Documentation Vix.cpp

Bien démarrer avec Vix.cpp et Le Livre Vix

Un guide pratique pour créer des applications C++ rapides, fiables et modernes avec Vix.cpp.

Auteur : Gaspard Kirira

Projet : Vix.cpp

GitHub : https://github.com/vixcpp/vix

Documentation : https://docs.vixcpp.com

Date : Mai 2026

À propos de ce document

Ce document rassemble dans un seul PDF la partie **Bien démarrer** et **Le Livre Vix**. Il a été conçu pour vous permettre d'apprendre **Vix.cpp** étape par étape, depuis l'installation jusqu'au déploiement en production.

En suivant ce livre, vous passerez progressivement d'un simple fichier C++ exécuté avec `vix run main.cpp` à une application backend complète comprenant :

- routage **HTTP** et **API JSON**,
- **middleware** (CORS, rate limiting, authentification),
- **validation** des entrées,
- gestion structurée des erreurs et **journalisation**,
- accès à une **base de données** (SQLite, MySQL),
- communication temps réel via **WebSocket**,
- **runtime asynchrone** C++20 (event loop, coroutines, timers, thread pool),
- **cache** local et offline-friendly,
- **synchronisation offline-first** avec WAL et outbox,
- communication **P2P** entre nœuds,
- **déploiement en production** avec Nginx, systemd et HTTPS.

La **Partie 1 — Bien démarrer** présente le chemin le plus court entre zéro et une application Vix en cours d'exécution. La **Partie 2 — Le Livre Vix** approfondit chaque concept avec un fil pédagogique progressif.

> **Note éditoriale.** Tous les blocs de code, commandes terminal, noms de classes, en-têtes, routes HTTP et variables d'environnement de la documentation originale anglaise ont été conservés à l'identique. Seul le texte explicatif a été traduit en français.

Table des matières

partie I

Bien démarrer

Chapitre 1

Bienvenue dans Vix.cpp

1.1 PDF hors ligne

Vous pouvez télécharger la documentation complète de Vix.cpp au format PDF :

Télécharger le PDF de la documentation Vix.cpp

Vix.cpp est un runtime C++ moderne et une boîte à outils pour développeurs, conçu pour créer des applications rapides et fiables avec un workflow plus fluide.

Il offre au C++ une expérience de développement directe :

```
vix run main.cpp
```

Et un workflow de projet :

```
vix new api
cd api
vix build
vix run
```

1.2 Qu'est-ce que Vix.cpp ?

Vix.cpp vous aide à construire des applications C++ sans avoir à câbler manuellement, à chaque nouveau projet, le système de build, les commandes runtime, les logs, les dépendances et le workflow de développement.

L'objectif est simple :

> Garder la puissance du C++, rendre le workflow applicatif plus simple.

Vix ne remplace pas le C++. Il fournit au C++ un workflow orienté runtime tout autour de lui.

1.3 Ce que vous pouvez construire

Avec Vix, vous pouvez construire :

- des serveurs HTTP,
- des API JSON,
- des services backend,
- des applications WebSocket,
- des outils CLI,
- des bibliothèques C++,
- des applications web basées sur des templates,
- des systèmes local-first et offline-first,
- des services en production derrière Nginx et systemd.

Une application HTTP Vix minimale ressemble à ceci :

```
#include <vix.hpp>

using namespace vix;

int main()
{
  App app;

  app.get("/", [](Request &, Response &res) {
    res.json({
      "message", "Hello from Vix",
      "framework", "Vix.cpp"
    });
  });

  app.run(8080);
  return 0;
}
```

Exécutez-la :

```
vix run main.cpp
```

Puis ouvrez :

```
http://localhost:8080/
```

1.4 Le workflow principal

Pour un seul fichier C++ :

```
vix run main.cpp
```

Pour un projet complet :

```
vix new api
cd api
vix build
vix run
```

Pour le mode développement :

```
vix dev
```

Pour les vérifications et les tests :

```
vix check
vix tests
```

1.5 Comment Bien démarrer est organisé

Cette section vous donne le chemin le plus court entre zéro et une application Vix en cours d'exécution.

Lisez-la dans l'ordre :

1. Installation
2. Configurer votre environnement
3. Exécuter votre premier fichier C++
4. Créer votre premier projet
5. Votre premier serveur HTTP

1.6 Bien démarrer vs Le Livre Vix

Bien démarrer est court et pratique. Cette partie vous accompagne pour :

```
installer → vérifier → exécuter → créer un projet → démarrer le serveur
```

Le Livre Vix va plus en profondeur. Il explique le modèle mental qui sous-tend Vix, puis enseigne les routes, les requêtes, les réponses, les API JSON, le middleware, la validation, la base de données, WebSocket, le runtime asynchrone, le cache, la synchronisation, P2P et le déploiement en production.

Commencez ici. Puis continuez avec le livre quand vous voulez comprendre Vix étape par étape.

1.7 Ce qu'il vous faut

Vous avez seulement besoin de connaissances C++ de base :

- les fonctions,
- les headers,
- `std::string`,
- les lambdas,
- l'utilisation basique du terminal.

Vous n'avez pas besoin d'être un expert CMake pour commencer. Vix peut créer un projet, le construire, l'exécuter et vous fournir une boucle de développement propre.

1.8 La première commande à retenir

```
vix run main.cpp
```

Cette commande est le moyen le plus rapide d'exécuter un fichier C++ avec Vix.

Quand votre application grandit, passez à un projet :

```
vix new api
cd api
vix dev
```

1.9 Étape suivante

Installez Vix sur votre machine.

Suivant : Installation

Chapitre 2

Installation

Cette page montre comment installer Vix.cpp et vérifier qu'il fonctionne correctement sur votre machine.

Pour cette section Bien démarrer, installez le SDK complet.

Le SDK complet inclut la CLI `vix`, les headers et les bibliothèques nécessaires pour construire des applications Vix.

2.1 Installer sur Linux ou macOS

Exécutez :

```
curl -fsSL https://vixcpp.com/install.sh | bash
```

Après l'installation, redémarrez votre terminal ou rechargez la configuration de votre shell.

2.2 Installer sur Windows

Ouvrez PowerShell et exécutez :

```
irm https://vixcpp.com/install.ps1 | iex
```

2.3 Vérifier l'installation

Vérifiez que la commande `vix` est disponible :

```
vix --version
```

Forme de sortie attendue :

```
Vix.cpp CLI
version : 2.5.3
author  : Gaspard Kirira
source  : https://github.com/vixcpp/vix
```

La version exacte peut différer selon la dernière release disponible.

2.4 Vérifier les headers du SDK

Pour les applications C++ qui utilisent Vix, les headers du SDK doivent être installés.

Vérifiez que `vix.hpp` est présent :

```
find ~/.local/include -name vix.hpp 2>/dev/null
```

Sortie attendue :

```
/home/your-user/.local/include/vix.hpp
```

Si rien n'apparaît, réinstallez le SDK complet.

2.5 Mode SDK vs mode CLI-only

Vix propose deux modes d'installation.

Mode	Ce qu'il installe	Quand l'utiliser
Mode SDK	CLI, headers et bibliothèques	Vous voulez construire des applications Vix.
Mode CLI-only	Uniquement le binaire `vix`	Vous avez seulement besoin de la CLI.

Pour ce guide Bien démarrer, utilisez le mode SDK par défaut.

N'utilisez pas le mode CLI-only si vous voulez compiler du code qui inclut :

```
#include <vix.hpp>
```

2.6 Mode CLI-only

Le mode CLI-only installe uniquement l'outil en ligne de commande.

```
VIX_INSTALL_KIND=cli curl -fsSL https://vixcpp.com/install.sh | bash
```

Ce mode n'est pas recommandé pour ce guide car les pages suivantes construisent de vraies applications Vix.

2.7 Corriger les problèmes de PATH

Si votre terminal affiche :

```
vix: command not found
```

Ajoutez `~/.local/bin` à votre `PATH`.

2.7.1 Bash

```
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

2.7.2 Zsh

```
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

Puis vérifiez à nouveau :

```
vix --version
```

2.8 Installer les prérequis de build

Vix utilise la chaîne d'outils C++ habituelle en interne.

Assurez-vous que votre système dispose d'un compilateur, de CMake, de Ninja et des bibliothèques de développement courantes.

2.8.1 Ubuntu ou Debian

```
sudo apt update
sudo apt install -y \
  build-essential cmake ninja-build pkg-config \
  libssl-dev libsqlite3-dev zlib1g-dev libbrotli-dev \
  nlohmann-json3-dev libspdlog-dev libfmt-dev
```

2.8.2 macOS

Avec Homebrew :

```
brew install cmake ninja pkg-config openssl@3 spdlog fmt nlohmann-json brotli
```

2.8.3 Windows

Installez Visual Studio Build Tools avec MSVC ou clang-cl.

Pour les dépendances supplémentaires, utilisez vcpkg.

2.9 Vérifier votre chaîne d'outils

Exécutez :

```
c++ --version
cmake --version
ninja --version
```

Si l'une de ces commandes manque, installez l'outil manquant avant de continuer.

2.10 Test rapide

Créez un dossier temporaire :

```
mkdir -p ~/tmp/vix-install-test
cd ~/tmp/vix-install-test
```

Créez `main.cpp` :

```
cat > main.cpp <<'CPP'
#include <iostream>

int main()
{
  std::cout << "Hello from Vix\n";
  return 0;
}
CPP
```

Exécutez-le :

```
vix run main.cpp
```

Sortie attendue :

```
Hello from Vix
```

2.11 Problèmes d'installation courants

2.11.1 `vix: command not found`

Votre shell ne trouve pas le binaire Vix.

Correctif :

```
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

Puis exécutez :

```
vix --version
```

2.11.2 `#include <vix.hpp>` introuvable

Le SDK complet n'est pas installé, ou les headers ne sont pas visibles.

Vérifiez :

```
find ~/.local/include -name vix.hpp 2>/dev/null
```

Si rien n'apparaît, réinstallez Vix sans le mode CLI-only :

```
curl -fsSL https://vixcpp.com/install.sh | bash
```

2.11.3 CMake ou Ninja sont absents

Vérifiez :

```
cmake --version
ninja --version
```

Sur Ubuntu ou Debian :

```
sudo apt install -y cmake ninja-build
```

2.12 Mettre à jour Vix plus tard

Pour mettre à jour la CLI :

```
vix upgrade
```

Pour inspecter votre environnement :

```
vix doctor
```

Pour inspecter les chemins de Vix et les informations de cache :

```
vix info
```

2.13 Ce qu'il faut retenir

Installez le SDK complet :

```
curl -fsSL https://vixcpp.com/install.sh | bash
```

Vérifiez la CLI :

```
vix --version
```

Vérifiez les headers du SDK :

```
find ~/.local/include -name vix.hpp 2>/dev/null
```

Pour ce guide, le mode SDK est le bon mode d'installation.

2.14 Étape suivante

Configurez maintenant votre environnement de développement.

Suivant : Configurer votre environnement

Chapitre 3

Configurer votre environnement

Cette page vous aide à préparer un environnement de développement propre pour Vix.cpp.

Avant d'écrire de vraies applications Vix, assurez-vous que votre terminal, votre compilateur, vos outils de build et votre dossier de projet sont prêts.

3.1 Vérifier Vix

D'abord, vérifiez que la commande `vix` est disponible :

```
vix --version
```

Forme de sortie attendue :

```
Vix.cpp CLI
version : 2.5.3
author  : Gaspard Kirira
source  : https://github.com/vixcpp/vix
```

La version exacte peut être différente.

3.2 Vérifier votre compilateur C++

Vix utilise le compilateur C++ de votre système en arrière-plan.

Exécutez :

```
c++ --version
```

Vous devriez voir un compilateur tel que GCC, Clang, MSVC ou clang-cl.

Sur Linux, GCC ou Clang sont recommandés.

3.3 Vérifier CMake

Vix utilise CMake pour la compilation des projets.

Exécutez :

```
cmake --version
```

Si CMake est absent sur Ubuntu ou Debian :

```
sudo apt update
sudo apt install -y cmake
```

3.4 Vérifier Ninja

Ninja est le backend de build recommandé pour les projets Vix.

Exécutez :

```
ninja --version
```

Si Ninja est absent sur Ubuntu ou Debian :

```
sudo apt install -y ninja-build
```

3.5 Vérifier les bibliothèques de développement courantes

Pour la plupart des applications Vix, installez les dépendances C++ de développement courantes.

3.5.1 Ubuntu ou Debian

```
sudo apt update
sudo apt install -y \
  build-essential cmake ninja-build pkg-config \
  libssl-dev libsqlite3-dev zlib1g-dev libbrotli-dev \
  nlohmann-json3-dev libspdlog-dev libfmt-dev
```

3.5.2 macOS

Avec Homebrew :

```
brew install cmake ninja pkg-config openssl@3 spdlog fmt nlohmann-json brotli
```

3.5.3 Windows

Installez Visual Studio Build Tools avec MSVC ou clang-cl.

Pour les bibliothèques optionnelles, utilisez vcpkg.

3.6 Dossier de projet recommandé

Créez un dossier propre pour vos expérimentations :

```
mkdir -p ~/tmp
cd ~/tmp
```

Vous pouvez utiliser ce dossier pour les premiers exemples de ce guide.

3.7 Créer un fichier de test rapide

Créez `main.cpp` :

```
cat > main.cpp <<'CPP'
#include <iostream>

int main()
{
  std::cout << "Hello from my C++ environment\n";
  return 0;
}
CPP
```

Exécutez-le avec Vix :

```
vix run main.cpp
```

Sortie attendue :

```
Hello from my C++ environment
```

Si cela fonctionne, votre environnement C++ de base est prêt.

3.8 Tester un programme HTTP Vix

Testez maintenant que les headers et les bibliothèques du SDK Vix sont disponibles.

Remplacez `main.cpp` :

```
cat > main.cpp <<'CPP'
#include <vix.hpp>
```

```cpp
using namespace vix;

int main()
{
  App app;

  app.get("/", [](Request &, Response &res) {
    res.json({
      "message", "Hello from Vix",
      "framework", "Vix.cpp"
    });
  });

  app.run(8080);

  return 0;
}
```
CPP

Exécutez-le :

```
vix run main.cpp
```

Forme de sortie attendue :

```
Vix.cpp   READY
HTTP:    http://localhost:8080/
Status:  ready
```

Dans un autre terminal, testez le serveur :

```
curl -i http://127.0.0.1:8080/
```

Forme de réponse attendue :

```
{
  "message": "Hello from Vix",
  "framework": "Vix.cpp"
}
```

Arrêtez le serveur avec :

```
Ctrl+C
```

3.9 Vérifier les commandes Vix utiles

Ces commandes vous aident à inspecter votre environnement :

```
vix doctor
vix info
```

Utilisez `vix doctor` quand vous voulez vérifier si votre chaîne d'outils est saine.

Utilisez `vix info` quand vous voulez inspecter les chemins, les dossiers de cache et les détails d'installation.

3.10 Configuration recommandée de l'éditeur

Vous pouvez utiliser n'importe quel éditeur.

Configuration recommandée :

Outil	Recommandation
Éditeur	VS Code, CLion, Vim, Neovim ou Zed
Compilateur	GCC ou Clang sur Linux/macOS, MSVC ou clang-cl sur Windows
Système de build	CMake
Backend de build	Ninja
Terminal	Bash, Zsh, PowerShell ou Windows Terminal

Pour VS Code, installez :

- l'extension C/C++,
- CMake Tools,
- clangd (optionnel).

3.11 Configuration Git recommandée

Si vous prévoyez de créer de vrais projets, configurez Git :

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

Vérifiez :

```
git config --global --list
```

3.12 Variables d'environnement

Les applications Vix lisent souvent leur configuration depuis un fichier `.env`.

Un fichier `.env` simple peut ressembler à ceci :

```
SERVER_PORT=8080
VIX_LOG_LEVEL=info
VIX_LOG_FORMAT=kv
```

Plus tard, les templates de projet peuvent générer des fichiers `.env` et `.env.example` pour vous.

3.13 Problèmes courants

3.13.1 `vix: command not found`

Votre terminal ne trouve pas le binaire Vix.

Corrigez votre PATH :

```
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

Pour Zsh :

```
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

Puis vérifiez :

```
vix --version
```

3.13.2 `c++: command not found`

Installez un compilateur.

Sur Ubuntu ou Debian :

```
sudo apt update
sudo apt install -y build-essential
```

3.13.3 `cmake: command not found`

Installez CMake :

```
sudo apt install -y cmake
```

3.13.4 `ninja: command not found`

Installez Ninja :

```
sudo apt install -y ninja-build
```

3.13.5 `#include <vix.hpp>` introuvable

Le SDK complet est manquant ou mal installé.

Vérifiez :

```
find ~/.local/include -name vix.hpp 2>/dev/null
```

Si rien n'apparaît, réinstallez le SDK complet :

```
curl -fsSL https://vixcpp.com/install.sh | bash
```

N'utilisez pas le mode CLI-only pour ce guide.

3.13.6 Le port 8080 est déjà utilisé

Trouvez le processus qui utilise le port :

```
sudo lsof -i :8080
```

Arrêtez le processus ou changez le port dans votre code.

3.14 Ce qu'il faut retenir

Un bon environnement Vix dispose de :

- `vix`,
- `c++`,
- `cmake`,
- `ninja`,
- `git`,
- `curl`.

Vérifiez-les avec :

```
vix --version
c++ --version
cmake --version
ninja --version
git --version
curl --version
```

Le test le plus important est :

```
vix run main.cpp
```

Si cela fonctionne, vous êtes prêt à écrire votre premier fichier C++ avec Vix.

3.15 Étape suivante

Exécutez votre premier fichier C++ avec Vix.

Suivant : Exécuter votre premier fichier C++

Chapitre 4

Exécuter votre premier fichier C++

Cette page montre comment exécuter un seul fichier C++ avec Vix.

La commande principale est :

```
vix run main.cpp
```

C'est la façon la plus rapide de commencer à utiliser Vix.

4.1 Créer un espace de travail

Créez un dossier temporaire :

```
mkdir -p ~/tmp/vix-first-file
cd ~/tmp/vix-first-file
```

Créez `main.cpp` :

```
touch main.cpp
```

4.2 Écrire votre premier programme C++

Ouvrez `main.cpp` et écrivez :

```cpp
#include <iostream>

int main()
{
  std::cout << "Hello from Vix\n";
  return 0;
}
```

Exécutez-le :

```
vix run main.cpp
```

Sortie attendue :

```
Hello from Vix
```

4.3 Que s'est-il passé ?

Quand vous exécutez :

```
vix run main.cpp
```

Vix détecte que vous avez passé un seul fichier `.cpp`.

Il :

```
détecte le fichier → le compile → l'exécute → affiche la sortie
```

Vous n'avez pas encore besoin de créer un projet complet.

Ce mode est utile pour :

- les expérimentations rapides,
- apprendre Vix,
- tester de petites idées,
- écrire de petits outils C++,
- exécuter des exemples.

4.4 Exécuter un fichier HTTP Vix

Remplacez maintenant `main.cpp` par une petite application HTTP Vix :

```cpp
#include <vix.hpp>

using namespace vix;

int main()
{
  App app;

  app.get("/", [](Request &, Response &res) {
    res.text("Hello Vix");
  });
```

```
  app.run(8080);

  return 0;
}
```

Exécutez-le :

```
vix run main.cpp
```

Forme de sortie attendue :

```
Vix.cpp   READY
HTTP:    http://localhost:8080/
Status:  ready
```

Ouvrez un autre terminal et testez le serveur :

```
curl -i http://127.0.0.1:8080/
```

Réponse attendue :

```
Hello Vix
```

Arrêtez le serveur avec :

```
Ctrl+C
```

4.5 Retourner du JSON

La plupart des applications backend Vix retournent du JSON.

Mettez la route à jour :

```
#include <vix.hpp>

using namespace vix;

int main()
{
  App app;

  app.get("/", [](Request &, Response &res) {
    res.json({
      "message", "Hello from Vix",
      "framework", "Vix.cpp"
    });
  });
```

```
  app.get("/health", [](Request &, Response &res) {
    res.json({
      "ok", true,
      "service", "first-file"
    });
  });

  app.run(8080);

  return 0;
}
```

Relancez-le :

```
vix run main.cpp
```

Testez-le :

```
curl -i http://127.0.0.1:8080/
curl -i http://127.0.0.1:8080/health
```

4.6 Ajouter un paramètre de route

Les routes Vix peuvent contenir des paramètres de chemin.

Ajoutez cette route avant `app.run(8080)` :

```
app.get("/hello/{name}", [](Request &req, Response &res) {
  const std::string name = req.param("name");

  res.json({
    "greeting", "Hello " + name,
    "powered_by", "Vix.cpp"
  });
});
```

Exécutez :

```
vix run main.cpp
```

Testez :

```
curl -i http://127.0.0.1:8080/hello/Gaspard
```

Forme de réponse attendue :

```
{
  "greeting": "Hello Gaspard",
  "powered_by": "Vix.cpp"
}
```

4.7 Ajouter un paramètre de requête

Les paramètres de requête (query parameters) viennent après le ? dans l'URL.

Ajoutez cette route avant `app.run(8080)` :

```
app.get("/users/{id}", [](Request &req, Response &res) {
  const std::string id = req.param("id");
  const std::string page = req.query_value("page", "1");

  res.json({
    "id", id,
    "page", page
  });
});
```

Exécutez :

```
vix run main.cpp
```

Testez :

```
curl -i "http://127.0.0.1:8080/users/42?page=2"
```

Forme de réponse attendue :

```
{
  "id": "42",
  "page": "2"
}
```

4.8 Exemple complet

Votre `main.cpp` complet peut ressembler à ceci :

```
#include <vix.hpp>

using namespace vix;

int main()
```

```
{
  App app;

  app.get("/", [](Request &, Response &res) {
    res.json({
      "message", "Hello from Vix",
      "framework", "Vix.cpp"
    });
  });

  app.get("/health", [](Request &, Response &res) {
    res.json({
      "ok", true,
      "service", "first-file"
    });
  });

  app.get("/hello/{name}", [](Request &req, Response &res) {
    const std::string name = req.param("name");

    res.json({
      "greeting", "Hello " + name,
      "powered_by", "Vix.cpp"
    });
  });

  app.get("/users/{id}", [](Request &req, Response &res) {
    const std::string id = req.param("id");
    const std::string page = req.query_value("page", "1");

    res.json({
      "id", id,
      "page", page
    });
  });

  app.run(8080);

  return 0;
}
```

Exécutez :

```
vix run main.cpp
```

Testez :

```
curl -i http://127.0.0.1:8080/
curl -i http://127.0.0.1:8080/health
curl -i http://127.0.0.1:8080/hello/Gaspard
curl -i "http://127.0.0.1:8080/users/42?page=2"
```

4.9 Passer des arguments runtime

Les arguments runtime sont les arguments passés à votre programme.

Utilisez `--run` :

```
vix run main.cpp --run --port 8080
```

Utilisez ceci quand votre programme lit `argv`.

Exemple :

```
#include <iostream>

int main(int argc, char **argv)
{
  std::cout << "argc = " << argc << "\n";

  for (int i = 0; i < argc; ++i)
  {
    std::cout << "argv[" << i << "] = " << argv[i] << "\n";
  }

  return 0;
}
```

Exécutez :

```
vix run main.cpp --run --port 8080
```

4.10 Important : `--run` vs `--`

En mode script, `--run` est utilisé pour les arguments runtime.

```
vix run main.cpp --run --port 8080
```

Le séparateur `--` est utilisé pour les flags du compilateur ou du linker.

```
vix run main.cpp -- -O2 -DNDEBUG
```

N'utilisez pas `--` pour les arguments runtime en mode script.

Incorrect :

```
vix run main.cpp -- --port 8080
```

Correct :

```
vix run main.cpp --run --port 8080
```

4.11 Passer des flags de compilation

Utilisez `--` pour passer des flags au compilateur ou au linker :

```
vix run main.cpp -- -O2 -DNDEBUG
```

Lier avec des bibliothèques :

```
vix run main.cpp -- -lssl -lcrypto
```

Ajouter des chemins d'include :

```
vix run main.cpp -- -I./include
```

4.12 Utiliser le mode watch

Pendant le développement, vous pouvez reconstruire et relancer dès que le fichier change :

```
vix run main.cpp --watch
```

Pour le développement d'un projet, vous utiliserez en général :

```
vix dev
```

Vous apprendrez cela dans les pages suivantes.

4.13 Utiliser les sanitizers

Pour le débogage de problèmes mémoire :

```
vix run main.cpp --san
```

Pour les vérifications de comportement indéfini :

```
vix run main.cpp --ubsan
```

4.14 Erreurs courantes

4.14.1 Utiliser l'installation CLI-only

Si votre code utilise :

```
#include <vix.hpp>
```

vous avez besoin du SDK complet.

Vérifiez :

```
find ~/.local/include -name vix.hpp 2>/dev/null
```

Si rien n'apparaît, réinstallez Vix :

```
curl -fsSL https://vixcpp.com/install.sh | bash
```

4.14.2 Passer des arguments runtime après --

Incorrect :

```
vix run main.cpp -- --port 8080
```

Correct :

```
vix run main.cpp --run --port 8080
```

4.14.3 Le port 8080 est déjà utilisé

Si le serveur ne peut pas démarrer, un autre processus utilise déjà le port 8080.

Vérifiez :

```
sudo lsof -i :8080
```

Arrêtez l'autre processus ou changez le port :

```
app.run(3000);
```

4.14.4 Exécution depuis le mauvais répertoire

Les chemins relatifs dépendent du dossier dans lequel vous exécutez `vix`.

Par exemple :

```
res.file("public/index.html");
```

cherchera :

```
public/index.html
```

relativement à votre répertoire courant.

4.15 Quand créer un projet

Un seul fichier est parfait pour apprendre.

Passez à un projet quand vous avez besoin de :

- plusieurs fichiers `.cpp`,
- de headers,
- de tests,
- de dépendances,
- d'une configuration `.env`,
- d'une structure de dossiers stable,
- de release builds.

La page suivante montre comment créer un vrai projet Vix.

4.16 Ce qu'il faut retenir

La commande principale est :

```
vix run main.cpp
```

Utilisez :

- `--run` pour les arguments runtime,
- `--` pour les flags du compilateur et du linker.

Le mode fichier unique est le moyen le plus rapide de commencer.

Quand votre application grandit, créez un projet.

4.17 Étape suivante

Créez votre premier projet Vix.

Suivant : Créer votre premier projet

Chapitre 5

Créer votre premier projet

Cette page montre comment créer votre premier vrai projet Vix.

Jusqu'à présent, vous avez utilisé :

```
vix run main.cpp
```

C'est très bien pour un fichier unique.

Pour une vraie application, utilisez :

```
vix new api
```

Un projet Vix vous fournit une structure propre, un workflow de build, des tests, des fichiers de configuration et une meilleure boucle de développement.

5.1 Créer un projet

Créez un nouveau projet :

```
cd ~/tmp
vix new api
```

Vix vous demandera quel template vous voulez utiliser.

Choisissez :

```
Application
```

La sortie devrait ressembler à ceci :

```
Template
❯ Application
  Library (header-only)

Core
```

```
  ○ ORM
  ○ Sanitizers
  ○ Static C++ runtime

Advanced
  ○ Full static

✔ Project created.
  • Location  : /home/your-user/tmp/api

✔  api  application
──────────────────────────────────────

next
1  cd api/     enter project
2  vix build   compile
3  vix run     start app
```

5.2 Entrer dans le projet

```
cd api
```

5.3 Construire le projet

Exécutez :

```
vix build
```

Forme de sortie attendue :

```
Compiling api (dev)
  build [============================] done
  ✔ Configured
  ✔ Built
  ✔ Done in 1.6s
```

Ceci compile le projet sans démarrer l'application.

5.4 Exécuter le projet

Exécutez :

```
vix run
```

Forme de sortie attendue :

```
Vix.cpp   READY   v2.5.3   run
  › HTTP:    http://localhost:8080/
  i Threads: 8/8
  i Mode:    run
  i Status:  ready
  i Hint:    Ctrl+C to stop the server
```

Ouvrez un autre terminal et testez-le :

```
curl -i http://127.0.0.1:8080/
```

Arrêtez le serveur avec :

```
Ctrl+C
```

5.5 Mode développement

Pour le développement au quotidien, utilisez :

```
vix dev
```

`vix dev` démarre le projet en mode développement.

Utilisez-le quand vous êtes en train d'éditer du code et que vous voulez une boucle de développement plus rapide.

Ouvrez ensuite :

```
http://localhost:8080/
```

5.6 Structure de projet générée

Un projet d'application basique ressemble généralement à ceci :

```
api/
├── CMakeLists.txt
├── CMakePresets.json
├── README.md
├── app.vix
├── vix.json
├── .env
├── .env.example
├── src/
```

```
|   └── main.cpp
└── tests/
    └── test_basic.cpp
```

5.7 Le rôle de chaque fichier

Fichier ou dossier	Rôle
`src/`	Code source de l'application.
`tests/`	Tests du projet.
`CMakeLists.txt`	Configuration de build C++.
`CMakePresets.json`	Presets de build utilisés par CMake et Vix.
`app.vix`	Manifeste de l'application Vix.
`vix.json`	Métadonnées, dépendances et tâches du projet.
`.env`	Configuration runtime locale.
`.env.example`	Exemple de fichier d'environnement.
`README.md`	Documentation du projet.

5.8 Ouvrir le fichier d'entrée

Ouvrez :

```
src/main.cpp
```

Une application minimale générée peut ressembler à ceci :

```cpp
#include <vix.hpp>

using namespace vix;

int main()
{
  App app;

  app.get("/", [](Request &, Response &res) {
    res.send("Hello world");
  });

  app.run(8080);
```

```
  return 0;
}
```

C'est le point d'entrée principal de votre application.

5.9 Modifier la première route

Remplacez la route par une réponse JSON :

```
app.get("/", [](Request &, Response &res) {
  res.json({
    "message", "Hello from your first Vix project",
    "framework", "Vix.cpp"
  });
});
```

Ajoutez une route de health :

```
app.get("/health", [](Request &, Response &res) {
  res.json({
    "ok", true,
    "service", "api"
  });
});
```

Votre `src/main.cpp` complet peut ressembler à ceci :

```
#include <vix.hpp>

using namespace vix;

int main()
{
  App app;

  app.get("/", [](Request &, Response &res) {
    res.json({
      "message", "Hello from your first Vix project",
      "framework", "Vix.cpp"
    });
  });

  app.get("/health", [](Request &, Response &res) {
```

```
    res.json({
      "ok", true,
      "service", "api"
    });
  });

  app.run(8080);

  return 0;
}
```

Exécutez :

```
vix dev
```

Testez :

```
curl -i http://127.0.0.1:8080/
curl -i http://127.0.0.1:8080/health
```

5.10 Utiliser `.env`

Les projets Vix peuvent utiliser des fichiers `.env` pour la configuration locale.

Exemple :

```
SERVER_PORT=8080
VIX_LOG_LEVEL=info
VIX_LOG_FORMAT=kv
```

Dans le code, vous pouvez charger la configuration :

```
#include <vix.hpp>

using namespace vix;

int main()
{
  config::Config cfg{".env"};

  App app;

  app.get("/", [](Request &, Response &res) {
    res.json({
      "message", "Hello from Vix",
```

```
      "framework", "Vix.cpp"
    });
  });

  app.run(cfg.getServerPort());

  return 0;
}
```

Maintenant, le port du serveur vient du fichier `.env`.

5.11 Commandes utiles du projet

À l'intérieur du dossier de projet :

```
vix build
vix run
vix dev
vix check
vix tests
vix fmt
```

Commande	Rôle
`vix build`	Compiler le projet.
`vix run`	Construire et démarrer l'application.
`vix dev`	Démarrer le mode développement.
`vix check`	Valider le projet.
`vix tests`	Exécuter les tests.
`vix fmt`	Formater les fichiers sources.

5.12 Tâches de projet

Certains projets définissent des tâches dans `vix.json`.

Vous pouvez les exécuter avec :

```
vix task dev
vix task test
vix task check
```

Un fichier `vix.json` peut contenir des tâches comme celles-ci :

```
{
  "name": "api",
  "deps": [],
  "vars": {
    "preset": "dev-ninja",
    "release_preset": "release"
  },
  "tasks": {
    "dev": {
      "description": "Start dev mode",
      "command": "vix dev"
    },
    "test": {
      "description": "Run tests",
      "command": "vix tests --preset ${preset}"
    },
    "check": {
      "description": "Validate project",
      "command": "vix check --preset ${preset} --tests"
    }
  }
}
```

5.13 Application vs bibliothèque

Quand vous exécutez :

```
vix new api
```

Vix peut créer différents types de projets.

Pour ce guide, choisissez :

```
Application
```

Utilisez Application quand vous voulez construire une application, un serveur, une API ou un service exécutable.

Utilisez Library (header-only) quand vous voulez construire une bibliothèque C++ réutilisable.

Exemple :

```
vix new tree
```

Puis choisissez :

```
Library (header-only)
```

Un projet de bibliothèque est utile quand vous voulez créer des headers et des tests réutilisables, mais ce n'est pas le chemin principal de ce guide Bien démarrer.

5.14 Erreurs courantes

5.14.1 Exécuter des commandes en dehors du projet

Incorrect :

```
cd ~/tmp
vix dev
```

Correct :

```
cd ~/tmp/api
vix dev
```

Exécutez les commandes de projet depuis le dossier du projet.

5.14.2 Oublier d'arrêter le serveur précédent

Si le port 8080 est déjà utilisé, arrêtez le serveur précédent avec :

```
Ctrl+C
```

Ou trouvez le processus :

```
sudo lsof -i :8080
```

5.14.3 Modifier les fichiers sans reconstruire

Quand vous utilisez :

```
vix run
```

vous devrez peut-être redémarrer manuellement après les modifications.

Pour le développement actif, utilisez :

```
vix dev
```

5.14.4 Ajouter de nouveaux fichiers `.cpp`

Si vous ajoutez de nouveaux fichiers sources, assurez-vous qu'ils font partie du build.

Pour un projet CMake, mettez à jour `CMakeLists.txt`.

Exemple :

```
add_executable(api
  src/main.cpp
  src/routes.cpp
)
```

5.15 Ce qu'il faut retenir

Créez un projet d'application :

```
vix new api
cd api
```

Construisez-le :

```
vix build
```

Exécutez-le :

```
vix run
```

Développez-le :

```
vix dev
```

Un projet Vix est le moment où une expérimentation rapide devient une vraie application.

5.16 Étape suivante

Construisez votre premier serveur HTTP avec Vix.

Suivant : Votre premier serveur HTTP

Chapitre 6

Votre premier serveur HTTP

Cette page montre comment construire votre premier serveur HTTP avec Vix.cpp.

Vous allez créer :

```
GET /
GET /health
GET /hello/{name}
GET /users/{id}
```

L'objectif est de comprendre le modèle HTTP central de Vix :

```
App → route → Request → Response → app.run()
```

6.1 Partir de votre projet

Utilisez le projet créé à la page précédente :

```
cd ~/tmp/api
```

Ouvrez :

```
src/main.cpp
```

Remplacez son contenu par ce serveur minimal :

```
#include <vix.hpp>

using namespace vix;

int main()
{
  App app;
```

```
  app.get("/", [](Request &, Response &res) {
    res.send("Hello world");
  });

  app.run(8080);

  return 0;
}
```

Exécutez-le :

```
vix dev
```

Ouvrez un autre terminal et testez-le :

```
curl -i http://127.0.0.1:8080/
```

Corps de réponse attendu :

```
Hello world
```

6.2 Ce que fait ce code

```
App app;
```

Crée l'application Vix.

```
app.get("/", [](Request &, Response &res) {
  res.send("Hello world");
});
```

Enregistre une route `GET /`.

```
app.run(8080);
```

Démarre le serveur HTTP sur le port `8080`.

6.3 Concepts fondamentaux

Élément	Rôle
`#include <vix.hpp>`	Importe l'API principale de Vix.
`using namespace vix;`	Permet d'utiliser `App`, `Request` et `Response` directement.
`App app;`	Crée l'application HTTP.
`app.get(...)`	Enregistre une route GET.

Élément	Rôle
`Request &req`	Lit ce que le client a envoyé.
`Response &res`	Envoie ce que votre application retourne.
`app.run(8080)`	Démarre le serveur.

6.4 Retourner du JSON

La plupart des services backend retournent du JSON.

Remplacez la route `/` par :

```
app.get("/", [](Request &, Response &res) {
  res.json({
    "message", "Hello from Vix",
    "framework", "Vix.cpp"
  });
});
```

Exécutez :

```
vix dev
```

Testez :

```
curl -i http://127.0.0.1:8080/
```

Forme de réponse attendue :

```
{
  "message": "Hello from Vix",
  "framework": "Vix.cpp"
}
```

6.5 Ajouter une route de santé

Une route de santé est utile pour vérifier que votre serveur est en vie.

Ajoutez ceci avant `app.run(8080)` :

```
app.get("/health", [](Request &, Response &res) {
  res.json({
    "ok", true,
    "service", "api"
  });
```

```
});
```

Testez :

```
curl -i http://127.0.0.1:8080/health
```

Forme de réponse attendue :

```
{
  "ok": true,
  "service": "api"
}
```

6.6 Ajouter un paramètre de chemin

Les paramètres de chemin (path parameters) vous permettent de lire des valeurs depuis l'URL.

Ajoutez cette route :

```
app.get("/hello/{name}", [](Request &req, Response &res) {
  const std::string name = req.param("name");

  res.json({
    "greeting", "Hello " + name,
    "powered_by", "Vix.cpp"
  });
});
```

Testez :

```
curl -i http://127.0.0.1:8080/hello/Gaspard
```

Forme de réponse attendue :

```
{
  "greeting": "Hello Gaspard",
  "powered_by": "Vix.cpp"
}
```

Ici :

```
req.param("name")
```

lit la partie `{name}` de la route :

```
/hello/{name}
```

6.7 Ajouter une route avec un identifiant

Ajoutez une autre route :

```
app.get("/users/{id}", [](Request &req, Response &res) {
  const std::string id = req.param("id");

  res.json({
    "ok", true,
    "id", id
  });
});
```

Testez :

```
curl -i http://127.0.0.1:8080/users/42
```

Forme de réponse attendue :

```
{
  "ok": true,
  "id": "42"
}
```

6.8 Ajouter des paramètres de requête

Les paramètres de requête (query parameters) viennent après le `?` dans l'URL.

Mettez à jour la route `/users/{id}` :

```
app.get("/users/{id}", [](Request &req, Response &res) {
  const std::string id = req.param("id");
  const std::string page = req.query_value("page", "1");
  const std::string limit = req.query_value("limit", "10");

  res.json({
    "ok", true,
    "id", id,
    "page", page,
    "limit", limit
  });
});
```

Testez :

```
curl -i "http://127.0.0.1:8080/users/42?page=2&limit=20"
```

Forme de réponse attendue :

```
{
  "ok": true,
  "id": "42",
  "page": "2",
  "limit": "20"
}
```

6.9 Méthodes de réponse

Vix vous offre plusieurs helpers de réponse :

```
res.send("Hello world");
res.text("Hello Vix");
res.json({"ok", true});
res.status(201).json({"ok", true});
res.header("Cache-Control", "no-cache");
res.file("public/index.html");
```

Utilisez :

Méthode	Quand l'utiliser
`res.send(...)`	Vous voulez une réponse générique.
`res.text(...)`	Vous voulez du texte brut.
`res.json(...)`	Vous voulez du JSON.
`res.status(...).json(...)`	Vous voulez fixer le statut HTTP.
`res.header(...)`	Vous voulez fixer un header de réponse.
`res.file(...)`	Vous voulez envoyer un fichier.

6.10 Exemple complet

Votre `src/main.cpp` complet peut désormais ressembler à ceci :

```
#include <vix.hpp>

using namespace vix;

int main()
{
```

```
App app;

app.get("/", [](Request &, Response &res) {
  res.json({
    "message", "Hello from Vix",
    "framework", "Vix.cpp"
  });
});

app.get("/health", [](Request &, Response &res) {
  res.json({
    "ok", true,
    "service", "api"
  });
});

app.get("/hello/{name}", [](Request &req, Response &res) {
  const std::string name = req.param("name");

  res.json({
    "greeting", "Hello " + name,
    "powered_by", "Vix.cpp"
  });
});

app.get("/users/{id}", [](Request &req, Response &res) {
  const std::string id = req.param("id");
  const std::string page = req.query_value("page", "1");
  const std::string limit = req.query_value("limit", "10");

  res.json({
    "ok", true,
    "id", id,
    "page", page,
    "limit", limit
  });
});

app.run(8080);
```

```
  return 0;
}
```

Exécutez :

```
vix dev
```

Testez toutes les routes :

```
curl -i http://127.0.0.1:8080/
curl -i http://127.0.0.1:8080/health
curl -i http://127.0.0.1:8080/hello/Gaspard
curl -i "http://127.0.0.1:8080/users/42?page=2&limit=20"
```

6.11 Organiser les routes avec des fonctions

Quand votre application grandit, évitez de tout mettre directement dans `main()`.

Vous pouvez organiser les routes comme ceci :

```
#include <vix.hpp>

using namespace vix;

static void register_public_routes(App &app)
{
  app.get("/", [](Request &, Response &res) {
    res.json({
      "message", "Hello from Vix",
      "framework", "Vix.cpp"
    });
  });

  app.get("/health", [](Request &, Response &res) {
    res.json({
      "ok", true,
      "service", "api"
    });
  });
}

static void register_user_routes(App &app)
{
  app.get("/users/{id}", [](Request &req, Response &res) {
```

```
    const std::string id = req.param("id");
    const std::string page = req.query_value("page", "1");

    res.json({
      "ok", true,
      "id", id,
      "page", page
    });
  });
}

int main()
{
  App app;

  register_public_routes(app);
  register_user_routes(app);

  app.run(8080);

  return 0;
}
```

Cela garde `main()` petit et rend l'application plus facile à maintenir.

6.12 Erreurs courantes

6.12.1 Oublier de lancer le serveur

Les routes ne font rien tant que vous n'appelez pas :

```
app.run(8080);
```

6.12.2 Oublier de retourner après une erreur

Quand vous envoyez une réponse d'erreur, retournez immédiatement.

```
app.get("/users/{id}", [](Request &req, Response &res) {
  const std::string id = req.param("id");

  if (id == "0")
  {
    res.status(404).json({
```

```
      "ok", false,
      "error", "user not found"
    });

    return;
  }

  res.json({
    "ok", true,
    "id", id
  });
});
```

6.12.3 Le port 8080 est déjà utilisé

Si le serveur ne peut pas démarrer, un autre processus utilise déjà le port `8080`.

Vérifiez :

```
sudo lsof -i :8080
```

Arrêtez le processus ou changez le port :

```
app.run(3000);
```

6.12.4 Exécution depuis le mauvais dossier

Exécutez les commandes de projet depuis l'intérieur de votre projet :

```
cd ~/tmp/api
vix dev
```

6.13 Ce qu'il faut retenir

Le modèle HTTP basique de Vix est :

```
App → route → Request → Response → app.run()
```

Le serveur minimal est :

```
#include <vix.hpp>

using namespace vix;

int main()
{
```

```
  App app;

  app.get("/", [](Request &, Response &res) {
    res.send("Hello world");
  });

  app.run(8080);

  return 0;
}
```

Utilisez :

```
vix dev
```

pendant le développement.

Utilisez :

```
vix run
```

quand vous voulez démarrer l'application directement.

6.14 Lectures suivantes

Maintenant que vous avez un serveur HTTP fonctionnel, continuez avec :

- Le Livre Vix
- Routes
- Requête et réponse
- Construire une API REST
- Templates

partie II

Le Livre Vix

Chapitre 7

Introduction

Bienvenue dans le livre Vix.

Ce livre enseigne Vix étape par étape, comme une histoire. Vous commencerez par l'idée la plus simple :

```
Exécuter du code C++ rapidement.
```

Puis vous grandirez vers de vrais systèmes backend :

- API HTTP,
- JSON,
- middleware,
- validation,
- base de données,
- WebSocket,
- runtime asynchrone,
- cache,
- synchronisation offline-first,
- P2P,
- et déploiement en production.

L'objectif n'est pas seulement de montrer des commandes ou des APIs. L'objectif est de vous aider à comprendre le modèle mental qui sous-tend Vix.

7.1 Qu'est-ce que Vix ?

Vix est un runtime C++ moderne pour construire des applications rapides et fiables. Il offre au C++ une expérience de développement plus directe :

```
vix run main.cpp
```

ou :

```
vix new api
cd api
vix dev
```

Au lieu de vous forcer à configurer manuellement tout l'environnement avant d'écrire votre première ligne de code utile, Vix vous propose un workflow orienté runtime. Vous écrivez l'application. Vix se charge de la boucle de développement.

7.2 L'idée principale

Le C++ est puissant. Mais construire de vraies applications en C++ requiert souvent trop de configuration avant même que le développeur puisse commencer. Vous devez parfois réfléchir à CMake, aux flags du compilateur, aux flags du linker, aux dépendances, aux dossiers de build, aux arguments runtime, aux logs, aux tests, au démarrage du serveur, aux flags de base de données et à la structure du projet.

Vix essaie de rendre tout cela plus fluide. L'idée est simple :

```
Le C++ doit pouvoir être direct sans perdre sa puissance.
```

Vix ne remplace pas le C++. Vix offre au C++ un meilleur runtime applicatif tout autour de lui.

7.3 Pourquoi un runtime ?

Un langage seul ne suffit pas pour construire confortablement des applications modernes.

JavaScript est devenu largement utilisé pour le développement backend non seulement à cause du langage, mais parce que Node.js a donné aux développeurs un runtime, un workflow de packages et une boucle de feedback rapide.

Python est devenu populaire pour les scripts et le backend parce qu'exécuter un fichier est simple :

```
python app.py
node server.js
```

Vix apporte ce type de workflow au C++ :

```
vix run main.cpp
```

Mais avec la performance, le contrôle et l'explicité du C++.

7.4 Ce que Vix n'est pas

Vix n'est pas un remplacement du C++. Ce n'est pas un langage avec ramasse-miettes. Il n'essaie pas de cacher comment fonctionnent les systèmes ou de transformer le C++ en JavaScript ou en Python.

Vix est un runtime et une boîte à outils qui rendent le développement d'applications C++ plus pratique. Il garde la philosophie C++ — explicite, rapide, déterministe, proche du système — mais ajoute un workflow plus fluide.

7.5 Un petit exemple

```cpp
#include <vix.hpp>
using namespace vix;

int main()
{
  App app;

  app.get("/", [](Request &, Response &res){
    res.json({
      "message", "Hello from Vix",
      "framework", "Vix.cpp"
    });
  });

  app.run(8080);
  return 0;
}
```

Exécutez-le :

```
vix run main.cpp
```

Puis ouvrez `http://localhost:8080`.

Ce petit exemple montre déjà le style central de Vix :

- `App`,
- route,
- `Request`,
- `Response`,
- JSON,
- runtime.

7.6 L'expérience de développement

```
vix new api    # Créer un projet
vix dev        # Lancer en mode développement
vix build      # Construire
vix check      # Valider
vix tests      # Lancer les tests
vix fmt        # Formater le code
```

7.7 La structure de ce livre

Partie 1 : Comprendre Vix Introduction, Pourquoi Vix, Modèle mental

Partie 2 : Commencer à construire Installation, Exécuter votre premier fichier, Créer votre premier projet, Premier serveur HTTP

Partie 3 : Construire des APIs Routes, Requête et réponse, API JSON

Partie 4 : Ajouter des couches professionnelles Middleware, Validation, Erreurs et journalisation

Partie 5 : Connecter de vrais systèmes Base de données, WebSocket en temps réel, Runtime asynchrone

Partie 6 : Fonctionnalités runtime avancées Cache, Synchronisation offline-first, P2P

Partie 7 : Production Déploiement en production, Étapes suivantes

7.8 Comment lire ce livre

Lisez-le dans l'ordre la première fois. Chaque chapitre s'appuie sur le précédent.

Vous n'avez besoin que de connaissances C++ de base :

- les fonctions,
- les classes,
- les headers,
- `std::string`,
- `std::vector`,
- les lambdas,
- et une notion de base de CMake.

7.9 Le changement mental principal

Le développement C++ traditionnel commence souvent par le système de build. Vix commence par l'application.

Au lieu de penser d'abord à comment configurer, lier et construire, Vix veut que la première question soit :

```
Qu'est-ce que je veux construire ?
```

7.10 Vix et la production

Une application Vix peut tourner comme un service Linux normal :

```
navigateur → Nginx → application Vix → systemd
```

Le chapitre sur la production montrera comment déployer avec un build release, systemd, Nginx, TLS, des logs et des health checks.

7.11 Ce qu'il faut retenir

Vix est un runtime C++ moderne pour construire des applications rapides et fiables. Il offre au C++ un workflow d'exécution direct, un workflow de projet, un modèle d'application HTTP, des API JSON, du middleware, de la validation, l'accès à une base de données, le support WebSocket, des outils orientés runtime et un chemin vers le déploiement en production.

L'idée centrale : garder la puissance du C++, rendre le workflow applicatif plus simple.

7.12 Chapitre suivant

Suivant : Pourquoi Vix

Chapitre 8

Pourquoi Vix existe

Vix existe parce que construire de vraies applications en C++ devrait être plus rapide, plus clair et plus pratique.

Le C++ offre : la performance, le contrôle, des durées de vie déterministes, l'accès bas niveau, des types forts, des binaires natifs et une puissance au niveau système. Mais pour de nombreux développeurs, l'expérience de construire des applications en C++ semble plus lourde qu'elle ne devrait l'être.

8.1 Le problème

Pour construire un simple backend, vous devez parfois réfléchir à la configuration du compilateur, à la configuration CMake, à l'installation des dépendances, aux chemins d'include, aux flags du linker, aux dossiers de build, aux arguments runtime, au cycle de vie du serveur, à la journalisation, au JSON, au routage HTTP, à la configuration de la base de données, aux tests et au déploiement.

Chaque pièce est gérable séparément. Le problème, c'est qu'il faut tout câbler avant que l'application ne devienne utile.

8.2 La première étape devrait être simple

Dans de nombreux écosystèmes, la première étape est directe :

```
python app.py
node server.js
deno run server.ts
```

Vix offre au C++ un point de départ similaire :

```
vix run main.cpp
```

L'objectif n'est pas de copier ces écosystèmes. L'objectif est de rendre le C++ plus direct pour le développement d'applications.

8.3 Le C++ ne manque pas de puissance

Le problème n'est pas le langage lui-même. Le problème, c'est le workflow applicatif manquant autour du langage. Le C++ a déjà d'excellents compilateurs, des performances exceptionnelles, RAII, les templates, des abstractions à coût nul, de la concurrence native, des outils matures et des binaires portables. Ce qui manque souvent aux développeurs, c'est une expérience de runtime unifiée.

8.4 La couche manquante

Quand vous construisez un backend en C++, vous combinez généralement de nombreux éléments :

- une bibliothèque HTTP,
- une bibliothèque JSON,
- un système de build,
- une bibliothèque de journalisation,
- une couche base de données,
- une logique de middleware,
- une couche de validation,
- une bibliothèque WebSocket,
- et des scripts de déploiement.

Cela crée de la friction. Vix répond à ces questions avec un workflow unique et cohérent.

8.5 L'approche Vix

```
vix run main.cpp          # Exécuter un fichier
vix new api && vix dev    # Créer et lancer un projet
vix build                 # Construire
vix check && vix tests    # Valider et tester
```

8.6 Pourquoi pas simplement utiliser CMake ?

Vix ne remplace pas CMake. CMake répond à : « Comment configurer et construire ce projet C++ ? »

Vix répond à : « Comment construire, exécuter, tester, développer, packager et exploiter cette application C++ ? »

Vix peut utiliser CMake en interne tout en offrant une expérience plus simple au niveau supérieur.

8.7 Pourquoi pas simplement utiliser un framework web C++ ?

Un framework web se concentre généralement sur le HTTP. Vix est plus large.

Vix inclut également le workflow applicatif environnant :

- la CLI,
- la création de projets,
- l'exécution directe de fichiers,
- les commandes de build,
- le workflow de dépendances,
- les tests,
- le formatage,
- la journalisation,
- l'accès à la base de données,
- le runtime WebSocket,
- le middleware,
- la validation,
- le cache,
- la synchronisation,
- P2P,
- et un chemin de déploiement.

8.8 L'application avant tout

La configuration traditionnelle d'un projet C++ commence souvent par la configuration du build. Vix veut que la première étape soit : écrire l'application, l'exécuter.

```
#include <vix.hpp>
using namespace vix;

int main()
{
  App app;
```

```
  app.get("/", [](Request &, Response &res){
    res.text("Hello Vix");
  });

  app.run(8080);
  return 0;
}
```

```
vix run main.cpp
```

8.9 Vix est explicite

Une route est explicite :

```
app.get("/users/{id}", [](Request &req, Response &res){
  const std::string id = req.param("id");
  res.json({"id", id});
});
```

Une requête de base de données est explicite :

```
auto stmt = conn->prepare("SELECT id, name FROM users WHERE id = ?");
stmt->bind(1, id);
auto rows = stmt->query();
```

C'est important parce que les systèmes sérieux doivent être compréhensibles. Vix doit réduire la friction sans cacher le comportement.

8.10 Pourquoi la fiabilité compte

Vix est également connecté à une idée plus grande : les applications doivent continuer à fonctionner dans des conditions du monde réel.

Les vrais systèmes font face à des pannes réseau, des redémarrages de processus, des serveurs lents, des écritures partielles, des timeouts, des clients hors ligne, des reprises (retries), des requêtes dupliquées et des connexions perdues.

C'est pourquoi Vix inclut des idées runtime plus profondes comme le cache, la synchronisation offline-first, le WAL, l'outbox, le retry et le P2P.

8.11 Ce qu'il faut retenir

Vix existe parce que le C++ mérite un runtime applicatif plus fluide. Le problème central n'est pas que le C++ soit faible. Le problème, c'est que construire des

applications en C++ commence souvent avec trop de friction.

Vix fournit :

- l'exécution directe,
- un workflow de projet,
- un modèle d'application HTTP,
- le support JSON,
- du middleware,
- de la validation,
- l'accès à une base de données,
- un runtime WebSocket,
- un chemin de déploiement en production,
- et des modules avancés de fiabilité.

L'idée centrale : garder la puissance du C++, rendre la construction d'applications avec lui directe.

8.12 Chapitre suivant

Suivant : Modèle mental

Chapitre 9

Modèle mental

Ce chapitre explique comment penser Vix.

Avant d'apprendre toutes les commandes et tous les modules, vous avez besoin d'une image claire :

```
Vix est un workflow runtime pour les applications C++.
```

Il connecte quatre couches :

```
CLI
  ↓
Runtime
  ↓
Application
  ↓
Modules
```

La CLI vous aide à travailler. Le runtime exécute votre code. La couche application expose les APIs telles que `App`, `Request` et `Response`. Les modules ajoutent des capacités comme JSON, base de données, middleware, validation, WebSocket, cache, synchronisation et P2P.

9.1 Le modèle mental le plus simple

```cpp
#include <vix.hpp>
using namespace vix;

int main()
{
  App app;
```

```
  app.get("/", [](Request &, Response &res){
    res.text("Hello Vix");
  });

  app.run(8080);
  return 0;
}
```

```
vix run main.cpp
```

Cet exemple contient l'essentiel du modèle mental de Vix :

- `vix run` → workflow CLI,
- `App` → objet application,
- `app.get` → enregistrement de route,
- `Request` → requête entrante,
- `Response` → réponse sortante,
- `app.run` → le runtime démarre le serveur.

9.2 Couche 1 : La CLI

La CLI est le point d'entrée du développeur. Elle fournit des commandes telles que : `vix run`, `vix new`, `vix dev`, `vix build`, `vix check`, `vix tests`, `vix fmt`, `vix doctor`.

Elle offre une boucle de développement cohérente : créer → exécuter → éditer → recharger → vérifier → tester → construire → déployer.

Commande	Rôle
`vix run`	Construit et exécute un fichier, un projet ou un manifeste.
`vix dev`	Démarre une boucle de développement avec rechargement à chaud.
`vix build`	Configure, compile et lie le projet.
`vix check`	Valide builds, tests et sanitizers.

9.3 Couche 2 : Le runtime

Le runtime est ce qui exécute réellement l'application. Dans un programme HTTP simple :

```
app.run(8080);
```

Pour des applications avancées avec HTTP + WebSocket ensemble :

```
struct Runtime
{
  vix::config::Config config{".env"};

  std::shared_ptr<vix::executor::RuntimeExecutor> executor{
      std::make_shared<vix::executor::RuntimeExecutor>(1u)
  };

  vix::App app{executor};
  vix::websocket::Server ws{config, executor};
};

vix::run_http_and_ws(runtime.app, runtime.ws, runtime.executor, http_port);
```

9.4 Couche 3 : L'application

```
App app;

app.get("/users/{id}", [](Request &req, Response &res){
  const std::string id = req.param("id");
  res.json({"id", id});
});
```

Request est l'entrée en lecture seule provenant du client :

- paramètres de chemin,
- paramètres de requête,
- headers,
- corps,
- corps JSON.

Response est la façon dont la route renvoie la sortie :

- texte,
- JSON,
- fichiers,
- codes de statut,
- headers.

9.4.1 Gardez `main()` petit

```
int main()
{
  App app;

  app.get("/", [](Request &, Response &res){
    res.json({"message", "Hello"});
  });

  app.run(8080);
  return 0;
}
```

9.5 Couche 4 : Les modules

```
#include <vix.hpp>           // cœur
#include <vix/db.hpp>        // base de données
#include <vix/websocket.hpp> // WebSocket
#include <vix/middleware.hpp>// middleware
#include <vix/validation.hpp>// validation
```

Utilisez le plus petit header de module public qui donne la fonctionnalité dont vous avez besoin.

9.5.1 Modules clés

Middleware s'exécute autour des routes — CORS, rate limiting, authentification, headers de sécurité, fichiers statiques.

Validation vérifie l'entrée avant la logique métier :

```
auto result = vix::validation::validate("email", email).required().email().result();
```

Base de données offre un accès explicite :

```
auto db = vix::db::Database::sqlite("vix.db");
vix::db::PooledConn conn(db.pool());
auto stmt = conn->prepare("SELECT id, name FROM users WHERE id = ?");
stmt->bind(1, id);
```

WebSocket ajoute des événements temps réel :

```
ws.on_typed_message([&ws](auto &, const std::string &type, const auto &payload){
  if (type == "chat.message") ws.broadcast_json("chat.message", payload);
});
```

9.6 Configuration

```
vix::config::Config cfg{".env"};
const int port = cfg.getServerPort();
vix::db::Database db{cfg};
```

Variables d'environnement :

```
SERVER_PORT=8080
DATABASE_ENGINE=sqlite
DATABASE_DEFAULT_NAME=vix.db
```

9.7 Cycle de vie d'une requête

```
requête client → Nginx (production) → runtime Vix → middleware → handler de route → Response
```

Avec base de données : requête → middleware → route → validation → requête base de données → retour JSON

9.8 Flux d'erreur

```
échec de validation   → 400 Bad Request
token manquant        → 401 Unauthorized
non autorisé          → 403 Forbidden
ressource manquante   → 404 Not Found
trop de requêtes      → 429 Too Many Requests
```

9.9 Croissance d'une application

```
Début :  src/main.cpp

Plus tard : src/
            ├── main.cpp
            ├── routes/
            ├── validation/
```

```
            ├── database/
            └── services/
```

Commencez petit. Déplacez le code dans des modules lorsque cela le justifie. Gardez `main()` comme câblage.

9.10 Forme en production

```
navigateur → Nginx → application Vix sur localhost → systemd
```

9.11 Ce qu'il faut retenir

Le modèle mental Vix a quatre couches : **CLI**, **Runtime**, **Application** et **Modules**.

La **CLI** contrôle le workflow du développeur. Le **Runtime** exécute l'application. La couche **Application** est construite autour de `App`, `Request` et `Response`. Les **Modules** ajoutent des capacités comme JSON, middleware, validation, base de données, WebSocket, cache, synchronisation et P2P.

La meilleure façon de faire grandir une application Vix est simple :

```
commencer avec un fichier → garder main() petit → enregistrer les routes via des fonctions → a
```

Suivant : Routes

Chapitre 10

Routes

Les routes sont le cœur d'une application HTTP Vix. Elles relient une requête HTTP à du code C++ :

```
GET /users/42  →  app.get("/users/{id}", handler);
```

10.1 Anatomie d'une route

```
app.get("/", [](Request &req, Response &res){
  res.text("Hello");
});
```

Élément	Rôle
`app.get`	Méthode HTTP.
`"/"`	Motif de chemin.
Lambda	Handler.
`Request &req`	Requête entrante.
`Response &res`	Réponse sortante.

10.2 Méthodes HTTP

```
app.get("/users", list_handler);          // lecture
app.post("/users", create_handler);       // création
app.put("/users/{id}", replace_handler);  // remplacement
app.patch("/users/{id}", update_handler); // mise à jour partielle
app.del("/users/{id}", delete_handler);   // suppression
```

10.3 Paramètres de chemin

```
app.get("/users/{id}", [](Request &req, Response &res){
  const std::string id = req.param("id");
  res.json({"id", id});
});
```

Correspond à : `/users/1`, `/users/42`, `/users/abc`.

10.3.1 Plusieurs paramètres de chemin

```
app.get("/posts/{year}/{slug}", [](Request &req, Response &res){
  res.json({"year", req.param("year"), "slug", req.param("slug")});
});
```

10.4 Paramètres de requête

Les query params ne font PAS partie du motif de route — ils viennent après `?` :

```
app.get("/users", [](Request &req, Response &res){
  const std::string page = req.query_value("page", "1");
  const std::string limit = req.query_value("limit", "20");
  res.json({"page", page, "limit", limit});
});
```

```
curl -i "http://127.0.0.1:8080/users?page=2&limit=10"
```

10.5 Paramètres de chemin vs paramètres de requête

```
Paramètres de chemin  → identifient la ressource : /users/42
Paramètres de requête → modifient la lecture :    /users?page=2&limit=10
```

10.6 Routes d'erreur

```
app.get("/users/{id}", [](Request &req, Response &res){
  const std::string id = req.param("id");

  if (id == "0"){
    es.status(404).json({"ok", false, "error", "user not found"});
    return;    // retournez toujours après une erreur
```

```
  }

  res.json({"ok", true, "id", id});
});
```

> **Note éditoriale.** Dans le code ci-dessus, on lit `es.status(...)` au lieu de `res.status(...)`. Cette coquille provient de la source originale et a été conservée à l'identique pour respecter la fidélité ; à l'usage, il faut bien sûr lire `res.status(404).json(...)`.

10.7 Organiser les routes par fonctionnalité

```
static void public_routes(App &app) { /* / and /health */ }
static void user_routes(App &app) { /* /users */ }
static void auth_routes(App &app) { /* /auth/login */ }
static void admin_routes(App &app) { /* /admin */ }

int main()
{
  App app;

  public_routes(app);
  user_routes(app);
  auth_routes(app);
  admin_routes(app);

  app.run(8080);

  return 0;
}
```

10.8 L'ordre des routes compte

Les routes sont matchées dans l'ordre d'enregistrement.

Les routes spécifiques doivent être enregistrées avant les routes génériques de fallback.

```
// Correct
app.get("/users/search", search_handler);
app.get("/users/{id}", user_handler);
```

```
app.get("/*", fallback_handler);

// Incorrect — le wildcard intercepte tout
app.get("/*", fallback_handler);
app.get("/users/{id}", user_handler);

// Dans la version incorrecte, la route wildcard peut intercepter les requêtes
// avant que /users/{id} n'ait l'occasion de s'exécuter.
```

10.9 Routes wildcard

Une route wildcard correspond à de nombreux chemins et est utile comme fallback.

```
app.get("/*", [](Request &req, Response &res){
  res.json({"path", req.path()});
});
```

Utilisée pour :

- fallback de fichiers statiques,
- fallback SPA,
- comportement 404 personnalisé.

Utilisée pour : fallback SPA, fallback de fichiers statiques, 404 personnalisée.

10.10 Fallback de fichiers statiques

```
#include <vix.hpp>
using namespace vix;

int main()
{
  App app;

  app.get("/*", [](Request &req, Response &res){
    std::string path = "public" + req.path();

    res.header("Cache-Control", "public, max-age=86400");
    res.file(path);

  });
```

```
  app.run(8080);
}
```

Avec cette structure :

```
project/
├── main.cpp
└── public/
    └── index.html
```

Une requête vers :

```
/index.html
```

sert :

```
public/index.html
```

Exécutez :

```
vix run main.cpp
curl http://localhost:8080/index.html
```

10.11 Fallback SPA

```
app.get("/api/users", users_handler);
app.static_dir("public");

app.get("/*", [](Request &, Response &res){
  res.file("public/index.html");
});
```

10.12 Séparer les routes dans plusieurs fichiers

Header (`src/routes/PublicRoutes.hpp`) :

```
#pragma once
#include <vix.hpp>
void public_routes(vix::App &app);
```

Source (`src/routes/PublicRoutes.cpp`) :

```
#include "PublicRoutes.hpp"

void public_routes(vix::App &app)
{
```

```
  app.get("/", [](vix::Request &, vix::Response &res){
    res.json({"message", "Hello"});
  });
}
```

Mettez à jour `CMakeLists.txt` :

```
add_executable(app src/main.cpp src/routes/PublicRoutes.cpp)
```

10.13 Exemple complet

```
#include <vix.hpp>
using namespace vix;

static void public_routes(App &app)
{
  app.get("/", [](Request &, Response &res){
    res.json({"message", "Vix routes example"});
  });

  app.get("/health", [](Request &, Response &res){
    res.json({"ok", true, "service", "routes-example"});
  });
}

static void user_routes(App &app)
{
  app.get("/users", [](Request &req, Response &res){
    res.json({
      "ok", true,
      "page", req.query_value("page", "1"),
      "items", vix::json::array({"Alice", "Bob"})
    });
  });

  app.get("/users/{id}", [](Request &req, Response &res){
    const std::string id = req.param("id");

    if (id == "0") {
      res.status(404).json({
        "ok", false, "error",
```

```
        "not found"
      });
      return;
    }

    res.json({
      "ok", true,
      "user", vix::json::o("id", id),
      "name", "Ada"
    });

  });

  app.post("/users", [](Request &req, Response &res){
    res.status(201).json({
      "ok", true,
      "body", req.json()
    });
  });

}

int main()
{
  App app;

  public_routes(app);
  user_routes(app);

  app.run(8080);

  return 0;
}
```

10.14 Erreurs courantes

10.14.1 Barre oblique manquante

```
// Incorrect
app.get("health", handler);
```

```
// Correct
app.get("/health", handler);
```

10.14.2 Confondre paramètres de chemin et paramètres de requête

```
// Chemin :  /users/{id}    →  req.param("id")
// Requête : /users?page=2  →  req.query_value("page", "1")
```

10.14.3 Oublier de retourner après une erreur

```
// Retournez toujours immédiatement après avoir envoyé une erreur
if (bad) {
  res.status(400).json(...);
  return;
}
```

10.14.4 Route wildcard trop tôt

Les routes spécifiques doivent être enregistrées avant les routes wildcard.

10.15 Ce qu'il faut retenir

Une route relie : méthode HTTP + chemin → handler C++. Utilisez les paramètres de chemin pour l'identité de la ressource, les paramètres de requête pour les options, et des fonctions groupées au fur et à mesure que l'application grandit. L'idée centrale : les routes sont la forme publique de votre application — gardez-les claires, prévisibles et bien organisées.

10.16 Chapitre suivant

Suivant : Requête et réponse

Chapitre 11

Requête et réponse

Ils sont au cœur du modèle HTTP de Vix :

```
Request lit ce que le client a envoyé.
Response envoie ce que votre application retourne.
```

11.1 Qu’est-ce que Request ?

Méthode	Rôle
`req.param("id")`	Lit un paramètre de chemin de la route.
`req.param("id", "0")`	Lit un paramètre de route avec valeur par défaut.
`req.query_value("page", "1")`	Lit un paramètre de requête avec valeur par défaut.
`req.query()`	Lit tous les paramètres de requête.
`req.header("Authorization")`	Lit la valeur d’un header de requête.
`req.body()`	Lit le corps brut de la requête.
`req.json()`	Lit le corps JSON parsé.
`req.path()`	Lit le chemin courant de la requête.

11.2 Qu’est-ce que Response ?

Méthode	Rôle
`res.text("Hello")`	Envoie une réponse texte brut.
`res.json({"ok", true})`	Envoie une réponse JSON.
`res.status(201).json(...)`	Fixe le statut avant l’envoi.
`res.header("X-Foo", "bar")`	Fixe la valeur d’un header.

Méthode	Rôle
`res.file("public/index.html")`	Envoie une réponse fichier statique.

11.3 Lire les paramètres de chemin

```
app.get("/users/{id}", [](Request &req, Response &res){
  const std::string id = req.param("id");
  res.json({"id", id});
});
```

11.4 Lire les paramètres de requête

```
app.get("/search", [](Request &req, Response &res){
  const std::string q = req.query_value("q", "");
  const std::string page = req.query_value("page", "1");

  res.json({
    "q", q,
    "page", page
  });
});
```

```
curl -i "http://127.0.0.1:8080/search?q=vix&page=2"
```

11.5 Lire tous les paramètres de requête

```
app.get("/debug/query", [](Request &req, Response &res){
  res.json({"query", req.query()});
});
```

11.6 Lire les headers

```
app.get("/debug/headers", [](Request &req, Response &res){
  res.json({
    "user_agent", req.header("User-Agent"),
    "authorization", req.header("Authorization")
```

```
  });
});
```

11.7 Lire le corps brut

```
app.post("/debug/body", [](Request &req, Response &res){
  res.json({"body", req.body()});
});
```

11.8 Lire un corps JSON

```
app.post("/users", [](Request &req, Response &res){
  const auto &body = req.json();

  if (!body.is_object()){
    res.status(400).json({
      "ok", false,
      "error", "expected JSON object"
    });

    return;
  }

  const std::string name = body.value("name", "");
  const std::string role = body.value("role", "user");

  if (name.empty()){
    res.status(400).json({
      "ok", false,
      "error", "name is required"
    });

    return;
  }

  res.status(201).json({
    "ok", true,
    "user", vix::json::o("name", name),
    "role", role
```

```
  });

});
```

11.9 Définir des headers

```
app.get("/health", [](Request &, Response &res){
  res.header("X-Powered-By", "Vix.cpp");
  res.json({"ok", true});
});
```

11.10 Header Cache-Control

```
res.header("Cache-Control", "public, max-age=3600");
res.json({"ok", true});
```

11.11 Réponse de téléchargement

```
res.header("Content-Disposition", "attachment; filename=\"hello.txt\"");
res.file("public/hello.txt");
```

11.12 Helper d'erreur

```
static void respond_error(Response &res, int status, const std::string &message){
  res.status(status).json({
    "ok", false,
    "error", message
  });
}
```

11.13 Forme de réponse recommandée

```
// Succès
{ "ok": true, "data": {} }

// Erreur
{ "ok": false, "error": "message" }
```

```
// Liste
{ "ok": true, "count": 2, "data": [] }
```

11.14 Cycle de vie de Request et Response

```
le client envoie une requête
  ↓
Vix crée Request
  ↓
le handler de route lit Request
  ↓
le handler de route écrit Response
  ↓
Vix envoie la réponse au client
```

11.15 Erreurs courantes

11.15.1 Oublier de nommer Request quand vous en avez besoin

```
// Incorrect — req est anonyme mais utilisé
app.get("/users/{id}", [](Request &, Response &res){
  const std::string id = req.param("id");
});   // erreur !

// Correct
app.get("/users/{id}", [](Request &req, Response &res){
  const std::string id = req.param("id");
  res.json({"id", id});
});
```

11.15.2 Oublier de retourner après une erreur

```
// Incorrect
if (name.empty()) {
  respond_error(res, 400, "name is required");
}

res.status(201).json({"ok", true});
```

```
// Correct
if (name.empty()) {
  respond_error(res, 400, "name is required");
  return;
}

res.status(201).json({"ok", true});
```

11.15.3 Faire confiance au corps JSON sans vérifier sa forme

```
// Meilleur
if (!body.is_object()) {
  respond_error(res, 400, "expected JSON object");
  return;
}
```

11.16 Ce qu'il faut retenir

Request est l'entrée : params, query, headers, body, json. **Response** est la sortie : status, headers, text, json, file.

Le flux de base d'une route : lire Request → valider l'entrée → écrire Response → retourner.

11.17 Chapitre suivant

Suivant : API JSON

Chapitre 12

API JSON

Vous allez maintenant construire une API JSON complète. Les API JSON sont l'une des choses les plus courantes que vous construirez avec Vix.

```
le client envoie du JSON → Vix lit Request → la route valide l'entrée → la route retourne une R
```

12.1 Routes à construire

```
GET  /
GET  /health
GET  /api/users
GET  /api/users/{id}
POST /api/users
```

12.2 Forme de réponse recommandée

```
{ "ok": true, "data": {} }
{ "ok": true, "count": 2, "data": [] }
{ "ok": false, "error": "message" }
```

12.3 Struct User

```
struct User
{
  std::int64_t id{};
  std::string name;
  std::string role;
```

```
};

static std::vector<User> make_seed_users()
{
  return { {1, "Alice", "admin"}, {2, "Bob", "user"} };
}
```

Les données sont en mémoire pour l'instant. Le chapitre sur la base de données remplacera ceci par SQLite ou MySQL.

12.4 Helpers JSON

```
static json::Json user_to_json(const User &user)
{
  return json::kv({
      {"id", json::Json(user.id)},
      {"name", json::Json(user.name)},
      {"role", json::Json(user.role)},
  });
}

static json::Json users_to_json(const std::vector<User> &users)
{
  json::Json items = json::Json::array();
  for (const auto &user : users)
    items.push_back(user_to_json(user));

  return items;
}

static void respond_error(Response &res, int status, const std::string &message)
{
  res.status(status).json(json::kv({
      {"ok", json::Json(false)},
      {"error", json::Json(message)},
  }));
}

static std::optional<User> find_user_by_id(const std::vector<User> &users, std::int64_t id)
{
```

```
  for (const auto &user : users)
    if (user.id == id)
      return user;

  return std::nullopt;
}

static std::optional<std::int64_t> parse_id(const std::string &text)
{
  try {
    return std::stoll(text);
  }
  catch (...) {
    return std::nullopt;
  }
}
```

12.5 GET /api/users

```
app.get("/api/users", [&users](Request &, Response &res){
  const auto data = users_to_json(users);
  res.json(json::kv({
      {"ok", json::Json(true)},
      {"count", json::Json(static_cast<int>(users.size()))},
      {"data", data}
  }));
});
```

```
curl -i http://127.0.0.1:8080/api/users
```

12.6 GET /api/users/{id}

```
app.get("/api/users/{id}", [&users](Request &req, Response &res){

  const auto id = parse_id(req.param("id"));
  if (!id) {
    respond_error(res, 400, "invalid user id");
    return;
  }
```

```
  const auto user = find_user_by_id(users, *id);
  if (!user) {
    respond_error(res, 404, "user not found");
    return;
  }

  res.json(json::kv({
      {"ok", json::Json(true)},
      {"data", user_to_json(*user)}
  }));

});
```

```
curl -i http://127.0.0.1:8080/api/users/1
curl -i http://127.0.0.1:8080/api/users/999   # 404
curl -i http://127.0.0.1:8080/api/users/abc   # 400
```

12.7 POST /api/users

```
app.post("/api/users", [&users](Request &req, Response &res){

  const auto &body = req.json();
  if (!body.is_object())
  {
    respond_error(res, 400, "expected JSON object body");
    return;
  }

  const std::string name = body.value("name", "");
  const std::string role = body.value("role", "user");

  if (name.empty()) {
    respond_error(res, 400, "field 'name' is required");
    return;
  }

  const std::int64_t next_id = users.empty() ? 1 : users.back().id + 1;
  User user{next_id, name, role.empty() ? "user" : role};
  users.push_back(user);
```

```
  res.status(201).json(json::kv({
      {"ok", json::Json(true)},
      {"message", json::Json("user created")},
      {"data", user_to_json(user)}
  }));

});
```

```
curl -i -X POST http://127.0.0.1:8080/api/users \
  -H "Content-Type: application/json" \
  -d '{"name":"Charlie","role":"user"}'
```

12.8 Exemple complet

```
#include <vix.hpp>
#include <cstdint>
#include <optional>
#include <string>
#include <vector>

using namespace vix;

struct User {
  std::int64_t id{};
  std::string name;
  std::string role;
};

static std::vector<User> make_seed_users(){
  return {
    {1, "Alice", "admin"},
    {2, "Bob", "user"}
  };
}

static json::Json user_to_json(const User &u){
  return json::kv({
    {"id", json::Json(u.id)},
    {"name", json::Json(u.name)},
    {"role", json::Json(u.role)}
```

```
  });
}

static json::Json users_to_json(const std::vector<User> &users)
{
  json::Json items = json::Json::array();
  for (const auto &u : users)
    items.push_back(user_to_json(u));

  return items;
}

static void respond_error(Response &res, int status, const std::string &msg){
  res.status(status).json(json::kv({
    {"ok", json::Json(false)},
    {"error", json::Json(msg)}
  }));
}

static std::optional<User> find_user_by_id(const std::vector<User> &users, std::int64_t id)
{
  for (const auto &u : users)
    if (u.id == id)
      return u;

  return std::nullopt;
}

static std::optional<std::int64_t> parse_id(const std::string &text)
{
  try {
    return std::stoll(text);
  } catch (...) {
    return std::nullopt;
  }
}

static void public_routes(App &app)
{
  app.get("/", [](Request &, Response &res){
```

```
    res.json(json::kv({
      {"message", json::Json("Vix JSON API")}
    }));
  });

  app.get("/health", [](Request &, Response &res){
    res.json(json::kv({
      {"ok", json::Json(true)},
      {"service", json::Json("json-api")}
    }));
  });

}

static void user_routes(App &app, std::vector<User> &users)
{
  app.get("/api/users", [&users](Request &, Response &res){
    res.json(json::kv({
        {"ok", json::Json(true)},
        {"count", json::Json(static_cast<int>(users.size()))},
        {"data", users_to_json(users)}
    }));
  });

  app.get("/api/users/{id}", [&users](Request &req, Response &res){
    const auto id = parse_id(req.param("id"));

    if (!id) {
      respond_error(res, 400, "invalid user id");
      return;
    }

    const auto user = find_user_by_id(users, *id);
    if (!user) {
      respond_error(res, 404, "user not found");
      return;
    }

    res.json(json::kv({
      {"ok", json::Json(true)},
```

```
      {"data", user_to_json(*user)}
    }));
  });

  app.post("/api/users", [&users](Request &req, Response &res){
    const auto &body = req.json();

    if (!body.is_object()) {
      respond_error(res, 400, "expected JSON object body");
      return;
    }

    const std::string name = body.value("name", "");
    const std::string role = body.value("role", "user");
    if (name.empty()) {
      respond_error(res, 400, "field 'name' is required");
      return;
    }

    const std::int64_t next_id = users.empty() ? 1 : users.back().id + 1;
    User user{next_id, name, role.empty() ? "user" : role};
    users.push_back(user);

    res.status(201).json(json::kv({
        {"ok", json::Json(true)},
        {"message", json::Json("user created")},
        {"data", user_to_json(user)}
    }));

  });
}

int main()
{
  std::vector<User> users = make_seed_users();

  App app;

  public_routes(app);
  ruser_routes(app, users);
```

```
  app.run(8080);

  return 0;
}
```

Note éditoriale. Dans l'exemple complet ci-dessus, on lit `ruser_routes(...)` au lieu de `user_routes(...)`. Cette coquille provient de la source originale et a été conservée à l'identique ; la fonction réelle s'appelle bien `user_routes`.

12.9 Tester l'API complète

```
curl -i http://127.0.0.1:8080/health
curl -i http://127.0.0.1:8080/api/users
curl -i http://127.0.0.1:8080/api/users/1
curl -i http://127.0.0.1:8080/api/users/999
curl -i http://127.0.0.1:8080/api/users/abc
curl -i -X POST http://127.0.0.1:8080/api/users \
  -H "Content-Type: application/json" \
  -d '{"name":"Charlie","role":"user"}'
curl -i -X POST http://127.0.0.1:8080/api/users \
  -H "Content-Type: application/json" \
  -d '{}'
```

12.10 Codes de statut pour les API JSON

Statut	Signification
200	OK, requête réussie.
201	Created, ressource ajoutée.
400	Bad Request, entrée invalide.
401	Unauthorized, authentification requise.
403	Forbidden, accès refusé.
404	Not Found, ressource introuvable.
409	Conflict, conflit d'état.
429	Too Many Requests, rate limit atteint.
500	Internal Server Error.

12.11 Flux d'une route d'API JSON

```
lire la requête → parser params ou body → valider l'entrée → exécuter la logique → formater le
```

12.12 Préparer les chapitres suivants

- **Base de données :** le vecteur en mémoire sera remplacé par SQLite ou MySQL.
- **Middleware :** CORS, rate limiting et authentification envelopperont les routes.
- **Validation :** les vérifications manuelles deviendront déclaratives avec `vix::validation`.

12.13 Erreurs courantes

12.13.1 Oublier le `Content-Type` avec curl

```
curl -i -X POST http://127.0.0.1:8080/api/users \
  -H "Content-Type: application/json" \
  -d '{"name":"Ada"}'
```

12.13.2 Faire confiance à la forme du body

```
if (!body.is_object()) {
  respond_error(res, 400, "expected JSON object body");
  return;
}
```

12.13.3 Oublier de retourner après une erreur

```
if (name.empty()) {
  respond_error(res, 400, "field 'name' is required");
  return;
}
```

12.13.4 Retourner des erreurs incohérentes

Utilisez un seul helper : `respond_error(res, 400, "message")`.

12.14 Ce qu'il faut retenir

Une route d'API JSON suit : Request → validation → logique → Response JSON. Utilisez `res.json(...)` pour les réponses, `req.json()` pour les corps JSON, et des helpers pour des erreurs et un formatage JSON cohérents. L'idée centrale : les API JSON deviennent simples quand le parsing de la requête, la validation, la logique et le formatage de la réponse restent séparés.

12.15 Chapitre suivant

Suivant : Middleware

Chapitre 13

Middleware

Dans le chapitre précédent, vous avez construit une API JSON. Vous allez maintenant apprendre le middleware.

Un middleware est du code qui s'exécute autour de vos routes. Il peut inspecter la requête avant que le handler de route ne s'exécute, et modifier la réponse avant qu'elle ne soit envoyée.

```
requête → middleware → handler de route → réponse
```

13.1 Pourquoi le middleware existe

Sans middleware, la logique partagée comme CORS, le rate limiting, l'authentification et les headers de sécurité doit être répétée dans chaque route. Le middleware vous permet d'écrire un comportement commun une seule fois.

13.2 Headers publics

```
#include <vix.hpp>
#include <vix/middleware.hpp>
```

13.3 Ordre du middleware

L'ordre compte. Configurez le middleware avant d'enregistrer les routes :

```
int main()
{
  App app;

  configure_middlewares(app);
```

```
  register_routes(app);

  app.run(8080);
  return 0;
}
```

Un ordre courant :

```
CORS → rate limit → headers de sécurité → limite de body → authentification → routes
```

13.4 Middleware CORS

```
app.use(vix::middleware::app::cors_dev({
    "http://localhost:5173",
    "http://127.0.0.1:5173"
}));
```

CORS n'est pas de l'authentification. C'est une règle du navigateur qui répond à : quelles origines de navigateur sont autorisées à appeler cette API ?

Pour la production, n'autorisez que le vrai domaine de votre frontend. N'utilisez pas un CORS ouvert sauf si l'API est volontairement publique.

13.5 Middleware de rate limiting

```
app.use(vix::middleware::app::rate_limit({
    .max_requests = 60,
    .window_seconds = 60
}));
```

Utilisez le rate limiting pour protéger les API publiques, les endpoints de login et les déploiements sur petits VPS. Quand la limite est dépassée, l'API retourne `429 Too Many Requests`.

13.5.1 Tester le rate limiting

```
app.use(vix::middleware::app::rate_limit({
    .max_requests = 5,
    .window_seconds = 30
}));
```

```
for i in $(seq 1 10); do
  curl -s -o /dev/null -w "%{http_code}\n" http://127.0.0.1:8080/api/data
done
```

13.6 Middleware plus strict pour les routes d'authentification

```
vix::middleware::app::use_on_prefix(
    app,
    "/auth",
    vix::middleware::app::rate_limit({
        .max_requests = 5,
        .window_seconds = 60
    }));
```

13.7 Middleware de fichiers statiques

```
#include <vix/middleware/app/adapter.hpp>
#include <vix/middleware/performance/static_files.hpp>

app.use(vix::middleware::app::adapt_ctx(
  vix::middleware::performance::static_files(
      std::filesystem::path{"public"},
      {
          .mount = "/",
          .index_file = "index.html",
          .add_cache_control = true,
          .cache_control = "public, max-age=3600",
          .fallthrough = true,
      }
  )
));
```

13.8 Vérification d'authentification manuelle dans une route

```
app.get("/api/private", [](Request &req, Response &res){

  const std::string auth = req.header("Authorization");
```

```
  if (auth != "Bearer dev-token")
  {
    res.status(401).json({
      "ok", false,
      "error", "unauthorized"
    });

    return;
  }

  res.json({
    "ok", true,
    "message", "private data"
  });

});
```

13.9 Exemple complet

```
#include <vix.hpp>
#include <vix/middleware.hpp>
using namespace vix;

static void respond_error(Response &res, int status, const std::string &message)
{
  res.status(status).json({"ok", false, "error", message});
}

static void configure_middlewares(App &app)
{
  app.use(vix::middleware::app::cors_dev({
      "http://localhost:5173",
      "http://127.0.0.1:5173"
  }));

  app.use(vix::middleware::app::rate_limit({
      .max_requests = 60,
      .window_seconds = 60
  }));
```

```
  vix::middleware::app::use_on_prefix(
      app,
      "/auth",
      vix::middleware::app::rate_limit({
          .max_requests = 5,
          .window_seconds = 60
      })
  );
}

static void public_routes(App &app)
{
  app.get("/", [](Request &, Response &res){
    res.json({"message", "Vix middleware example"});
  });

  app.get("/health", [](Request &, Response &res){
    res.json({
      "ok", true,
      "service", "middleware-example"
    });
  });
}

static void api_routes(App &app)
{
  app.get("/api/data", [](Request &, Response &res){
    res.json({
      "ok", true,
      "data", vix::json::o("name", "Vix.cpp"),
      "type", "runtime"
    });
  });

  app.get("/api/private", [](Request &req, Response &res){
    if (req.header("Authorization") != "Bearer dev-token"){
      respond_error(res, 401, "unauthorized");
      return;
    }
    res.json({"ok", true, "message", "private data"});
```

```
  });
}

static void auth_routes(App &app)
{
  app.post("/auth/login", [](Request &req, Response &res)
           {
             const auto &body = req.json();
             if (!body.is_object()) { respond_error(res, 400, "expected JSON object body"); ret
             const std::string email = body.value("email", "");
             const std::string password = body.value("password", "");
             if (email != "ada@example.com" || password != "password123")
             {
               respond_error(res, 401, "invalid credentials");
               return;
             }
             res.json({"ok", true, "token", "dev-token"});
           });
}

int main()
{
  App app;

  configure_middlewares(app);
  public_routes(app);
  api_routes(app);
  auth_routes(app);

  app.run(8080);
  return 0;
}
```

13.10 Test

```
curl -i http://127.0.0.1:8080/health
curl -i http://127.0.0.1:8080/api/data
curl -i http://127.0.0.1:8080/api/private
curl -i http://127.0.0.1:8080/api/private -H "Authorization: Bearer dev-token"
curl -i -X POST http://127.0.0.1:8080/auth/login \
```

```
  -H "Content-Type: application/json" \
  -d '{"email":"ada@example.com","password":"password123"}'

# Tester CORS
curl -i http://127.0.0.1:8080/api/data -H "Origin: http://localhost:5173"
```

13.11 Middleware et responsabilité des routes

Middleware	Exemple d'action sur la route
CORS	Autorise les appels d'API du navigateur.
Rate limiting	Protège les endpoints publics.
Authentification	Protège les routes du dashboard.
Request IDs	Trace chaque requête dans les logs.
Logging	Enregistre l'activité des requêtes.
Limites de body	Rejette les requêtes trop volumineuses.

13.12 Erreurs courantes

13.12.1 Enregistrer le middleware après les routes

```
// Incorrect
register_routes(app);
configure_middlewares(app);

// Correct
configure_middlewares(app);
register_routes(app);
```

13.12.2 Rendre CORS trop ouvert en production

Le CORS de développement peut autoriser localhost. La production ne devrait autoriser que votre véritable frontend.

13.12.3 Utiliser un seul rate limit pour tout

Le login nécessite une limite plus stricte que les routes d'API normales.

13.12.4 Retourner 403 au lieu de 429 pour les rate limits

Les échecs de rate limit doivent retourner `429 Too Many Requests`.

13.13 Ce qu'il faut retenir

Le middleware enveloppe les handlers de routes. Utilisez-le pour un comportement partagé : CORS, rate limiting, authentification, journalisation, sécurité, limites de body, fichiers statiques.

```
int main()
{
  App app;

  configure_middlewares(app);
  register_routes(app);

  app.run(8080);
}
```

L'idée centrale : le middleware garde les handlers de route propres en déplaçant le comportement partagé des requêtes dans une couche réutilisable unique.

13.14 Chapitre suivant

Suivant : Validation

Chapitre 14

Validation

Dans le chapitre précédent, vous avez appris le middleware. Vous allez maintenant apprendre la validation.

La validation vérifie si les données entrantes sont correctes avant que votre application ne les utilise.

```
données de la requête → validation → logique métier → réponse
```

14.1 Pourquoi la validation existe

Une route qui fait aveuglément confiance à l'entrée peut recevoir des champs manquants, des emails invalides, des mots de passe faibles, des types incorrects ou des données dangereuses. La validation empêche les mauvaises entrées d'atteindre la vraie logique applicative.

14.2 Header public

```
#include <vix/validation.hpp>
```

14.3 Valider une chaîne

```
auto result = vix::validation::validate("email", email)
                  .required()
                  .email()
                  .length_max(120)
                  .result();

if (!result.ok())
```

```
{
  for (const auto &error : result.errors.all())
  {
    std::cout << "field=" << error.field << " message=" << error.message << "\n";
  }
}
```

14.4 Règles courantes

Règle	Rôle
`required()`	Exige une valeur présente et non vide.
`email()`	Exige un format d'email valide.
`length_min(n)`	Exige une longueur de chaîne d'au moins `n`.
`length_max(n)`	Exige une longueur de chaîne d'au plus `n`.
`min(n)`	Exige une valeur numérique d'au moins `n`.
`max(n)`	Exige une valeur numérique d'au plus `n`.
`between(a, b)`	Exige une valeur entre `a` et `b`.
`in_set({...})`	Exige une des valeurs autorisées.

14.5 Valider des nombres

```
int age = 17;
auto result = vix::validation::validate("age", age)
                  .min(18, "must be adult")
                  .max(120)
                  .result();
```

14.6 Valider des valeurs autorisées

```
auto result = vix::validation::validate("role", role)
                  .required()
                  .in_set({"admin", "user", "guest"})
                  .result();
```

14.7 Validation avec parsing (string → nombre)

```
auto result = vix::validation::validate_parsed<int>("age", input)
                  .between(18, 120)
                  .result("age must be a number");
```

Utile pour les paramètres de requête, les paramètres de route et les champs de formulaire qui arrivent sous forme de chaînes.

14.8 Validation par schéma

```
struct UserInput
{
  std::string email;
  std::string password;

  static vix::validation::Schema<UserInput> schema()
  {
    return vix::validation::schema<UserInput>()
        .field("email", &UserInput::email,
               vix::validation::field<std::string>().required().email().length_max(120))

        .field("password", &UserInput::password,
               vix::validation::field<std::string>().required().length_min(8).length_max(64));
  }
};

UserInput input;
input.email = "bad-email";
input.password = "123";

auto result = UserInput::schema().validate(input);
```

14.9 BaseModel

```
struct RegisterForm : vix::validation::BaseModel<RegisterForm>
{
  std::string email;
  std::string password;
```

```
  static vix::validation::Schema<RegisterForm> schema()
  {
    return vix::validation::schema<RegisterForm>()
        .field("email", &RegisterForm::email,
               vix::validation::field<std::string>().required().email().length_max(120))

        .field("password", &RegisterForm::password,
               vix::validation::field<std::string>().required().length_min(8).length_max(64));
  }
};

RegisterForm form;
auto result = form.validate();            // appel sur l'objet
auto result2 = RegisterForm::validate(form);  // appel statique
```

14.10 Validation cross-champ

```
.check([](const ResetPassword &obj, vix::validation::ValidationErrors &errors){
  if (!obj.password.empty() && !obj.confirm.empty() && obj.password != obj.confirm){
      errors.add("confirm", vix::validation::ValidationErrorCode::Custom,
                      "passwords do not match");
    }
});
```

14.11 Validation dans une route

```
static json::Json validation_errors_to_json(
    const vix::validation::ValidationErrors &errors)
{
  json::Json items = json::Json::array();
  for (const auto &error : errors.all())
  {
    items.push_back(json::kv({
        {"field", json::Json(error.field)},
        {"code", json::Json(vix::validation::to_string(error.code))},
        {"message", json::Json(error.message)},
    }));
  }
  return items;
```

```
}

template <typename Result>
static void respond_validation_error(Response &res, const Result &result)
{
  res.status(400).json(json::kv({
      {"ok", json::Json(false)},
      {"error", json::Json("validation failed")},
      {"errors", validation_errors_to_json(result.errors)}
  }));
}

app.post("/api/register", [](Request &req, Response &res){

  const auto &body = req.json();
  if (!body.is_object()) {
    res.status(400).json({
      "ok", false,
      "error", "expected JSON object body"
    });
    return;
  }

  RegisterInput input;
  input.email = body.value("email", "");
  input.password = body.value("password", "");

  auto result = input.validate();
  if (!result.ok()) {
    respond_validation_error(res, result);
    return;
  }

  res.status(201).json({
    "ok", true,
    "message", "registered"
  });

});
```

14.12 Forme d'erreur structurée

```
{
  "ok": false,
  "error": "validation failed",
  "errors": [
    { "field": "email", "code": "format", "message": "invalid email format" },
    { "field": "password", "code": "length_min", "message": "password too short" }
  ]
}
```

14.13 Erreurs courantes

14.13.1 Valider après la logique métier

```
// Incorrect : créer l'utilisateur puis valider
// Correct  : valider puis créer l'utilisateur
```

14.13.2 Oublier la vérification de la forme du body

```
if (!body.is_object()) {
  respond_error(res, 400, "expected JSON object body");
  return;
}
```

14.13.3 Oublier de retourner après l'échec de la validation

```
if (!result.ok()) {
  respond_validation_error(res, result);
  return;
}
```

14.13.4 Retourner une seule erreur de validation

Pour les formulaires, retournez toutes les erreurs de champs en une seule fois afin que les utilisateurs puissent tout corriger ensemble.

14.14 Ce qu'il faut retenir

Le flux normal : Request → validation → logique métier → Response. Utilisez la validation à valeur unique pour les champs simples, les schémas pour les structs et des erreurs structurées pour les APIs. L'idée centrale : les mauvaises entrées doivent s'arrêter à la frontière de votre application.

14.15 Chapitre suivant

Suivant : Erreurs et journalisation

Chapitre 15

Erreurs et journalisation

Dans le chapitre précédent, vous avez appris la validation. Vous allez maintenant apprendre la gestion des erreurs et la journalisation.

```
requête → validation → logique métier → erreur ou succès → réponse structurée → logs structurés
```

Une application de production doit retourner des réponses claires aux clients et conserver des logs utiles pour les développeurs.

15.1 Pourquoi la gestion des erreurs compte

Si chaque route retourne une forme d'erreur différente, l'API devient difficile à utiliser. Une API Vix doit utiliser une seule forme prévisible.

15.2 Forme d'erreur recommandée

```
{ "ok": false, "error": "message" }
{ "ok": false, "error": "validation_failed", "errors": [] }
{ "ok": true, "data": {} }
```

15.3 Codes de statut HTTP

Statut	Signification
200	OK, requête réussie.
201	Created, ressource ajoutée.
400	Bad Request, entrée invalide.
401	Unauthorized, authentification requise.
403	Forbidden, accès refusé.

Statut	Signification
404	Not Found, ressource introuvable.
409	Conflict, conflit d'état.
429	Too Many Requests, rate limit atteint.
500	Internal Server Error.

Ne retournez pas 200 pour des erreurs.

15.4 Helper d'erreur basique

```
static void respond_error(
    vix::Response &res,
    int status,
    const std::string &code,
    const std::string &message)
{
  res.status(status).json(vix::json::kv({
      {"ok", vix::json::Json(false)},
      {"error", vix::json::Json(code)},
      {"message", vix::json::Json(message)},
  }));
}
```

Retournez toujours immédiatement après avoir envoyé une erreur :

```
if (name.empty()) {
  respond_error(res, 400, "validation_failed", "name is required");
  return;
}
```

15.5 Codes d'erreur

Utilisez des codes stables pour les APIs en production :

```
invalid_request,
validation_failed,
unauthorized,
forbidden,
not_found,
conflict,
rate_limited,
```

```
internal_error,
user_not_found,
email_already_used,
invalid_credentials,
product_not_found,
invalid_token,
session_expired
```

15.6 Ne pas exposer les erreurs internes

```
catch (const std::exception &e)
{
  vix::log::error("unhandled route error", "details", e.what());
  res.status(500).json({
    "ok", false,
    "error", "internal_error",
    "message", "Internal server error"
  });
}
```

15.7 Header de journalisation public

```
#include <vix/log.hpp>
```

15.8 Logs basiques

```
vix::log::set_level(vix::log::LogLevel::Trace);
vix::log::trace("trace message");
vix::log::debug("debug message");
vix::log::info("info message");
vix::log::warn("warn message");
vix::log::error("error message");
vix::log::critical("critical message");
```

15.9 Niveaux de log

Niveau	Utilisation
`trace`	Enregistre des événements de débogage très fins.
`debug`	Enregistre des informations de débogage utiles.
`info`	Enregistre les événements normaux de l'application.
`warn`	Enregistre les événements inhabituels mais non fatals.
`error`	Enregistre les opérations échouées.
`critical`	Enregistre les défaillances système graves.

Recommandé : `info` en production, `debug` ou `trace` pendant le développement.

15.10 Logs structurés

```
vix::log::logf(
    vix::log::LogLevel::Info,
    "user authenticated successfully",
    "status", 200,
    "method", "POST",
    "path", "/login"
);
```

15.11 Formats de log

```
vix::log::set_format(vix::log::LogFormat::KV);    // développement local
vix::log::set_format(vix::log::LogFormat::JSON);   // collecteurs de logs en production
```

15.12 Contexte de log

```
vix::log::LogContext ctx;
ctx.request_id = "req-abc-123";
ctx.module = "auth";
ctx.fields["user_id"] = "42";
vix::log::set_context(ctx);
vix::log::info("user authenticated successfully");
vix::log::clear_context();
```

15.13 Fixer le niveau de log

```
vix::log::set_level(vix::log::LogLevel::Info);  // dans le code
// vix run main.cpp --log-level debug             // via la CLI
// VIX_LOG_LEVEL=info                             // via l'environnement
```

15.14 Que journaliser

Bien : application démarrée, utilisateur enregistré, échec de login, échec de connexion à la base de données, exception inattendue.

À ne jamais journaliser : mots de passe, tokens, clés privées, données personnelles sensibles.

15.15 Exemple complet

```
#include <vix.hpp>
#include <vix/log.hpp>
#include <vix/validation.hpp>

using namespace vix;

struct RegisterInput : vix::validation::BaseModel<RegisterInput>
{
  std::string email;
  std::string password;

  static vix::validation::Schema<RegisterInput> schema()
  {
    return vix::validation::schema<RegisterInput>()
        .field("email", &RegisterInput::email,
               vix::validation::field<std::string>().required().email().length_max(120))

        .field("password", &RegisterInput::password,
               vix::validation::field<std::string>().required().length_min(8).length_max(64));
  }
};

static json::Json validation_errors_to_json(const vix::validation::ValidationErrors &errors)
{
```

```
  json::Json items = json::Json::array();

  for (const auto &error : errors.all())
    items.push_back(json::kv({
      {"field", json::Json(error.field)},
      {"code", json::Json(vix::validation::to_string(error.code))},
      {"message", json::Json(error.message)}
    }));

  return items;
}

static void respond_error(Response &res, int status, const std::string &code, const std::string
{
  res.status(status).json(json::kv({
    {"ok", json::Json(false)},
    {"error", json::Json(code)},
    {"message", json::Json(message)}
  }));
}

int main()
{
  vix::log::set_level(vix::log::LogLevel::Info);
  vix::log::set_format(vix::log::LogFormat::KV);
  vix::log::info("starting errors and logging example");

  App app;

  app.get("/health", [](Request &, Response &res){
    vix::log::debug("health check requested");
    res.json({
      "ok", true,
      "service", "errors-logging-example"
    });
  });

  app.post("/api/register", [](Request &req, Response &res){

    try{
```

```
    const auto &body = req.json();
    if (!body.is_object()) {
      respond_error(res, 400, "invalid_request", "Expected JSON object body");
      return;
    }

    RegisterInput input;
    input.email = body.value("email", "");
    input.password = body.value("password", "");

    auto result = input.validate();
    if (!result.ok()){
      vix::log::warn("register validation failed", "email", input.email);

      res.status(400).json(json::kv({
        {"ok", json::Json(false)},
        {"error", json::Json("validation_failed")},
        {"errors", validation_errors_to_json(result.errors)}
      }));

      return;
    }

    vix::log::logf(vix::log::LogLevel::Info, "user registered", "email", input.email);

    res.status(201).json({
      "ok", true,
      "message", "registered"
    });

  }
  catch (const std::exception &e){
    vix::log::error("register failed with exception", "details", e.what());
    respond_error(res, 500, "internal_error", "Internal server error");
  }

});

app.run(8080);
```

```
  return 0;
}
```

15.16 Test

```
curl -i http://127.0.0.1:8080/health
curl -i -X POST http://127.0.0.1:8080/api/register \
  -H "Content-Type: application/json" \
  -d '{"email":"ada@example.com","password":"password123"}'
curl -i -X POST http://127.0.0.1:8080/api/register \
  -H "Content-Type: application/json" \
  -d '{"email":"bad-email","password":"123"}'
```

15.17 Erreurs courantes

15.17.1 Retourner HTTP 200 pour les erreurs

Utilisez toujours le bon code de statut d'erreur.

15.17.2 Journaliser des secrets

Ne journalisez jamais de mots de passe, tokens, clés privées ou headers d'autorisation.

15.17.3 Exposer les exceptions internes

```
// Incorrect
res.json({"error", e.what()});

// Correct
vix::log::error("failed", "details", e.what());

res.json({
  "ok", false,
  "error", "internal_error"
});
```

15.17.4 Oublier de retourner après une erreur

```
if (!result.ok()) {
  respond_validation_error(res, result);
  return;
}
```

15.18 Configuration de production

```
VIX_LOG_LEVEL=info
VIX_LOG_FORMAT=json
VIX_COLOR=never
```

15.19 Ce qu'il faut retenir

Les erreurs sont pour les clients. Les logs sont pour les développeurs et les opérateurs. Les erreurs d'API doivent être cohérentes : `{ "ok": false, "error": "stable_code", "message": "Safe message" }`. Les logs doivent conserver un contexte interne utile avec des champs structurés. L'idée centrale : une application fiable ne fonctionne pas seulement quand tout réussit — elle explique aussi ce qui s'est passé quand quelque chose échoue.

15.20 Chapitre suivant

Suivant : Base de données

Chapitre 16

Base de données

Dans le chapitre précédent, vous avez appris les erreurs et la journalisation. Vous allez maintenant connecter une application Vix à une base de données.

```
Request → validation → requête base de données → Response JSON
```

La mémoire disparaît au redémarrage de l'application. Une base de données donne à votre application un état durable.

16.1 Header public

```
#include <vix/db.hpp>
```

16.2 SQLite ou MySQL ?

Critère	SQLite	MySQL
Idéal pour	Développement local, petites apps, MVPs.	APIs en production multi-utilisateurs.
Configuration	Très simple, pas de serveur requis.	Nécessite un serveur de base de données.
Persistance	Stocke les données dans un fichier local.	Stocke les données sur le serveur.

Commencez par SQLite pour apprendre.

16.3 Flags de build

```
vix build --with-sqlite
vix run main.cpp --with-sqlite

vix build --with-mysql
vix run main.cpp --with-mysql
```

16.4 Première connexion SQLite

```
#include <vix/db.hpp>

auto db = vix::db::Database::sqlite("vix.db");
db.exec("CREATE TABLE IF NOT EXISTS healthcheck (id INTEGER PRIMARY KEY)");
```

16.5 Première connexion MySQL

```
auto db = vix::db::Database::mysql("tcp://127.0.0.1:3306", "root", "", "vixdb");
```

16.6 Base de données depuis le fichier .env

```
DATABASE_ENGINE=sqlite
DATABASE_DEFAULT_NAME=vix.db
```

```
vix::config::Config cfg{".env"};
vix::db::Database db{cfg};
```

16.7 Créer une table

```
// SQLite
db.exec(
    "CREATE TABLE IF NOT EXISTS users ("
    "id INTEGER PRIMARY KEY AUTOINCREMENT, "
    "name TEXT NOT NULL, "
    "role TEXT NOT NULL)");

// MySQL
db.exec(
```

```
    "CREATE TABLE IF NOT EXISTS users ("
    "id BIGINT PRIMARY KEY AUTO_INCREMENT, "
    "name VARCHAR(255) NOT NULL, "
    "role VARCHAR(64) NOT NULL)");
```

16.8 Insérer des données

```
db.exec("INSERT INTO users (name, role) VALUES (?, ?)", "Alice", "admin");
```

Utilisez toujours des requêtes paramétrées. Ne construisez jamais de SQL par concaténation de chaînes.

16.9 Interroger des données

```
auto rows = db.query("SELECT id, name, role FROM users");
while (rows->next())
{
  const auto &row = rows->row();
  std::cout << row.getInt64(0) << " " << row.getString(1) << " " << row.getString(2) << "\n";
}
```

16.10 Requêtes préparées

```
vix::db::PooledConn conn(db.pool());
auto stmt = conn->prepare("SELECT id, name FROM users WHERE id = ?");
stmt->bind(1, static_cast<std::int64_t>(1));
auto rows = stmt->query();
```

Utilisez des requêtes préparées pour : les entrées utilisateur, les paramètres de route, les filtres de requête, les inserts, les updates, les deletes.

16.11 Pool de connexions

```
vix::db::PooledConn conn(db.pool());
// la connexion retourne automatiquement au pool quand PooledConn est détruit (RAII)
```

16.12 Transactions

```
db.transaction([&](vix::db::Connection &conn){
  conn.prepare("INSERT INTO users (name, role) VALUES (?, ?)")
      ->bind(1, "Alice")->bind(2, "admin")->exec();
  conn.prepare("INSERT INTO users (name, role) VALUES (?, ?)")
      ->bind(1, "Bob")->bind(2, "user")->exec();
});
```

Utilisez des transactions pour : les commandes + items, l'utilisateur + le profil, les transferts d'argent, toute écriture multi-étapes.

16.13 API base de données complète

```
#include <vix.hpp>
#include <vix/db.hpp>
#include <vix/log.hpp>
#include <cstdint>
#include <optional>
#include <stdexcept>
#include <string>

using namespace vix;

struct User {
  std::int64_t id{};
  std::string name;
  std::string role;
};

static json::Json user_to_json(const User &u)
{
  return json::kv({
    {"id", json::Json(u.id)},
    {"name", json::Json(u.name)},
    {"role", json::Json(u.role)}
  });
}

static void respond_error(Response &res,
```

```
                          int status,
                          const std::string &code,
                          const std::string &msg)
{
  res.status(status).json(json::kv({
    {"ok", json::Json(false)},
    {"error", json::Json(code)},
    {"message", json::Json(msg)}
  }));
}

static std::optional<std::int64_t> parse_id(const std::string &text)
{
  try {
    return std::stoll(text);
  } catch (...) {
    return std::nullopt;
  }
}

static void initialize_schema(vix::db::Database &db)
{
  db.exec("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NO
}

static void seed_users(vix::db::Database &db)
{
  auto rows = db.query("SELECT COUNT(*) FROM users");

  if (rows->next() && rows->row().getInt64(0) > 0)
    return;

  db.exec("INSERT INTO users (name, role) VALUES (?, ?)", "Alice", "admin");
  db.exec("INSERT INTO users (name, role) VALUES (?, ?)", "Bob", "user");
}

static json::Json list_users(vix::db::Database &db)
{
  json::Json items = json::Json::array();
  auto rows = db.query("SELECT id, name, role FROM users ORDER BY id ASC");
```

```
  while (rows->next())
  {
    const auto &row = rows->row();
    items.push_back(user_to_json({row.getInt64(0), row.getString(1), row.getString(2)}));
  }
  return items;
}

static std::optional<User> find_user_by_id(vix::db::Database &db, std::int64_t id)
{
  vix::db::PooledConn conn(db.pool());
  auto stmt = conn->prepare("SELECT id, name, role FROM users WHERE id = ?");
  stmt->bind(1, id);
  auto rows = stmt->query();

  if (!rows->next())
    return std::nullopt;

  const auto &row = rows->row();
  return User{row.getInt64(0), row.getString(1), row.getString(2)};
}

static User create_user(vix::db::Database &db, const std::string &name, const std::string &role
{
  db.exec("INSERT INTO users (name, role) VALUES (?, ?)", name, role);
  auto rows = db.query("SELECT id, name, role FROM users ORDER BY id DESC LIMIT 1");

  if (!rows->next())
    throw std::runtime_error("failed to load created user");

  const auto &row = rows->row();

  return {
    row.getInt64(0),
    row.getString(1),
    row.getString(2)
  };
}

static void register_user_routes(App &app, vix::db::Database &db)
```

```
{
  app.get("/api/users", [&db](Request &, Response &res){
    try {
      res.json(json::kv({
        {"ok", json::Json(true)},
        {"data", list_users(db)}
      }));

    }catch (const std::exception &e) {
      vix::log::error("failed to list users", "details", e.what()); respond_error(res, 500, "in
    }
  });

  app.get("/api/users/{id}", [&db](Request &req, Response &res){
    const auto id = parse_id(req.param("id"));

    if (!id) {
      respond_error(res, 400, "invalid_id", "Invalid user id");
      return;
    }

    const auto user = find_user_by_id(db, *id);
    if (!user) {
      respond_error(res, 404, "user_not_found", "User not found");
      return;
    }

    res.json(json::kv({
      {"ok", json::Json(true)},
      {"data", user_to_json(*user)}
    }));
  });

  app.post("/api/users", [&db](Request &req, Response &res){
    try{
      const auto &body = req.json();
      if (!body.is_object()) {
        respond_error(res, 400, "invalid_request", "Expected JSON object body");
        return;
      }
```

```
      const std::string name = body.value("name", "");
      const std::string role = body.value("role", "user");
      if (name.empty()) {
        respond_error(res, 400, "validation_failed", "Field 'name' is required");
        return;
      }

      const User user = create_user(db, name, role.empty() ? "user" : role);
      res.status(201).json(json::kv({
        {"ok", json::Json(true)},
        {"message", json::Json("user created")},
        {"data", user_to_json(user)}
      }));

    }catch (const std::exception &e) {
      vix::log::error("failed to create user", "details", e.what()); respond_error(res, 500, "i
    }
  });
}

int main()
{
  vix::log::set_level(vix::log::LogLevel::Info);
  try{
    auto db = vix::db::Database::sqlite("vix.db");
    initialize_schema(db);
    seed_users(db);

    App app;

    app.get("/health", [](Request &, Response &res) {
      res.json({"ok", true, "service", "database-api"});
    });

    register_user_routes(app, db);

    app.run(8080);

    return 0;
  }catch (const std::exception &e){
```

```
    vix::log::critical("application startup failed", "details", e.what());
    return 1;
  }
}
```

16.14 Test

```
curl -i http://127.0.0.1:8080/health
curl -i http://127.0.0.1:8080/api/users
curl -i http://127.0.0.1:8080/api/users/1
curl -i http://127.0.0.1:8080/api/users/999
curl -i -X POST http://127.0.0.1:8080/api/users \
  -H "Content-Type: application/json" \
  -d '{"name":"Charlie","role":"user"}'
```

Redémarrez l'application — contrairement à l'API en mémoire, le nouvel utilisateur devrait toujours exister.

16.15 Migrations

Pour les vrais projets, utilisez des migrations au lieu de `CREATE TABLE IF NOT EXISTS` au démarrage.

```
class CreateUsersTable final : public vix::db::Migration
{
public:
  std::string id() const override { return "2026-01-22-create-users"; }

  void up(vix::db::Connection &conn) override
  {
    conn.prepare("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY AUTOINCREMENT, name
  }

  void down(vix::db::Connection &conn) override
  {
    conn.prepare("DROP TABLE IF EXISTS users")->exec();
  }
};
```

16.16 Erreurs courantes

16.16.1 Construire du SQL par concaténation

```
// Incorrect
std::string sql = "SELECT * FROM users WHERE name = '" + name + "'";

// Correct — utilisez des requêtes préparées
auto stmt = conn->prepare("SELECT * FROM users WHERE name = ?");
stmt->bind(1, name);
```

16.16.2 Retourner des erreurs brutes de base de données

Journalisez les détails internes, retournez aux clients des erreurs sûres.

16.16.3 Ne pas valider avant l'insertion

Validez toujours les entrées avant les écritures en base de données.

16.17 Ce qu'il faut retenir

Le modèle DB de Vix est explicite : se connecter → préparer → binder → interroger → lire les lignes → committer si nécessaire.

Utilisez des requêtes préparées pour les entrées utilisateur. Utilisez des transactions pour les écritures multi-étapes. Les routes doivent valider l'entrée, appeler la logique base de données et retourner des réponses JSON sûres.

16.18 Chapitre suivant

Suivant : WebSocket en temps réel

Chapitre 17

WebSocket en temps réel

Dans le chapitre précédent, vous avez connecté une application Vix à une base de données. Vous allez maintenant apprendre WebSocket.

HTTP est requête/réponse — la connexion se termine. WebSocket reste ouvert pour la communication temps réel.

```
le client se connecte → la connexion reste ouverte → le client envoie des événements → le serve
```

17.1 Quand utiliser WebSocket

Utilisez HTTP pour : les API CRUD, le chargement de pages, l'authentification, les API JSON normales.

Utilisez WebSocket pour : les messages en direct, les dashboards, la présence, le streaming d'événements, les notifications.

17.2 Headers publics

```
#include <vix.hpp>
#include <vix/websocket.hpp>
#include <vix/websocket/AttachedRuntime.hpp>  // pour HTTP + WebSocket ensemble
```

17.3 Modèle de message typé Vix

```
{ "type": "chat.message", "payload": { "user": "Ada", "text": "Hello" } }
```

17.4 Serveur WebSocket minimal

```cpp
#include <memory>
#include <vix.hpp>
#include <vix/websocket.hpp>

int main()
{
  vix::config::Config config{".env"};
  auto executor = std::make_shared<vix::executor::RuntimeExecutor>(1u);
  vix::websocket::Server ws{config, executor};

  ws.on_open([&ws](vix::websocket::Session &session){
    ws.broadcast_json(
      "system.connected",
      {"message", "A client connected"
    });
  });

  ws.on_message([&ws](vix::websocket::Session &session, const std::string &payload){
    ws.broadcast_json(
      "echo.raw",
      {"text", payload});
  });

  ws.on_typed_message([&ws](vix::websocket::Session &session,
                            const std::string &type,
                            const vix::json::kvs &payload){
    ws.broadcast_json(type, payload);
  });

  ws.listen_blocking();
  return 0;
}
```

17.5 HTTP + WebSocket ensemble

```cpp
struct BasicRuntime
{
  vix::config::Config config{".env"};
```

```
  std::shared_ptr<vix::executor::RuntimeExecutor> executor{
      std::make_shared<vix::executor::RuntimeExecutor>(1u)};
  vix::App app{executor};
  vix::websocket::Server ws{config, executor};
};
```

17.5.1 Enregistrer les routes HTTP

```
static void register_http_routes(vix::App &app)
{
  app.get("/", [](vix::Request &, vix::Response &res){
    res.json({"name", "Vix HTTP + WebSocket"});
  });

  app.get("/health", [](vix::Request &, vix::Response &res){
    res.json({ "status", "ok", "service", "http-ws" });
  });
}
```

17.5.2 Enregistrer le protocole WebSocket

```
static void register_ws_protocol(vix::websocket::Server &ws)
{
  ws.on_typed_message(
      [&ws](vix::websocket::Session &session,
            const std::string &type,
            const vix::json::kvs &payload)
    {
      if (type == "app.ping")
      {
        ws.broadcast_json("app.pong", {"status", "ok", "transport", "websocket"});
        return;
      }
      if (type == "chat.message")
      {
        ws.broadcast_json("chat.message", payload);
        return;
      }
      ws.broadcast_json("app.unknown", {"type", type, "message", "Unknown event type"});
  });
```

```
}
```

17.5.3 Exécuter les deux ensemble

```
vix::run_http_and_ws(runtime.app, runtime.ws, runtime.executor, runtime.config.getServerPort())
```

17.6 Alternative compacte : `serve_http_and_ws`

```
vix::serve_http_and_ws(".env", 8080, [](auto &app, auto &ws) {

  app.get("/", [](auto &, auto &res) {
    res.json({"framework", "Vix.cpp"});
  });

  ws.on_typed_message([&ws](auto &, const std::string &type, const vix::json::kvs &payload) {
    if (type == "chat.message") ws.broadcast_json("chat.message", payload);
  });

});
```

17.7 Protocole d'événements recommandé

Événement	Direction	Rôle
`chat.join`	Client -> serveur	Rejoint une room ou session de chat.
`chat.leave`	Client -> serveur	Quitte une room ou session de chat.
`chat.message`	Les deux directions	Envoie ou reçoit du contenu de chat.
`chat.error`	Serveur -> client	Signale une erreur liée au chat.
`app.ping`	Client -> serveur	Envoie une demande de health check.
`app.pong`	Serveur -> client	Retourne une réponse de health check.
`system.connected`	Serveur -> client	Confirme la connexion du client.
`system.disconnected`	Serveur -> client	Confirme la déconnexion du client.

17.8 Valider les payloads WebSocket

```
if (type == "chat.message")
{
  const std::string text = payload.get_string_or("text", "");
  if (text.empty())
  {
    ws.broadcast_json(
      "chat.error", {
        "error", "message_required",
        "message", "Message text is required"
      }
    );
    return;
  }
  ws.broadcast_json("chat.message", payload);
}
```

17.9 WebSocket et Nginx

```
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
proxy_read_timeout 3600;
proxy_send_timeout 3600;
```

17.10 Erreurs courantes

17.10.1 Utiliser WebSocket pour du CRUD normal

Utilisez HTTP pour create/read/update/delete. Utilisez WebSocket uniquement pour les événements en direct.

17.10.2 Diffuser des payloads non validés

Validez les payloads avant de les diffuser.

17.10.3 Oublier les headers d'upgrade Nginx

Sans les headers d'upgrade, WebSocket échoue à travers Nginx.

17.10.4 Ne pas gérer les types d'événements inconnus

Ayez toujours un fallback pour les types inconnus retournant `app.unknown`.

17.11 Ce qu'il faut retenir

HTTP est requête/réponse. WebSocket est une connexion temps réel de longue durée.

Vix WebSocket utilise : `Server`, `Session`, `on_open`, `on_close`, `on_error`, `on_message`, `on_typed_message`, `broadcast_json`.

L'idée centrale : utilisez HTTP pour les requêtes d'API normales et WebSocket pour les événements en direct qui doivent atteindre les clients immédiatement.

17.12 Chapitre suivant

Suivant : Runtime asynchrone

Chapitre 18

Runtime asynchrone

Dans le chapitre précédent, vous avez appris WebSocket. Vous allez maintenant apprendre le runtime asynchrone.

```
le code synchrone bloque tout le thread.
le code asynchrone permet à de nombreuses opérations de s'exécuter ensemble sans bloquer.
```

Une application backend doit gérer de nombreuses connexions, de nombreuses requêtes lentes, des appels réseau, des timers, de l'I/O fichier et la concurrence sans s'effondrer.

18.1 Pourquoi un runtime asynchrone

Sans runtime asynchrone, votre code doit créer des threads manuellement, gérer les durées de vie, synchroniser l'accès partagé, attendre l'I/O, gérer les timers et coordonner les opérations.

Le runtime asynchrone de Vix vous donne :

- une boucle d'événements (`io_context`),
- un scheduler,
- des coroutines C++20 (`task<T>`),
- des timers,
- un thread pool,
- des helpers réseau (TCP / UDP),
- la gestion de signaux,
- la composition avec `when_all` et `when_any`.

18.2 Headers publics

```
#include <vix/async/io_context.hpp>
#include <vix/async/scheduler.hpp>
#include <vix/async/task.hpp>
#include <vix/async/timer.hpp>
#include <vix/async/thread_pool.hpp>
#include <vix/async/signal_set.hpp>
#include <vix/async/net.hpp>
#include <vix/async/when_all.hpp>
#include <vix/async/when_any.hpp>
```

18.3 La boucle d'événements

```
vix::async::io_context io;
io.run();
```

`io_context` est la boucle d'événements centrale. Tout ce qui est asynchrone tourne autour d'elle :

- les requêtes HTTP,
- les sessions WebSocket,
- les timers,
- l'I/O réseau,
- la composition de tâches.

18.4 Le scheduler

```
vix::async::Scheduler scheduler{io};
```

Le scheduler décide quand reprendre les coroutines. Il poste le travail sur la boucle d'événements.

18.5 La tâche

```
vix::async::task<int> compute_value()
{
  co_return 42;
}
```

Une `task<T>` représente du travail asynchrone qui produira un résultat plus tard.

Vous l'exécutez :

```
auto value = co_await compute_value();
```

ou :

```
vix::async::run(io, compute_value());
```

18.6 Timer

```
vix::async::Timer timer{io};
timer.expires_after(std::chrono::seconds(1));
co_await timer.async_wait();
```

Les timers permettent d'attendre une durée sans bloquer le thread.

Cas d'utilisation : retry après une panne, polling, animations, rate limiting interne.

18.7 Thread pool

```
vix::async::ThreadPool pool{4};
auto result = co_await pool.async([] { return heavy_work(); });
```

Le thread pool exécute des fonctions bloquantes en arrière-plan sans bloquer la boucle d'événements.

Utilisez-le pour :

- les calculs lourds,
- le code legacy synchrone,
- les appels d'API bloquantes,
- les opérations CPU intensives.

18.8 Signal set

```
vix::async::SignalSet signals{io, SIGINT, SIGTERM};
co_await signals.async_wait();
```

Utilisez `SignalSet` pour un arrêt propre quand l'utilisateur appuie sur Ctrl+C ou quand systemd envoie un signal.

18.9 Réseau (TCP)

```
vix::async::net::tcp::Acceptor acceptor{io, 9000};

while (true){
  auto socket = co_await acceptor.async_accept();
  // gérer la connexion
}
```

18.9.1 Lire et écrire

```
char buffer[1024];
const std::size_t bytes_read = co_await socket.async_read(buffer, sizeof(buffer));
co_await socket.async_write(buffer, bytes_read);
```

18.10 Réseau (UDP)

```
vix::async::net::udp::Socket socket{io, 9001};
char buffer[1024];
const auto [bytes, endpoint] = co_await socket.async_receive_from(buffer, sizeof(buffer));
co_await socket.async_send_to(buffer, bytes, endpoint);
```

18.11 Composer plusieurs tâches

18.11.1 when_all

```
auto [a, b, c] = co_await vix::async::when_all(
    fetch_user(id),
    fetch_orders(id),
    fetch_recommendations(id)
);
```

Attend que toutes les tâches se terminent.

18.11.2 when_any

```
auto result = co_await vix::async::when_any(
    fetch_from_primary(),
    fetch_from_replica()
);
```

Retourne dès que la première tâche se termine.

18.12 Exemple : serveur TCP echo

```cpp
#include <vix/async/io_context.hpp>
#include <vix/async/task.hpp>
#include <vix/async/net.hpp>
#include <vix/async/signal_set.hpp>

vix::async::task<void> handle_connection(vix::async::net::tcp::Socket socket)
{
  char buffer[1024];
  try
  {
    while (true)
    {
      const std::size_t n = co_await socket.async_read(buffer, sizeof(buffer));
      if (n == 0) break;
      co_await socket.async_write(buffer, n);
    }
  }
  catch (const std::exception &) {
    // la connexion s'est fermée
  }
}

vix::async::task<void> run_server(vix::async::io_context &io)
{
  vix::async::net::tcp::Acceptor acceptor{io, 9000};

  while (true)
  {
    auto socket = co_await acceptor.async_accept();
    vix::async::spawn(io, handle_connection(std::move(socket)));
  }
}

int main()
{
  vix::async::io_context io;
```

```
  vix::async::SignalSet signals{io, SIGINT, SIGTERM};

  vix::async::spawn(io, run_server(io));

  vix::async::spawn(io, [&]() -> vix::async::task<void> {
    co_await signals.async_wait();
    io.stop();
  }());

  io.run();
  return 0;
}
```

Compilez et lancez :

```
vix run main.cpp
```

Testez avec `netcat` :

```
nc 127.0.0.1 9000
```

18.13 Le runtime asynchrone dans une application Vix

Une application HTTP Vix utilise déjà le runtime asynchrone en interne.

Pour les applications avancées qui mixent HTTP, WebSocket, timers et travail en arrière-plan :

```
struct Runtime
{
  vix::config::Config config{".env"};

  std::shared_ptr<vix::executor::RuntimeExecutor> executor{
      std::make_shared<vix::executor::RuntimeExecutor>(2u)
  };

  vix::App app{executor};
  vix::websocket::Server ws{config, executor};
};
```

L'`executor` partage la boucle d'événements entre les composants HTTP et WebSocket.

18.14 Erreurs courantes

18.14.1 Bloquer la boucle d'événements

```
// Incorrect : bloque la boucle pour tout le monde
std::this_thread::sleep_for(std::chrono::seconds(5));

// Correct : utilise un Timer
vix::async::Timer timer{io};
timer.expires_after(std::chrono::seconds(5));
co_await timer.async_wait();
```

18.14.2 Exécuter du travail CPU lourd sur la boucle d'événements

Pour les calculs lourds, utilisez le thread pool :

```
auto result = co_await pool.async([] { return heavy_work(); });
```

18.14.3 Oublier d'attendre `co_await`

```
// Incorrect : la tâche ne s'exécute pas
fetch_user(id);

// Correct
auto user = co_await fetch_user(id);
```

18.14.4 Mélanger threads et coroutines sans synchronisation

Toute interaction entre threads et la boucle d'événements doit passer par le scheduler ou le thread pool.

18.15 Ce qu'il faut retenir

Le runtime asynchrone de Vix repose sur :

```
io_context → scheduler → task<T> → coroutines C++20
```

Avec ces briques, vous obtenez :

- des timers,
- un thread pool,
- de l'I/O réseau,
- des signaux,

- la composition (`when_all`, `when_any`).

L'idée centrale : ne bloquez pas la boucle d'événements — laissez le runtime gérer la concurrence pour vous.

18.16 Chapitre suivant

Suivant : Cache

Chapitre 19

Cache

Dans le chapitre précédent, vous avez appris le runtime asynchrone. Vous allez maintenant apprendre le cache.

```
le cache stocke des résultats déjà calculés pour les retourner rapidement la fois suivante.
```

Un cache bien utilisé permet de réduire la charge sur la base de données, d'accélérer les API et de garder l'expérience utilisateur fluide même quand la source des données est lente.

19.1 Pourquoi le cache existe

Sans cache, chaque requête doit recalculer la réponse depuis la source : base de données, fichier, API distante. Le cache permet de réutiliser le résultat précédent quand il est encore valide.

```
client → cache (HIT) → réponse rapide
client → cache (MISS) → source de données → mémoriser → réponse
```

19.2 Header public

```
#include <vix/cache.hpp>
```

19.3 Concepts principaux

Concept	Rôle
`Cache`	Façade unique pour stocker et récupérer des entrées.
`CacheEntry`	Représente une valeur en cache avec ses métadonnées.

Concept	Rôle
`CachePolicy`	Décrit le TTL, la taille max et les règles d'éviction.
`CacheContext`	Regroupe le store et la policy.
`MemoryStore`	Backend en mémoire, rapide, volatile.
`FileStore`	Backend sur disque, persistant entre redémarrages.
`LruMemoryStore`	Cache en mémoire avec éviction LRU.
`CacheKey`	Clé typée et stable pour identifier une entrée.

19.4 Cache en mémoire basique

```
vix::cache::MemoryStore store;
vix::cache::CachePolicy policy;
policy.ttl = std::chrono::minutes(5);

vix::cache::Cache cache{store, policy};

cache.set("user:1", "Alice");
auto value = cache.get("user:1");
```

19.5 TTL (durée de vie)

```
vix::cache::CachePolicy policy;
policy.ttl = std::chrono::seconds(60);
```

Quand le TTL expire, l'entrée n'est plus servie.

TTL recommandé	Usage
5 à 30 secondes	Données très fraîches (feed, dashboard).
1 à 5 minutes	Listes d'API normales.
1 à 24 heures	Métadonnées de produits, configuration.
Plusieurs jours	Assets statiques, données rarement modifiées.

19.6 Cache LRU

```
vix::cache::LruMemoryStore store{1024};   // capacité max
vix::cache::CachePolicy policy;
policy.ttl = std::chrono::minutes(10);
```

```
vix::cache::Cache cache{store, policy};
```

Quand le cache atteint sa capacité, l'entrée la moins récemment utilisée est évincée.

19.7 Cache fichier persistant

```
vix::cache::FileStore store{"./cache"};
vix::cache::Cache cache{store, vix::cache::CachePolicy{}};
```

Le cache fichier reste valide après le redémarrage de l'application.

19.8 Pattern : cache aside

```
auto user = cache.get("user:" + id);

if (!user.has_value())
{
  user = load_user_from_db(id);
  cache.set("user:" + id, *user);
}

return *user;
```

C'est le pattern le plus courant : lire le cache d'abord, sinon charger depuis la source et mémoriser.

19.9 Helper `get_or_compute`

```
auto user = cache.get_or_compute("user:" + id, [&] {
  return load_user_from_db(id);
});
```

Si l'entrée existe, retourne directement. Sinon, exécute le calcul, mémorise et retourne.

19.10 Clés stables

```
const std::string key = vix::cache::CacheKey()
    .add("users")
    .add("list")
```

```
    .add("page", page)
    .add("limit", limit)
    .build();
```

Une clé stable est :

- déterministe (mêmes entrées → même clé),
- explicite (lisible dans les logs),
- isolée (préfixée par module ou ressource).

19.11 Invalidation

```
cache.invalidate("user:" + id);

cache.invalidate_prefix("users:");
```

Invalidation explicite à chaque écriture :

```
auto user = update_user_in_db(id, payload);
cache.invalidate("user:" + std::to_string(id));
cache.invalidate_prefix("users:list:");
```

19.12 Cache dans une route Vix

```
app.get("/api/users/{id}", [&](Request &req, Response &res){
  const std::string id = req.param("id");

  auto cached = cache.get("user:" + id);
  if (cached.has_value())
  {
    res.json({ "ok", true, "data", *cached, "from", "cache" });
    return;
  }

  auto user = find_user_by_id(db, std::stoll(id));
  if (!user) {
    res.status(404).json({ "ok", false, "error", "user_not_found" });
    return;
  }

  cache.set("user:" + id, user_to_json(*user));
```

```
  res.json({ "ok", true, "data", user_to_json(*user), "from", "db" });
});
```

19.13 Headers HTTP Cache-Control

Le cache côté serveur peut être combiné avec le cache navigateur :

```
res.header("Cache-Control", "public, max-age=300");
```

Directive	Effet
`no-cache`	Le client revalide à chaque requête.
`no-store`	Aucun cache, aucune copie.
`public, max-age=300`	Le navigateur peut cacher 5 minutes.
`private, max-age=60`	Cache utilisateur uniquement (pas de proxy).

19.14 Quand ne pas cacher

- Endpoints de login.
- Endpoints d'authentification.
- Données qui changent à chaque requête.
- Réponses dépendant fortement d'un utilisateur unique.
- Endpoints d'admin sensibles.

19.15 Erreurs courantes

19.15.1 Cacher des données utilisateur sans clé par utilisateur

```
// Incorrect : la même clé pour tous les utilisateurs
cache.set("profile", profile);

// Correct
cache.set("profile:" + user_id, profile);
```

19.15.2 Oublier l'invalidation après une écriture

```
update_user_in_db(id, payload);
cache.invalidate("user:" + std::to_string(id));
```

19.15.3 TTL trop long pour des données fréquemment modifiées

Un TTL de 24h sur un dashboard temps réel rend l'application fausse.

19.15.4 Cacher des réponses d'erreur

Évitez de cacher les codes 4xx et 5xx ; ils doivent être recalculés à chaque tentative.

19.16 Ce qu'il faut retenir

Le cache de Vix repose sur :

```
Cache → CacheStore (MemoryStore | LruMemoryStore | FileStore) + CachePolicy (TTL, taille)
```

Patterns à utiliser :

- cache aside,
- `get_or_compute`,
- invalidation explicite après écriture,
- clés stables et préfixées.

L'idée centrale : un cache bien utilisé accélère votre application sans la rendre incohérente.

19.17 Chapitre suivant

Suivant : Synchronisation offline-first

Chapitre 20

Synchronisation offline-first

Dans le chapitre précédent, vous avez appris le cache. Vous allez maintenant apprendre la synchronisation offline-first.

```
offline-first signifie que l'application continue de fonctionner même sans réseau, puis se synchronise dès qu'elle est connectée.
```

C'est essentiel pour les applications mobiles, les outils de terrain, les applications PWA et tout système qui doit rester utilisable malgré une connexion intermittente.

20.1 Pourquoi la synchronisation offline-first

Sans support offline, une coupure réseau bloque l'utilisateur. Avec une approche offline-first :

- les écritures sont mises en file localement,
- elles sont rejouées vers le serveur dès le retour de la connexion,
- les conflits sont résolus selon une stratégie claire.

20.2 Header public

```
#include <vix/sync.hpp>
```

20.3 Concepts principaux

Concept	Rôle
`Operation`	Représente une écriture locale à synchroniser.
`WAL`	Write-Ahead Log : journal append-only des opérations.

Concept	Rôle
`Outbox`	File durable des opérations en attente d'envoi.
`RetryPolicy`	Stratégie de retry (backoff, nombre max).
`NetworkProbe`	Détecte l'état du réseau (online/offline).
`SyncWorker`	Boucle de fond qui rejoue les opérations vers le serveur.
`SyncEngine`	Orchestre WAL + outbox + worker + retry.
`ISyncTransport`	Interface vers le backend distant (HTTP, WebSocket…).

20.4 Flux global

```
écriture utilisateur
  ↓
écriture locale (DB)
  ↓
ajout dans le WAL
  ↓
ajout dans l'outbox
  ↓
SyncWorker → ISyncTransport → serveur
  ↓
ACK reçu
  ↓
opération retirée de l'outbox
```

20.5 Définir une opération

```cpp
struct CreateNoteOp
{
  std::string id;       // identifiant client
  std::string title;
  std::string body;
  std::int64_t created_at_ms{};
};
```

Sérialisez-la en JSON pour le WAL et l'outbox :

```cpp
vix::json::Json to_json(const CreateNoteOp &op)
{
  return vix::json::kv({
```

```
    {"id", vix::json::Json(op.id)},
    {"title", vix::json::Json(op.title)},
    {"body", vix::json::Json(op.body)},
    {"created_at_ms", vix::json::Json(op.created_at_ms)}
  });
}
```

20.6 Write-Ahead Log (WAL)

```
vix::sync::WAL wal{"data/wal.log"};
wal.append("note.create", to_json(op).dump());
```

Le WAL est append-only. Il est rejoué au démarrage de l'application :

```
wal.replay([](const std::string &type, const std::string &payload){
  // appliquer l'opération à l'état local
});
```

20.7 Outbox

```
vix::sync::Outbox outbox{"data/outbox.db"};
outbox.enqueue("note.create", to_json(op).dump());
```

L'outbox est durable : les opérations restent jusqu'à confirmation par le serveur.

20.8 RetryPolicy

```
vix::sync::RetryPolicy policy;
policy.max_attempts = 8;
policy.initial_backoff = std::chrono::seconds(1);
policy.max_backoff = std::chrono::minutes(5);
policy.jitter = true;
```

Paramètre	Effet
`max_attempts`	Nombre maximum de tentatives par opération.
`initial_backoff`	Délai initial entre deux tentatives.
`max_backoff`	Plafond du délai entre deux tentatives.
`jitter`	Ajoute une variation aléatoire pour éviter les pics.

20.9 NetworkProbe

```
vix::sync::NetworkProbe probe;

probe.on_online([] {
  vix::log::info("network is back online");
});

probe.on_offline([] {
  vix::log::warn("network is offline");
});
```

Le probe surveille la connectivité et notifie le `SyncWorker`.

20.10 ISyncTransport

```
class HttpSyncTransport final : public vix::sync::ISyncTransport
{
public:
  vix::async::task<vix::sync::SendResult>
  send(const vix::sync::OutboxItem &item) override
  {
    // construire une requête HTTP
    // l'envoyer
    // retourner le résultat
    co_return vix::sync::SendResult::ack();
  }
};
```

`ISyncTransport` est l'abstraction entre le moteur de sync et le réseau.

20.11 SyncWorker

```
vix::sync::SyncWorker worker{
  outbox,
  transport,
  probe,
  policy
};
```

```
worker.start();
```

Le worker :

- attend des éléments dans l'outbox,
- attend que le réseau soit online,
- envoie chaque élément via le transport,
- applique le retry en cas d'échec,
- retire l'élément après ACK.

20.12 SyncEngine

```
vix::sync::SyncEngine engine{
  wal,
  outbox,
  worker
};

engine.start();
```

Le `SyncEngine` rejoint WAL + Outbox + Worker en un point d'entrée unique.

20.13 Pattern d'écriture utilisateur

```
void create_note(const std::string &title, const std::string &body)
{
  CreateNoteOp op;
  op.id = generate_uuid();
  op.title = title;
  op.body = body;
  op.created_at_ms = now_ms();

  // 1. écrire localement
  save_note_locally(op);

  // 2. journaliser dans le WAL
  wal.append("note.create", to_json(op).dump());

  // 3. mettre en file dans l'outbox
  outbox.enqueue("note.create", to_json(op).dump());
}
```

L'utilisateur voit immédiatement la note, même hors ligne. Elle sera synchronisée plus tard automatiquement.

20.14 Résolution de conflits

Stratégies courantes :

Stratégie	Description
Last-write-wins	La dernière écriture gagne (par timestamp).
Server-wins	Le serveur a toujours raison.
Client-wins	Le client a toujours raison.
Merge	Fusionner les champs (par exemple sets, compteurs).
CRDT	Structures conçues pour la convergence automatique.

Choisissez la stratégie au cas par cas selon la nature de la donnée.

20.15 Idempotence

Chaque opération doit avoir un ID stable côté client. Si la même opération arrive deux fois sur le serveur, elle ne doit pas créer deux ressources.

```
op.id = generate_uuid();
```

Le serveur vérifie `op.id` et ignore les doublons.

20.16 Erreurs courantes

20.16.1 Ne pas générer d'ID côté client

Sans ID stable, vous ne pouvez pas dédupliquer ni résoudre les conflits.

20.16.2 Ne pas écrire dans le WAL avant l'outbox

Si l'application crashe entre les deux, l'état local et l'outbox divergent.

20.16.3 Retry sans backoff

Un retry agressif peut surcharger un serveur déjà en difficulté.

20.16.4 Confondre "envoyé" et "appliqué"

Une opération envoyée n'est pas forcément acceptée. Attendez l'ACK avant de la retirer de l'outbox.

20.17 Ce qu'il faut retenir

La synchronisation offline-first de Vix repose sur :

```
WAL → Outbox → SyncWorker → ISyncTransport → ACK
```

Avec :

- des opérations idempotentes,
- un retry avec backoff,
- une détection réseau,
- une stratégie de résolution de conflits claire.

L'idée centrale : l'application doit rester utilisable même sans réseau, et converger vers le serveur dès que possible.

20.18 Chapitre suivant

Suivant : P2P

Chapitre 21

P2P

Dans le chapitre précédent, vous avez appris la synchronisation offline-first. Vous allez maintenant apprendre le P2P.

```
P2P signifie peer-to-peer : les nœuds se parlent directement,
sans dépendre uniquement d'un serveur central.
```

P2P est utile pour les outils local-first, les applications collaboratives, la synchronisation entre appareils sur un même réseau, les meshes IoT et les applications offline qui veulent partager des données sans cloud obligatoire.

21.1 Pourquoi P2P

Quand deux instances de la même application tournent sur le même réseau local, il est inutile de toujours passer par Internet. Vix propose un runtime P2P qui permet aux pairs de :

- se découvrir automatiquement,
- établir une connexion fiable,
- s'envoyer des messages typés,
- se synchroniser entre eux.

21.2 Header public

```
#include <vix/p2p.hpp>
```

21.3 Concepts principaux

Concept	Rôle
`Discovery`	Trouve les pairs sur le réseau (UDP broadcast / mDNS).
`Peer`	Représente un autre nœud distant.
`Handshake`	Échange initial d'identité et de capacités.
`Envelope`	Conteneur typé pour un message échangé entre pairs.
`Framing`	Encodage de longueur pour découper les messages sur TCP.
`Dispatch`	Route un message reçu vers le handler approprié.
`SyncMessage`	Message dédié à la synchronisation d'état (offline-first + P2P).
`P2PRuntime`	Orchestre Discovery + Peer + Dispatch.

21.4 Flux global

```
nœud A démarre
  ↓
Discovery envoie un broadcast UDP
  ↓
nœud B reçoit le broadcast
  ↓
B se connecte à A en TCP
  ↓
handshake (id, version, capabilities)
  ↓
échange d'enveloppes typées
  ↓
Dispatch → handler local
```

21.5 Discovery UDP

```cpp
vix::p2p::Discovery discovery;

discovery.on_peer_found([](const vix::p2p::PeerInfo &info){
  vix::log::info("peer found", "id", info.id, "address", info.address);
});

discovery.start(9100);   // port UDP de broadcast
```

21.6 Connexion à un pair

```
vix::p2p::Peer peer{io, "192.168.1.42", 9200};
co_await peer.connect();
```

21.7 Handshake

```
vix::p2p::Handshake hs;
hs.node_id = local_node_id;
hs.version = "1.0";
hs.capabilities = {"sync.notes", "chat.text"};

co_await peer.send_handshake(hs);
auto remote_hs = co_await peer.receive_handshake();
```

Le handshake permet à chaque côté de :

- s'identifier de manière stable,
- vérifier la compatibilité de version,
- annoncer les capacités supportées.

21.8 Envelope

```
struct Envelope
{
  std::string type;        // par ex. "chat.message"
  std::string from;        // node id émetteur
  std::string to;          // node id destinataire (peut être vide pour broadcast)
  vix::json::Json payload;
};
```

L'envelope est sérialisée en JSON puis encadrée par un framing de longueur sur TCP.

21.9 Framing

```
[4 octets : longueur big-endian][N octets : payload JSON]
```

Cela permet de découper proprement les messages sur un flux TCP continu.

21.10 Dispatch

```
vix::p2p::Dispatch dispatch;

dispatch.on("chat.message", [](const vix::p2p::Envelope &env){
  vix::log::info("chat from", "peer", env.from, "text", env.payload.value("text", ""));
});

dispatch.on("sync.note.create", [](const vix::p2p::Envelope &env){
  apply_note_create(env.payload);
});
```

Le dispatch route chaque enveloppe selon son type.

21.11 SyncMessage

```
vix::p2p::SyncMessage msg;
msg.kind = "note.create";
msg.payload = to_json(note);

co_await peer.send_sync(msg);
```

`SyncMessage` est l'intégration directe entre P2P et la synchronisation offline-first : chaque opération de l'outbox peut être envoyée à un pair plutôt qu'à un serveur central.

21.12 P2PRuntime

```
vix::p2p::P2PRuntime runtime{io};

runtime.set_node_id(local_node_id);
runtime.register_handler("chat.message", on_chat_message);
runtime.register_handler("sync.note.create", on_sync_note_create);

runtime.start({
  .discovery_port = 9100,
  .listen_port = 9200
});
```

`P2PRuntime` regroupe Discovery, écoute TCP, handshake, dispatch et gestion des pairs.

21.13 Routes HTTP de contrôle

Pour le debug, il est utile d'exposer des routes HTTP qui regardent l'état P2P :

```
app.get("/p2p/peers", [&](Request &, Response &res){
  vix::json::Json items = vix::json::Json::array();
  for (const auto &p : runtime.peers())
  {
    items.push_back(vix::json::kv({
      {"id", vix::json::Json(p.id)},
      {"address", vix::json::Json(p.address)},
      {"connected", vix::json::Json(p.connected)}
    }));
  }
  res.json({ "ok", true, "peers", items });
});

app.post("/p2p/broadcast", [&](Request &req, Response &res){
  const auto &body = req.json();
  runtime.broadcast(body.value("type", ""), body.value("payload", vix::json::Json::object()));
  res.json({ "ok", true });
});
```

21.14 Sécurité

Sur un réseau local de confiance, le P2P peut être simple. Sur un réseau ouvert :

- chiffrez le canal (TLS),
- authentifiez chaque pair (clé publique),
- signez les enveloppes,
- limitez les capacités acceptées,
- appliquez du rate limiting par pair.

21.15 Quand utiliser P2P

Bien adapté pour :

- la synchronisation entre les appareils d'un même utilisateur,
- les applications collaboratives en local,
- les meshes IoT,
- les outils local-first,
- les démonstrations sans cloud.

Moins adapté pour :

- une API publique sur Internet,
- un service nécessitant une autorité centrale forte,
- les flux de paiement.

21.16 Erreurs courantes

21.16.1 Pas de handshake

Sans handshake, vous ne savez pas avec qui vous parlez ni quelle version il utilise.

21.16.2 Pas de framing

Sans framing de longueur, les messages se collent ou se coupent sur TCP.

21.16.3 Pas de timeout sur la connexion

Un pair lent peut bloquer indéfiniment l'`accept`.

21.16.4 Pas de limite sur les capacités

Un pair malveillant pourrait demander à exécuter des commandes non prévues.

21.17 Ce qu'il faut retenir

Le P2P de Vix repose sur :

```
Discovery (UDP) → Peer (TCP) → Handshake → Envelope (JSON + framing) → Dispatch
```

Avec `P2PRuntime` pour orchestrer le tout et `SyncMessage` pour réutiliser les opérations offline-first directement entre pairs.

L'idée centrale : permettre aux nœuds de coopérer directement, sans dépendre obligatoirement d'un serveur central.

21.18 Chapitre suivant

Suivant : Déploiement en production

Chapitre 22

Déploiement en production

Dans le chapitre précédent, vous avez appris le P2P. Vous allez maintenant apprendre comment déployer une application Vix en production.

```
navigateur → Nginx (HTTPS) → application Vix sur 127.0.0.1:8080 → systemd → logs
```

L'objectif est d'avoir une application qui démarre automatiquement, redémarre en cas de crash, reste accessible via HTTPS et s'exécute comme un service Linux normal.

22.1 Vue d'ensemble du déploiement

Un déploiement Vix typique comprend :

- un utilisateur Linux dédié,
- un build release,
- un fichier `.env` de configuration,
- un service systemd,
- un reverse proxy Nginx,
- un certificat TLS via Let's Encrypt,
- un pare-feu (UFW),
- des logs centralisés.

22.2 Préparer le serveur

22.2.1 Créer un utilisateur dédié

```
sudo adduser --system --group --home /home/vix vix
```

22.2.2 Créer le dossier d'application

```
sudo mkdir -p /home/vix/apps/myapp
sudo chown vix:vix /home/vix/apps/myapp
```

22.2.3 Installer les dépendances

```
sudo apt update
sudo apt install -y \
  build-essential cmake ninja-build pkg-config \
  libssl-dev libsqlite3-dev zlib1g-dev libbrotli-dev \
  nlohmann-json3-dev libspdlog-dev libfmt-dev \
  nginx certbot python3-certbot-nginx ufw
```

22.3 Construire le build release

Dans votre dépôt :

```
vix build --release
```

ou :

```
cmake -S . -B build/release -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build/release
```

Copiez ensuite le binaire et les fichiers nécessaires sur le serveur :

```
scp build/release/api vix@server:/home/vix/apps/myapp/api
scp .env.production vix@server:/home/vix/apps/myapp/.env
```

22.4 Fichier .env de production

```
SERVER_PORT=8080
SERVER_HOST=127.0.0.1

VIX_LOG_LEVEL=info
VIX_LOG_FORMAT=json
VIX_COLOR=never

DATABASE_ENGINE=sqlite
DATABASE_DEFAULT_NAME=/home/vix/apps/myapp/data/app.db

CORS_ALLOWED_ORIGINS=https://myapp.example.com
```

Sécurisez le fichier :

```
sudo chown vix:vix /home/vix/apps/myapp/.env
sudo chmod 600 /home/vix/apps/myapp/.env
```

22.5 Service systemd

Créez `/etc/systemd/system/myapp.service` :

```
[Unit]
Description=My Vix.cpp application
After=network.target

[Service]
Type=simple
User=vix
Group=vix
WorkingDirectory=/home/vix/apps/myapp
ExecStart=/home/vix/apps/myapp/api
EnvironmentFile=/home/vix/apps/myapp/.env
Restart=on-failure
RestartSec=3

# Durcissement de base
NoNewPrivileges=true
PrivateTmp=true
ProtectSystem=full
ProtectHome=true
ReadWritePaths=/home/vix/apps/myapp

# Logs
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
```

Activez et démarrez :

```
sudo systemctl daemon-reload
sudo systemctl enable myapp
sudo systemctl start myapp
sudo systemctl status myapp
```

Consultez les logs :

```
sudo journalctl -u myapp -f
```

22.6 Reverse proxy Nginx

Créez `/etc/nginx/sites-available/myapp.conf` :

```
server {
    listen 80;
    server_name myapp.example.com;

    location / {
        proxy_pass http://127.0.0.1:8080;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        # Support WebSocket
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_read_timeout 3600;
        proxy_send_timeout 3600;
    }
}
```

Activez le site :

```
sudo ln -s /etc/nginx/sites-available/myapp.conf /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl reload nginx
```

22.7 HTTPS avec Let's Encrypt

```
sudo certbot --nginx -d myapp.example.com
```

Certbot :

- obtient un certificat TLS,
- met à jour automatiquement la configuration Nginx pour le HTTPS,
- configure la redirection HTTP → HTTPS,

- installe un timer de renouvellement.

Vérifiez le renouvellement :

```
sudo systemctl list-timers | grep certbot
```

22.8 Pare-feu (UFW)

```
sudo ufw allow OpenSSH
sudo ufw allow 'Nginx Full'
sudo ufw enable
sudo ufw status
```

Notez que le port `8080` n'est PAS ouvert : l'application n'est joignable que via Nginx en local.

22.9 Health check

Exposez une route `/health` dans votre application et utilisez-la pour le monitoring :

```
curl -i https://myapp.example.com/health
```

Réponse attendue :

```
{ "ok": true, "service": "myapp" }
```

22.10 Mises à jour

Un déploiement classique ressemble à :

```
# sur la machine de build
vix build --release

# transfert
scp build/release/api vix@server:/home/vix/apps/myapp/api.new

# sur le serveur
sudo systemctl stop myapp
sudo mv /home/vix/apps/myapp/api.new /home/vix/apps/myapp/api
sudo systemctl start myapp
sudo systemctl status myapp
```

Pour des mises à jour sans coupure :

- déployer une nouvelle version sur un autre port,

- basculer Nginx vers le nouveau backend,
- arrêter l'ancienne version.

22.11 Sauvegardes

Sauvegardez régulièrement :

- la base de données (`app.db`),
- les fichiers utilisateur,
- la configuration `.env`,
- les certificats TLS (gérés par Let's Encrypt mais récupérables).

Exemple SQLite :

```
sqlite3 /home/vix/apps/myapp/data/app.db ".backup /backups/app-$(date +%F).db"
```

22.12 Observabilité

Logs :

```
sudo journalctl -u myapp --since "1 hour ago"
sudo journalctl -u myapp -f
```

Métriques :

- exposer `/metrics` si vous utilisez Prometheus,
- ajouter un middleware de journalisation pour chaque requête,
- inclure un `request_id` dans chaque log.

22.13 Sécurité opérationnelle

- Désactivez le login root SSH.
- Utilisez des clés SSH plutôt que des mots de passe.
- Tenez le système à jour : `sudo apt update && sudo apt upgrade`.
- Limitez les ports ouverts au strict minimum.
- Ne journalisez jamais de secrets.
- Mettez en place un rate limit Nginx pour les endpoints sensibles.

Exemple de rate limit Nginx :

```
limit_req_zone $binary_remote_addr zone=login_zone:10m rate=5r/m;

location /auth/login {
    limit_req zone=login_zone burst=5 nodelay;
    proxy_pass http://127.0.0.1:8080;
```

```
}
```

22.14 Erreurs courantes

22.14.1 Exposer le port applicatif directement

N'ouvrez pas `8080` dans le pare-feu : passez par Nginx.

22.14.2 Stocker `.env` en clair dans Git

Le fichier `.env` de production ne doit pas être versionné.

22.14.3 Ignorer les headers WebSocket Nginx

Sans `Upgrade` / `Connection`, WebSocket échoue à travers Nginx.

22.14.4 Pas de `Restart=on-failure` dans systemd

Sans cela, un crash arrête définitivement le service.

22.14.5 Logs en mode `debug` en production

Les logs `debug` ou `trace` en production saturent le disque et fuitent des détails sensibles.

22.15 Ce qu'il faut retenir

Un déploiement Vix robuste combine :

```
build release → utilisateur dédié → systemd → Nginx → HTTPS → UFW → logs → backups
```

Avec :

- un fichier `.env` privé,
- un health check,
- des mises à jour contrôlées,
- une observabilité claire.

L'idée centrale : une application en production est un service Linux normal, géré par les outils standards : systemd, Nginx, certbot, ufw, journalctl.

22.16 Chapitre suivant

Suivant : Étapes suivantes

Chapitre 23

Étapes suivantes

Vous avez parcouru tout Le Livre Vix.

Vous avez appris à :

- créer un projet Vix,
- exécuter un fichier C++ en une commande,
- construire un serveur HTTP,
- exposer des routes,
- lire les requêtes et écrire les réponses,
- construire des API JSON,
- ajouter du middleware (CORS, rate limiting, authentification),
- valider les entrées,
- gérer les erreurs et journaliser,
- connecter une base de données (SQLite, MySQL),
- ajouter du temps réel via WebSocket,
- utiliser le runtime asynchrone (coroutines, timers, thread pool),
- mettre en cache des résultats,
- supporter le mode offline-first avec WAL et outbox,
- partager des données entre pairs avec P2P,
- déployer en production avec Nginx, systemd et HTTPS.

23.1 Récapitulatif du modèle mental

```
CLI → Runtime → Application → Modules
```

L'application est centrée sur :

```
App → route → Request → Response
```

Et grandit avec des modules :

```
JSON, middleware, validation, database, websocket, async, cache, sync, p2p
```

23.2 Checklist d'une application Vix prête pour la production

- ☐ `vix build --release` réussit sans warnings.
- ☐ `vix tests` passe.
- ☐ `vix check` passe.
- ☐ Le code est formaté avec `vix fmt`.
- ☐ Une route `/health` existe.
- ☐ Les routes sensibles ont du rate limiting.
- ☐ CORS est restreint au vrai domaine du frontend.
- ☐ La validation est en place pour toute entrée utilisateur.
- ☐ Les erreurs ont une forme stable (`{ ok, error, message }`).
- ☐ Les logs sont structurés (JSON) en production.
- ☐ Les secrets ne sont pas journalisés.
- ☐ La base de données utilise des requêtes préparées.
- ☐ Les transactions sont utilisées pour les écritures multi-étapes.
- ☐ systemd redémarre l'application en cas de crash.
- ☐ Nginx termine TLS et proxie vers `127.0.0.1`.
- ☐ UFW limite les ports exposés.
- ☐ Des sauvegardes régulières de la base de données existent.

23.3 Bonnes habitudes au quotidien

Pendant le développement :

```
vix dev
```

Avant chaque commit :

```
vix fmt
vix check
vix tests
```

Avant une release :

```
vix build --release
```

23.4 Continuer à apprendre

- Lisez la documentation officielle : https://docs.vixcpp.com

- Explorez le dépôt : https://github.com/vixcpp/vix
- Lisez le code source des modules qui vous intéressent (`vix::db`, `vix::websocket`, `vix::async`, `vix::sync`, `vix::p2p`).
- Construisez une vraie application : c'est le meilleur moyen d'ancrer ce que vous venez d'apprendre.

23.5 Idées de projets pour pratiquer

1. Une API REST de gestion de tâches avec SQLite, validation et authentification par token.
2. Un service de chat temps réel basé sur WebSocket.
3. Une application offline-first (notes ou journal) avec WAL et synchronisation.
4. Un outil P2P de partage de fichiers sur LAN.
5. Un proxy HTTP avec rate limiting et cache.
6. Un dashboard backend avec métriques temps réel.
7. Un bot ou worker en arrière-plan utilisant le runtime asynchrone et un thread pool.

23.6 Contribuer à Vix.cpp

Vix.cpp est un projet open source. Vous pouvez :

- ouvrir des issues sur GitHub,
- proposer des améliorations de documentation,
- soumettre des pull requests,
- partager vos retours d'expérience,
- écrire des articles ou des tutoriels.

23.7 Mot de la fin

Vix.cpp est un effort pour rendre le C++ moderne plus direct, plus pratique et plus agréable à utiliser pour construire des applications réelles.

Vous avez maintenant les bases nécessaires pour construire des applications C++ rapides, fiables et maintenables — du premier fichier `main.cpp` jusqu'au déploiement en production.

> Garder la puissance du C++, rendre le workflow applicatif plus simple.

Bon code, et à très vite dans la communauté Vix.cpp.

— Gaspard Kirira