

Exemples/ – Projets d'exemple Jenga2

Le dossier [Exemples/](#) contient une série de projets illustrant l'utilisation de Jenga2 dans divers contextes. Chaque sous-dossier est un workspace indépendant, prêt à être compilé avec [jenga2 build](#).

Structure proposée

```
Exemples/
├── 01_hello_console/          # Application console "Hello World"
├── 02_static_library/        # Bibliothèque statique + application de test
├── 03_shared_library/        # Bibliothèque dynamique
├── 04_unit_tests/            # Tests unitaires avec Unitest
├── 05_android_ndk/           # Application Android (native activity)
├── 06_ios_app/               # Application iOS (nécessite macOS)
├── 07_web_wasm/              # WebAssembly avec Emscripten
├── 08_custom_toolchain/       # Utilisation d'une toolchain personnalisée
├── 09_multi_projects/         # Workspace avec plusieurs projets et dépendances
├── 10_modules_cpp20/          # Modules C++20 (MSVC/Clang)
├── 11_benchmark/              # Benchmark avec Google Benchmark
├── 12_external_includes/      # Inclusion de fichiers .jenga externes
├── 13_packaging/              # Création de packages (APK, DEB, etc.)
├── 14_cross_compile/          # Cross-compilation Windows → Linux/Android
└── README.md                  # Présentation des exemples
```

◊ 01_hello_console

Objectif : compiler une application console simple sur Windows, Linux ou macOS.

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Hello from Jenga2!" << std::endl;
    return 0;
}
```

hello.jenga

```
from Jenga2.Api import *

with workspace("HelloConsole"):
    configurations(["Debug", "Release"])
```

```
targetoses([TargetOS.WINDOWS, TargetOS.LINUX, TargetOS.MACOS])
targetarchs([TargetArch.X86_64])

with project("Hello"):
    consoleapp()
    language("C++")
    files(["main.cpp"])
```

Utilisation :

```
cd Exemples/01_hello_console
jenga2 build
./Build/Bin/Debug>Hello # ou Hello.exe sur Windows
```

◊ 02_static_library

Objectif : créer une bibliothèque statique et l'utiliser dans une application.

mathlib/include/mathlib.h

```
#pragma once
int add(int a, int b);
```

mathlib/src/mathlib.cpp

```
#include "mathlib.h"
int add(int a, int b) { return a + b; }
```

app/main.cpp

```
#include <iostream>
#include <mathlib.h>
int main() {
    std::cout << "3 + 5 = " << add(3, 5) << std::endl;
    return 0;
}
```

staticlib.jenga

```

from Jenga2.Api import *

with workspace("StaticLibExample"):
    configurations(["Debug", "Release"])
    targetoses([TargetOS.WINDOWS, TargetOS.LINUX, TargetOS.MACOS])

    with project("MathLib"):
        staticlib()
        language("C++")
        location("mathlib")
        files(["src/**.cpp"])
        includedirs(["include"])

    with project("App"):
        consoleapp()
        location("app")
        files(["main.cpp"])
        includedirs(["../mathlib/include"])
        links(["MathLib"])
        dependson(["MathLib"])

```

Utilisation : même principe.

◊ 03_shared_library

Objectif : créer une bibliothèque dynamique (.dll/.so/.dylib).

Similaire à [02_static_library](#), mais avec `sharedlib()` au lieu de `staticlib()`.

◊ 04_unit_tests

Objectif : écrire et exécuter des tests unitaires avec Unitest.

src/calculator.cpp

```

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

```

include/calculator.h

```

#pragma once
int add(int a, int b);
int sub(int a, int b);

```

tests/test_calculator.cpp

```
#include <Unitest/TestMacro.h>
#include "calculator.h"

TEST_CASE(Calculator, Add) {
    ASSERT_EQUAL(5, add(2, 3));
}

TEST_CASE(Calculator, Sub) {
    ASSERT_EQUAL(1, sub(3, 2));
}
```

unittest.jenga

```
from Jenga2.Api import *

with workspace("UnitTestDemo"):
    configurations(["Debug", "Release"])
    targetoses([TargetOS.WINDOWS, TargetOS.LINUX, TargetOS.MACOS])

    # Configuration Unitest (précompilé)
    with unittest() as u:
        u.Precompiled()

    with project("Calculator"):
        staticlib()
        language("C++")
        files(["src/**.cpp"])
        includedirs(["include"])

        with test():
            testfiles(["tests/**.cpp"])
```

Utilisation : `jenga2 test`.

◊ 05_android_ndk

Objectif : compiler une application Android native (NativeActivity) et générer un APK signé.

android.jenga

```
from Jenga2.Api import *

with workspace("AndroidDemo"):
    configurations(["Debug", "Release"])
```

```

targetoses([TargetOS.ANDROID])
targetarchs([TargetArch.ARCH64, TargetArch.X86_64])

# Chemins SDK/NDK (à adapter)
androidsdkpath("C:/Android/Sdk")
androidndkpath("C:/Android/ndk")

with project("NativeApp"):
    windowedapp()
    language("C++")
    files(["src/**.cpp"])
    includedirs(["include"])

    androidapplicationid("com.jenga.demo")
    androidversioncode(1)
    androidversionname("1.0")
    androidminsdk(21)
    androidtargetsdk(33)
    androidabis(["arm64-v8a", "x86_64"])
    androidnativeactivity(True)

    # Signature (optionnelle)
    androidsign(True)
    androidkeystore("keystore.jks")
    androidkeystorepass("android")
    androidkeyalias("mykey")

```

Utilisation : jenga2 build --platform Android-arm64 puis jenga2 package --platform android --type apk.

◊ 06_ios_app

Objectif : compiler une application iOS (simulateur ou périphérique) et générer un IPA.

Nécessite macOS et Xcode CLI.

📄 ios.jenga

```

from Jenga2.Api import *

with workspace("IOSDemo"):
    configurations(["Debug", "Release"])
    targetoses([TargetOS.IOS])
    targetarchs([TargetArch.ARCH64, TargetArch.X86_64])

    with project("MyApp"):
        windowedapp()
        language("C++")
        files(["src/**.mm", "src/**.cpp"])

```

```
includedirs(["include"])

iosbundleid("com.jenga.demo")
iosversion("1.0")
iosminsdk("14.0")
iosbuildnumber("42")
# Identité de signature (à remplacer)
iossigningidentity("Apple Development: ...")
iosentitlements("MyApp.entitlements")
iosappicon("icon.png")
```

Utilisation : `jenga2 build --platform iOS` (simulateur) ou `jenga2 build --platform iOS-arm64` (device).

`jenga2 package --platform ios --type ipa` génère l'IPA.

◊ 07_web_wasm

Objectif : compiler une application WebAssembly avec Emscripten.

📄 `web.jenga`

```
from Jenga2.Api import *

with workspace("WebDemo"):
    configurations(["Release"])
    targetoses([TargetOS.WEB])
    targetarchs([TargetArch.WASM32])

    # Détection automatique d'Emscripten (sinon, spécifier via addtools)
    usetoolchain("emscripten")

    with project("WasmApp"):
        consoleapp()
        language("C++")
        files(["src/**.cpp"])
        embedresources(["assets/**"])
```

Utilisation : `jenga2 build --platform Web`.

Le résultat (`index.html`, `.js`, `.wasm`) se trouve dans `Build/Bin/Release/`.

`jenga2 package --platform web --type zip` crée une archive.

◊ 08_custom_toolchain

Objectif : définir manuellement une toolchain (ex: GCC personnalisé).

📄 `custom.jenga`

```

from Jenga2.Api import *

with workspace("CustomToolchain"):
    configurations(["Debug", "Release"])
    targetoses([TargetOS.LINUX])
    targetarchs([TargetArch.X86_64])

    with toolchain("mygcc", "gcc"):
        settarget("Linux", "x86_64", "gnu")
        ccompiler("/opt/gcc-11/bin/gcc")
        cppcompiler("/opt/gcc-11/bin/g++")
        ar("/opt/gcc-11/bin/ar")
        cflags(["-march=native"])
        cxxflags(["-std=c++20"])

usetoolchain("mygcc")

with project("Test"):
    consoleapp()
    files(["main.cpp"])

```

◊ 09_multi_projects

Objectif : workspace avec plusieurs projets interdépendants.

- **Engine** (bibliothèque statique)
- **Game** (application console liée à Engine)
- **Tools** (outil de conversion)

Les dépendances sont gérées via `dependsOn`.

◊ 10_modules_cpp20

Objectif : utiliser les modules C++20 (`.cppm` / `.ixx`).

math.cpp

```

export module math;
export int add(int a, int b) { return a + b; }

```

main.cpp

```

import math;
#include <iostream>

```

```
int main() { std::cout << add(2, 3) << std::endl; }
```

modules.jenga

```
with project("MathApp"):  
    consoleapp()  
    language("C++")  
    cppdialect("C++20")  
    files(["math.cpp", "main.cpp"])
```

Note : nécessite un compilateur compatible (MSVC 2019+ / Clang 16+).

◊ 11_benchmark

Objectif : intégrer Google Benchmark et exécuter des benchmarks.

bench.cpp

```
#include <benchmark/benchmark.h>  
  
static void BM_StringCreation(benchmark::State& state) {  
    for (auto _ : state)  
        std::string empty_string;  
}  
BENCHMARK(BM_StringCreation);  
  
BENCHMARK_MAIN();
```

bench.jenga

```
with project("Bench"):  
    consoleapp()  
    files(["bench.cpp"])  
    links(["benchmark"])  
    includedirs(["/usr/local/include"])  
    libdirs(["/usr/local/lib"])
```

Utilisation : `jenga2 bench --project Bench`.

◊ 12_external_includes

Objectif : inclure des bibliothèques externes via `include`.

```
libs/
logger.jenga          # définit un projet "Logger"
math.jenga             # définit "MathLib"
main.jenga              # workspace principal
```

📄 main.jenga

```
from Jenga2.Api import *

with workspace("MyApp"):
    with include("libs/logger.jenga"):
        pass
    with include("libs/math.jenga"):
        pass

    with project("App"):
        consoleapp()
        files(["main.cpp"])
        links(["Logger", "MathLib"])
        dependson(["Logger", "MathLib"])
```

◊ 13_packaging

Objectif : générer un package DEB (Linux) et un APK (Android) après compilation.

Reprend les exemples 05 et ajoute les commandes de packaging dans le fichier .jenga via postbuildCommands.

```
with project("MyApp"):
    # ...
    postbuildCommands([
        "jenga2 package --platform android --type apk --project MyApp",
        "jenga2 package --platform linux --type deb --project MyApp"
    ])
```

◊ 14_cross_compile

Objectif : depuis Windows, compiler pour Linux (via MinGW-cross) et pour Android.

📄 cross.jenga

```

with workspace("Cross"):
    targetoses([TargetOS.LINUX, TargetOS.ANDROID])
    targetarchs([TargetArch.X86_64, TargetArch.ARM64])

    # Toolchain pour Linux (cross depuis Windows)
    with addtools({
        "mingw-linux": {
            "type": "compiler",
            "cc": "x86_64-w64-mingw32-gcc",
            "cxx": "x86_64-w64-mingw32-g++"
        }
    }):
        usetool("mingw-linux")

    with project("App"):
        consoleapp()
        files(["main.cpp"])

```

Exemples/README.md

Exemples Jenga2

Ce dossier contient des exemples complets et fonctionnels du système de build Jenga2.

Chaque sous-dossier est un **workspace indépendant** qui peut être construit avec la commande `jenga2 build` .

Prérequis

- Jenga2 installé (`pip install -e .` depuis la racine du projet)
- Pour certains exemples, des toolchains ou SDK spécifiques sont nécessaires (voir le README de chaque exemple).

Liste des exemples

Dossier	Description
01_hello_console	Application console minimale
02_static_library	Bibliothèque statique + utilisation
03_shared_library	Bibliothèque dynamique
04_unit_tests	Tests unitaires avec Unitest
05_android_ndk	Application Android native (APK)

06_ios_app	Application iOS (IPA) - nécessite macOS
07_web_wasm	WebAssembly avec Emscripten
08_custom_toolchain	Définition manuelle d'une toolchain
09_multi_projects	Workspace avec projets interdépendants
10_modules_cpp20	Modules C++20 (`.cppm`)
11_benchmark	Benchmark avec Google Benchmark
12_external_includes	Inclusion de dépendances externes
13_packaging	Génération de packages (DEB, APK, ...)
14_cross_compile	Cross-compilation Windows → Linux/Android

Chaque exemple contient un fichier `README.md` propre avec les instructions de compilation.