plateforme donnée.

🗂 Sommaire

🎯 Rôle des builders

Un **builder** est responsable de :

1. **Compiler** les fichiers sources en objets (`.obj`, `.o`).
2. **Éditer les liens** pour produire l'exécutable ou la bibliothèque finale (`.exe`, `.dll`, `.so`, `.a`, `.lib`, `.app`, `.apk`, ...).
3. **Gérer les spécificités** de la plateforme : flags de compilation, extensions de fichiers, organisation des répertoires de sortie, signature, packaging, etc.

Les builders sont utilisés par la commande `jenga build` (via `BuildCommand`) et peuvent également être sollicités directement par d'autres commandes (`package`, `deploy`, `test`).

🏗 Architecture

Tous les builders héritent de `Builder` (classe abstraite) qui définit le squelette suivant :

rihen.universe@gmail.com • teuguiasederis@gmail.com
(+237) 693-761-773

```python
class Builder(abc.ABC):
    def __init__(self, workspace, config, platform, targetOs, targetArch,
targetEnv=None, verbose=False):
        # Initialisation commune, résolution de la toolchain, validation
hôte/cible
        ...

    @abc.abstractmethod
    def GetObjectExtension(self) -> str: ...
    @abc.abstractmethod
    def GetOutputExtension(self, project: Project) -> str: ...
    @abc.abstractmethod
    def Compile(self, project: Project, sourceFile: str, objectFile: str) ->
bool: ...
    @abc.abstractmethod
    def Link(self, project: Project, objectFiles: List[str], outputFile: str) -
> bool: ...
    @abc.abstractmethod
    def GetModuleFlags(self, project: Project, sourceFile: str) -> List[str]:
...

    # Méthodes utilitaires (GetObjectDir, GetTargetDir, IsModuleFile, …)
```

Chaque builder concret implémente ces méthodes avec les commandes et flags propres à la plateforme.

📋 Liste des builders disponibles

Builder	Cible(s)	Compilateur(s) supportés	État
`WindowsBuilder`	Windows (x86, x86_64, ARM64)	MSVC, Clang-cl, MinGW (GCC)	☑ Complet
`LinuxBuilder`	Linux (x86_64, ARM, ARM64)	GCC, Clang	☑ Complet
`MacosBuilder`	macOS (x86_64, ARM64)	AppleClang	☑ Complet
`AndroidBuilder`	Android (armeabi-v7a, arm64-v8a, x86, x86_64)	NDK Clang	☑ Complet
`IOSBuilder`	iOS (device, simulator)	AppleClang (via xcrun)	☑ Complet
`EmscriptenBuilder`	Web (WASM)	Emscripten	☑ Complet
`XboxBuilder`	Xbox One, Xbox Series X/S	MSVC + GDK	☑ Complet

Builder	Cible(s)	Compilateur(s) supportés	État
`Ps4Builder`	PlayStation 4	Sony Clang (propriétaire)	⚠ Placeholder
`Ps5Builder`	PlayStation 5	Sony Clang (propriétaire)	⚠ Placeholder
`SwitchBuilder`	Nintendo Switch	Nintendo Clang	⚠ Placeholder

> **Note** : Les builders pour PS4, PS5 et Switch nécessitent les SDK officiels des constructeurs. Ils sont fournis sous forme de squelettes et doivent être complétés par les détenteurs de licences.

🧩 Créer un nouveau builder

1. **Créer un fichier** `MaPlateforme.py` dans `Core/Builders/`.
2. **Définir une classe** `MaPlateformeBuilder(Builder)`.
3. **Implémenter les méthodes abstraites** (voir l'exemple ci-dessous).
4. **Ajouter l'export** dans `Core/Builders/__init__.py`.

Exemple minimal

```python
from ..Builder import Builder

class MaPlateformeBuilder(Builder):
    def GetObjectExtension(self) -> str:
        return ".o"

    def GetOutputExtension(self, project: Project) -> str:
        if project.kind == ProjectKind.SHARED_LIB:
            return ".so"
        elif project.kind == ProjectKind.STATIC_LIB:
            return ".a"
        else:
            return ".bin"

    def Compile(self, project, sourceFile, objectFile) -> bool:
        # ... commande de compilation
        pass

    def Link(self, project, objectFiles, outputFile) -> bool:
        # ... commande d'édition de liens
        pass

    def GetModuleFlags(self, project, sourceFile) -> List[str]:
        # ... flags pour les modules C++20 (ou [] si non supporté)
        return []
```

📐 Conventions et bonnes pratiques

- **Chemins de sortie** : utiliser `GetObjectDir(project)` et `GetTargetDir(project)` – ces méthodes fournissent les répertoires par défaut définis par le projet ou le workspace.
- **Flags utilisateur** : toujours inclure `project.cflags` / `project.cxxflags` / `project.ldflags` dans les commandes de compilation et d'édition de liens.
- **Modules C++20** : tester `IsModuleFile(sourceFile)` et appeler `GetModuleFlags(project, sourceFile)`.
- **Détection de la toolchain** : s'appuyer sur `self.toolchain` (instance de `Toolchain`), ne pas durcir de chemins absolus.
- **Gestion des erreurs** : retourner `False` en cas d'échec, le message d'erreur doit être affiché par la commande appelante (ou le builder peut logger via `Colored.PrintError`).
- **Validation hôte/cible** : dans le constructeur, vérifier que la combinaison est possible (ex: iOS nécessite macOS). Utiliser `_ValidateHostTarget()` hérité de `Builder`.

🔧 Intégration avec le système de toolchains

Chaque builder reçoit une **toolchain** résolue automatiquement par `Builder._ResolveToolchain()`.
La toolchain contient les chemins des exécutables (`ccPath`, `cxxPath`, `arPath`, ...) et les informations de cible (`targetOs`, `targetArch`, ...).
Le builder peut également accéder au gestionnaire global `ToolchainManager` pour détecter des compilateurs supplémentaires.

📦 Dépendances et prérequis

Certains builders nécessitent des outils externes :

Builder	Outils requis
Windows	Visual Studio ou MinGW, Windows SDK
Linux	GCC/Clang, binutils
macOS	Xcode Command Line Tools (clang, libtool, codesign)
Android	Android NDK, SDK, JDK
iOS	macOS, Xcode Command Line Tools (xcrun, clang, libtool, codesign)
Emscripten	Emscripten SDK
Xbox	Microsoft GDK, Visual Studio, Xbox Extensions (GDKX)
PS4/PS5/Switch	SDK propriétaires (non inclus)

🔍 Détail par builder

WindowsBuilder

- **Familles supportées** : MSVC (`cl.exe`), Clang-cl (`clang-cl`), MinGW (`gcc`, `g++`).
- **Extensions** : `.obj`, `.exe`, `.dll`, `.lib`.
- **Flags particuliers** : `/Zi` (PDB), `/O2`, `/Wall`, etc.
- **Modules C++20** : `/interface`, `/module:output<fichier.ifc>`, `/internalPartition`.
- **Ressources** : pas encore implémenté (`.rc` → `.res` → link).

LinuxBuilder

- **Familles** : GCC, Clang.
- **Extensions** : `.o`, `.so`, `.a`, exécutable sans extension.
- **Modules C++20** : `-fmodules-ts` (GCC), `-fmodules` (Clang).
- **RPATH** : géré via `-Wl,-rpath,<path>`.

MacosBuilder

- **Compilateur** : AppleClang (via `xcrun`).
- **Extensions** : `.o`, `.dylib`, `.a`, bundle `.app`.
- **Modules C++20** : `-fmodules`, `-fcxx-modules`.
- **Frameworks** : `-framework`, `-F`.

AndroidBuilder

- **Toolchain** : NDK Clang (LLVM).
- **ABI** : armeabi-v7a, arm64-v8a, x86, x86_64.
- **Packaging** : génération APK (signé) et AAB (via `bundletool`).
- **Flags** : `--target=`, `--sysroot`, `-DANDROID`, `-fPIC`.

IOSBuilder

- **Compilateur** : AppleClang via `xcrun`.
- **Modes** : device (`arm64`), simulateur (`x86_64`).
- **Sortie** : bibliothèque statique (`.a`) ou bundle `.app`.
- **Signature** : via `codesign` avec identité et entitlements.
- **Modules C++20** : `-fmodules`, `-fcxx-modules`.

EmscriptenBuilder

- **Compilateur** : `emcc`, `em++`.
- **Cible** : WebAssembly (`.wasm` + `.js` + `.html`).
- **Flags** : `-s WASM=1`, `-s USE_WEBGL2`, etc.
- **Optimisations** : `-O2`, `-Oz`.

XboxBuilder

- **Environnement** : Windows + Microsoft GDK.
- **Plateformes** : `Gaming.Xbox.XboxOne.x64`, `Gaming.Xbox.Scarlett.x64`.
- **Compilation** : MSVC avec `DWINAPI_FAMILY_GAMES`.

- **Packaging** : XVC avec `MakePkg`, signature, déploiement via `xbapp`.

PlayStation 4/5 & Switch

- **Placeholders** : non implémentés – nécessitent les SDK des constructeurs.
- **Contribution** : pour les développeurs agréés, suivre le même modèle que `XboxBuilder`.

🐞 Dépannage et erreurs fréquentes

Erreur	Cause probable	Solution
`No suitable toolchain found for ...`	Compilateur non installé ou non détecté	Vérifier la présence du compilateur (MSVC, GCC, NDK, ...) et les variables d'environnement.
`Cannot build for iOS from ...`	Build iOS depuis Windows/Linux	Uniquement possible sur macOS avec Xcode CLI.
`MicrosoftGame.config not found`	Fichier de configuration Xbox manquant	Généré automatiquement par`XboxBuilder._GenerateMicrosoftGameConfig`.
`apksigner not found`	Android SDK build-tools manquants	Installer les build-tools via SDK Manager.
`xcrun: error: unable to find utility ...`	Xcode CLI non installé	Exécuter`xcode-select --install`.
`Module file not supported`	Le builder ne supporte pas encore les modules C++20	Vérifier la version du compilateur (MSVC 2019+, Clang 16+, GCC 11+).

🔗 Liens utiles

- Documentation Core
- Système de toolchains
- Guide de contribution (à créer)

Ce document est maintenu par l'équipe Rihen. Toute suggestion d'amélioration est la bienvenue.