OFFICIAL DOCUMENTATION

# Vix.cpp Documentation

## Getting Started and The Vix Book

*A practical guide to building fast, reliable C++ applications with Vix.cpp.*


AUTHOR

Gaspard Kirira



**Vix.cpp Project**

github.com/vixcpp/vix

docs.vixcpp.com

# About this document

This document is the official combined documentation for **Vix.cpp**, a modern C++ runtime and developer toolkit for building fast, reliable applications with a smoother workflow.

It is composed of two parts. **Part 1 — Getting Started** walks you through the shortest path from installation to a running HTTP server. **Part 2 — The Vix Book** then explains the mental model behind Vix and progressively covers routes, requests, responses, JSON APIs, middleware, validation, error handling and logging, database access, real-time WebSocket, the async runtime, cache, offline-first synchronization, peer-to-peer messaging, and production deployment.

The two parts are designed to be read in order. Each chapter builds on the previous one, so you can move from running a single C++ file to operating a production service deployed behind Nginx and systemd without skipping a step.

All code samples, command-line invocations, configuration blocks, and routes are presented exactly as they appear in the official source documentation. The `--run` versus `--` distinction in the Vix CLI is highlighted wherever it matters.

> **Project:** Vix.cpp
> **Author:** Gaspard Kirira
> **Repository:** `github.com/vixcpp/vix`
> **Documentation:** `docs.vixcpp.com`
> **Generated:** May 13, 2026

# Table of Contents

## PART 1 — GETTING STARTED



## PART 2 — THE VIX BOOK

# Part 1

## Getting Started

*This section introduces the shortest path to running a Vix application. You will install Vix, set up your environment, run your first C++ file, create your first project, and build your first HTTP server.*

# 1. Welcome to Vix.cpp

Vix.cpp is a modern C++ runtime and developer toolkit for building fast, reliable applications with a smoother workflow.

It gives C++ a direct development experience:

```
vix run main.cpp
```

And a project workflow:

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

## What is Vix.cpp?

Vix.cpp helps you build C++ applications without starting every project by manually wiring the build system, runtime commands, logs, dependencies, and development workflow.

The goal is simple:

> *Keep the power of C++, make the application workflow simpler.*

Vix does not replace C++. It gives C++ a runtime-oriented workflow around it.

## What you can build

With Vix, you can build:

- HTTP servers
- JSON APIs
- backend services
- WebSocket apps
- CLI tools
- C++ libraries
- template-based web apps
- local-first and offline-first systems
- production services behind Nginx and systemd

A minimal Vix HTTP app looks like this:

```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;
}
```

Run it:

```
vix run main.cpp
```

Then open:

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

## The core workflow

For a single C++ file:

```
vix run main.cpp
```

For a real project:

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

For development mode:

```
vix dev
```

For checks and tests:

```
vix check
vix tests
```

## How Getting Started is organized

This section gives you the shortest path from zero to a running Vix application.

Read it in order:

1. Installation
2. Set Up Your Environment
3. Run Your First C++ File
4. Create Your First Project
5. Your First HTTP Server

## Getting Started vs The Vix Book

Getting Started is short and practical.

It helps you:

```
install → verify → run → create project → start server
```

The Vix Book goes deeper.

It explains the mental model behind Vix, then teaches routes, requests, responses, JSON APIs, middleware, validation, database, WebSocket, async runtime, cache, sync, P2P, and production deployment.

Start here first. Then continue with the book when you want to understand Vix step by step.

## What you need

You only need basic C++ knowledge:

- functions
- headers
- `std::string`
- lambdas
- basic terminal usage

You do not need to be a CMake expert to start. Vix can create a project, build it, run it, and give you a clean development loop.

## First command to remember

```
vix run main.cpp
```

This command is the fastest way to run a C++ file with Vix.

When your app grows, move to a project:

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

## Next step

Install Vix on your machine.

Next: **Installation**

# 2. Installation

This page shows how to install Vix.cpp and verify that it works on your machine.

For Getting Started, install the full SDK.

The full SDK includes the `vix` CLI, headers, and libraries needed to build Vix applications.

## Install on Linux or macOS

Run:

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

After installation, restart your terminal or reload your shell configuration.

## Install on Windows

Open PowerShell and run:

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

## Verify the installation

Check that the `vix` command is available:

```
vix --version
```

Expected output shape:

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

The exact version may be different depending on the latest release.

## Verify the SDK headers

For C++ applications using Vix, the SDK headers must be installed.

Check that `vix.hpp` exists:

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

Expected output:

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

If nothing appears, reinstall the full SDK.

## SDK mode vs CLI-only mode

Vix has two installation modes.

| Mode | What it installs | Use when |
|---|---|---|
| SDK mode | CLI, headers, and libraries | You want to build Vix applications |
| CLI-only mode | Only the `vix` binary | You only need the CLI |

For this Getting Started guide, use the default SDK mode.

Do not use CLI-only mode if you want to compile code that includes:

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

## CLI-only mode

CLI-only mode installs only the command-line tool.

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

This is not recommended for this guide because the next pages build real Vix applications.

## Fix PATH issues

If your terminal says:

```
vix: command not found
```

Add `~/.local/bin` to your `PATH`.

### Bash

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

### Zsh

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

Then check again:

```
vix --version
```

## Install build prerequisites

Vix uses the normal C++ toolchain underneath.

Make sure your system has a compiler, CMake, Ninja, and common development libraries.

### Ubuntu or 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
```

### macOS

With Homebrew:

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

### Windows

Install Visual Studio Build Tools with MSVC or clang-cl.

For extra dependencies, use vcpkg.

## Check your toolchain

Run:

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

If one of these commands is missing, install the missing tool before continuing.

## Quick test

Create a temporary folder:

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

Create `main.cpp`:

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

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

Run it:

```
vix run main.cpp
```

Expected output:

```
Hello from Vix
```

## Common installation problems

### `vix: command not found`

Your shell cannot find the Vix binary.

Fix:

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

Then run:

```
vix --version
```

### `#include <vix.hpp>` not found

The full SDK is not installed, or the headers are not visible.

Check:

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

If nothing appears, reinstall Vix without CLI-only mode:

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

### CMake or Ninja is missing

Check:

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

On Ubuntu or Debian:

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

## Upgrade Vix later

To upgrade the CLI:

```
vix upgrade
```

To inspect your environment:

```
vix doctor
```

To inspect Vix paths and cache information:

```
vix info
```

### What you should remember

Install the full SDK:

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

Verify the CLI:

```
vix --version
```

Verify the SDK headers:

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

For this guide, SDK mode is the correct installation mode.

## Next step

Now set up your development environment.

Next: **Set Up Your Environment**

# 3. Set Up Your Environment

This page helps you prepare a clean development environment for Vix.cpp.

Before writing real Vix applications, make sure your terminal, compiler, build tools, and project folder are ready.

## Check Vix

First, verify that the `vix` command is available:

```
vix --version
```

Expected output shape:

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

The exact version may be different.

## Check your C++ compiler

Vix uses your system C++ compiler underneath.

Run:

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

You should see a compiler such as GCC, Clang, MSVC, or clang-cl.

On Linux, GCC or Clang is recommended.

## Check CMake

Vix uses CMake for project builds.

Run:

```
cmake --version
```

If CMake is missing on Ubuntu or Debian:

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

## Check Ninja

Ninja is the recommended build backend for Vix projects.

Run:

```
ninja --version
```

If Ninja is missing on Ubuntu or Debian:

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

## Check common development libraries

For most Vix applications, install the common C++ development dependencies.

### Ubuntu or 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
```

### macOS

With Homebrew:

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

### Windows

Install Visual Studio Build Tools with MSVC or clang-cl.

For optional libraries, use vcpkg.

## Recommended project folder

Create a clean folder for experiments:

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

You can use this folder for the first examples in this guide.

## Create a quick test file

Create `main.cpp`:

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

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

Run it with Vix:

```
vix run main.cpp
```

Expected output:

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

If this works, your basic C++ environment is ready.

## Test a Vix HTTP program

Now test that the Vix SDK headers and libraries are available.

Replace `main.cpp`:

```
cat > main.cpp <<'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;
}
CPP
```

Run it:

```
vix run main.cpp
```

Expected output shape:

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

In another terminal, test the server:

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

Expected response shape:

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

Stop the server with:

```
Ctrl+C
```

## Check useful Vix commands

These commands help you inspect your environment:

```
vix doctor
vix info
```

Use `vix doctor` when you want to check whether your toolchain is healthy.

Use `vix info` when you want to inspect paths, cache directories, and installation details.

## Recommended editor setup

You can use any editor.

Recommended setup:

| Tool | Recommendation |
|---|---|
| Editor | VS Code, CLion, Vim, Neovim, or Zed |
| Compiler | GCC or Clang on Linux/macOS, MSVC or clang-cl on Windows |
| Build system | CMake |
| Build backend | Ninja |
| Terminal | Bash, Zsh, PowerShell, or Windows Terminal |

For VS Code, install:

- C/C++ extension
- CMake Tools
- clangd, optional

## Recommended Git setup

If you plan to create real projects, configure Git:

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

Check:

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

## Environment variables

Vix applications often read configuration from `.env`.

A simple `.env` file can look like this:

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

Later, project templates can generate `.env` and `.env.example` files for you.

## Common issues

`vix: command not found`

Your terminal cannot find the Vix binary.

Fix your PATH:

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

For Zsh:

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

Then check:

```
vix --version
```

`c++: command not found`

Install a compiler.

On Ubuntu or Debian:

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

`cmake: command not found`

Install CMake:

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

`ninja: command not found`

Install Ninja:

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

`#include <vix.hpp> not found`

The full SDK is missing or not installed correctly.

Check:

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

If nothing appears, reinstall the full SDK:

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

Do not use CLI-only mode for this guide.

### Port 8080 is already in use

Find the process using the port:

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

Stop the process or change the port in your code.

### What you should remember

A good Vix environment has:

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

Check them with:

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

The most important test is:

```
vix run main.cpp
```

If that works, you are ready to write your first C++ file with Vix.

## Next step

Run your first C++ file with Vix.

Next: **Run Your First C++ File**

# 4. Run Your First C++ File

This page shows how to run a single C++ file with Vix.

The main command is:

```
vix run main.cpp
```

This is the fastest way to start using Vix.

## Create a workspace

Create a temporary folder:

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

Create `main.cpp`:

```
touch main.cpp
```

## Write your first C++ program

Open `main.cpp` and write:

```cpp
#include <iostream>

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

Run it:

```
vix run main.cpp
```

Expected output:

```
Hello from Vix
```

## What happened?

When you run:

```
vix run main.cpp
```

Vix detects that you passed a single `.cpp` file.

It then:

```
detects the file → compiles it → runs it → prints the output
```

You do not need to create a full project yet.

This mode is useful for:

- quick experiments
- learning Vix
- testing small ideas
- writing small C++ tools
- running examples

## Run a Vix HTTP file

Now replace `main.cpp` with a small Vix HTTP app:

```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;
}
```

Run it:

```
vix run main.cpp
```

Expected output shape:

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

Open another terminal and test the server:

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

Expected response:

```
Hello Vix
```

Stop the server with:

```
Ctrl+C
```

## Return JSON

Most Vix backend apps return JSON.

Update the route:

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

  app.run(8080);

  return 0;
}
```

Run it again:

```
vix run main.cpp
```

Test it:

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

## Add a route parameter

Vix routes can contain path parameters.

Add this route before `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"
  });
});
```

Run:

```
vix run main.cpp
```

Test:

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

Expected response shape:

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

## Add a query parameter

Query parameters come after `?` in the URL.

Add this route before `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
  });
});
```

Run:

```
vix run main.cpp
```

Test:

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

Expected response shape:

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

## Complete example

Your full `main.cpp` can look like this:

```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.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;
}
```

Run:

```
vix run main.cpp
```

Test:

```
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"
```

## Pass runtime arguments

Runtime arguments are arguments passed to your program.

Use `--run`:

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

Use this when your program reads `argv`.

Example:

```cpp
#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;
}
```

Run:

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

**Important: `--run` vs `--`**

In script mode, `--run` is for runtime arguments.

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

The `--` separator is for compiler or linker flags.

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

Do not use `--` for runtime arguments in script mode.

Wrong:

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

Correct:

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

## Pass compiler flags

Use `--` to pass compiler or linker flags:

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

Link with libraries:

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

Add include paths:

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

## Use watch mode

During development, you can rebuild and restart when the file changes:

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

For project development, you will usually use:

```
vix dev
```

You will learn this in the next pages.

## Use sanitizers

For debugging memory issues:

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

For undefined behavior checks:

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

### Common mistakes

#### Using CLI-only installation

If your code uses:

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

you need the full SDK.

Check:

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

If nothing appears, reinstall Vix:

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

#### Passing runtime args after `--`

Wrong:

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

Correct:

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

#### Port 8080 is already in use

If the server cannot start, another process may already be using port 8080.

Check:

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

Stop the other process or change the port:

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

#### Running from the wrong directory

Relative paths depend on the directory where you run `vix`.

For example:

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

will look for:

```
public/index.html
```

relative to your current working directory.

## When to create a project

A single file is perfect for learning.

Move to a project when you need:

- multiple `.cpp` files
- headers
- tests
- dependencies
- `.env` configuration
- a stable folder structure
- release builds

The next page shows how to create a real Vix project.

### What you should remember

The main command is:

```
vix run main.cpp
```

Use:

- `--run` for runtime arguments
- `--` for compiler and linker flags

Single-file mode is the fastest way to start.

When your app grows, create a project.

## Next step

Create your first Vix project.

Next: **Create Your First Project**

# 5. Create Your First Project

This page shows how to create your first real Vix project.

Until now, you used:

```
vix run main.cpp
```

That is great for a single file.

For a real application, use:

```
vix new api
```

A Vix project gives you a clean structure, a build workflow, tests, configuration files, and a better development loop.

## Create a project

Create a new project:

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

Vix will ask which template you want to use.

Choose:

```
Application
```

The output should look similar to this:

```
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
```

## Enter the project

```
cd api
```

## Build the project

Run:

```
vix build
```

Expected output shape:

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

This compiles the project without starting the application.

## Run the project

Run:

```
vix run
```

Expected output shape:

```
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
```

Open another terminal and test it:

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

Stop the server with:

```
Ctrl+C
```

## Development mode

For day-to-day development, use:

```
vix dev
```

`vix dev` starts the project in development mode.

Use it when you are editing code and want a faster development loop.

Then open:

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

## Generated project structure

A basic application project usually looks like this:

```
api/
├── CMakeLists.txt
├── CMakePresets.json
├── README.md
├── app.vix
├── vix.json
├── .env
├── .env.example
├── src/
│   └── main.cpp
└── tests/
    └── test_basic.cpp
```

## What each file does

| File or folder | Purpose |
|---|---|
| `src/` | Application source code. |
| `tests/` | Project tests. |
| `CMakeLists.txt` | C++ build configuration. |
| `CMakePresets.json` | Build presets used by CMake and Vix. |
| `app.vix` | Vix app manifest. |
| `vix.json` | Vix project metadata, dependencies, and tasks. |
| `.env` | Local runtime configuration. |
| `.env.example` | Example environment file. |
| `README.md` | Project documentation. |

## Open the entry file

Open:

```
src/main.cpp
```

A minimal generated app can look like this:

```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;
}
```

This is the main entrypoint of your application.

## Edit the first route

Replace the route with a JSON response:

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

Add a health route:

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

Your full `src/main.cpp` can look like this:

```cpp
#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;
}
```

Run:

```
vix dev
```

Test:

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

## Use `.env`

Vix projects can use `.env` files for local configuration.

Example:

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

In code, you can load configuration:

```cpp
#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;
}
```

Now the server port comes from `.env`.

## Useful project commands

Inside the project folder:

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

| Command | Purpose |
|---|---|
| `vix build` | Compile the project. |
| `vix run` | Build and start the app. |
| `vix dev` | Start development mode. |
| `vix check` | Validate the project. |
| `vix tests` | Run tests. |
| `vix fmt` | Format source files. |

## Project tasks

Some projects define tasks in `vix.json`.

You can run them with:

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

A `vix.json` file can contain tasks like:

```
{
  "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"
    }
  }
}
```

## Application vs library

When you run:

```
vix new api
```

Vix can create different kinds of projects.

For this guide, choose:

```
Application
```

Use Application when you want to build a runnable app, server, API, or service.

Use Library (header-only) when you want to build a reusable C++ library.

Example:

```
vix new tree
```

Then choose:

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

A library project is useful when you want to create reusable headers and tests, but it is not the main path for this Getting Started guide.

### Common mistakes

#### Running commands outside the project

Wrong:

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

Correct:

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

Run project commands from the project folder.

#### Forgetting to stop the previous server

If port 8080 is already in use, stop the previous server with:

```
Ctrl+C
```

Or find the process:

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

#### Editing files but not rebuilding

When using:

```
vix run
```

you may need to restart manually after changes.

For active development, use:

```
vix dev
```

#### Adding new `.cpp` files

If you add new source files, make sure they are part of the build.

For a CMake project, update `CMakeLists.txt`.

Example:

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

### What you should remember

Create an application project:

```
vix new api
cd api
```

Build it:

```
vix build
```

Run it:

```
vix run
```

Develop it:

```
vix dev
```

A Vix project is the point where a quick experiment becomes a real application.

## Next step

Build your first HTTP server with Vix.

Next: **Your First HTTP Server**

# 6. Your First HTTP Server

This page shows how to build your first HTTP server with Vix.cpp.

You will create:

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

The goal is to understand the core Vix HTTP model:

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

## Start from your project

Use the project created in the previous page:

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

Open:

```
src/main.cpp
```

Replace its content with this minimal server:

```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;
}
```

Run it:

```
vix dev
```

Open another terminal and test it:

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

Expected response body:

```
Hello world
```

## What this code does

```
App app;
```

Creates the Vix application.

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

Registers a `GET /` route.

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

Starts the HTTP server on port `8080`.

## Core concepts

| Part | Purpose |
|---|---|
| `#include <vix.hpp>` | Imports the main Vix API. |
| `using namespace vix;` | Lets you use `App`, `Request`, and `Response` directly. |
| `App app;` | Creates the HTTP application. |
| `app.get(...)` | Registers a GET route. |
| `Request &req` | Reads what the client sent. |
| `Response &res` | Sends what your app returns. |
| `app.run(8080)` | Starts the server. |

## Return JSON

Most backend services return JSON.

Replace the `/` route with:

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

Run:

```
vix dev
```

Test:

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

Expected response shape:

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

## Add a health route

A health route is useful for checking whether your server is alive.

Add this before `app.run(8080)`:

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

Test:

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

Expected response shape:

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

## Add a path parameter

Path parameters let you read values from the URL.

Add this 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"
  });
});
```

Test:

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

Expected response shape:

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

Here:

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

reads the `{name}` part from the route:

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

## Add a route with an ID

Add another route:

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

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

Test:

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

Expected response shape:

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

## Add query parameters

Query parameters come after `?` in the URL.

Update the `/users/{id}` route:

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

Test:

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

Expected response shape:

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

## Response methods

Vix gives you several response helpers:

```
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");
```

Use:

| Method | Use when |
|---|---|
| `res.send(...)` | You want a generic response. |
| `res.text(...)` | You want plain text. |
| `res.json(...)` | You want JSON. |
| `res.status(...).json(...)` | You want to set the HTTP status. |
| `res.header(...)` | You want to set a response header. |
| `res.file(...)` | You want to send a file. |

## Complete example

Your full `src/main.cpp` can now look like this:

```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.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;
}
```

Run:

```
vix dev
```

Test all 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"
```

## Organize routes with functions

When your app grows, avoid putting everything directly in `main()`.

You can organize routes like this:

```cpp
#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;
}
```

This keeps `main()` small and makes the app easier to maintain.

### Common mistakes

### Forgetting to run the server

Routes do nothing until you call:

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

### Forgetting to return after an error

When you send an error response, return immediately.

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

### Port 8080 is already in use

If the server cannot start, another process may already be using port `8080`.

Check:

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

Stop the process or change the port:

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

### Running from the wrong folder

Run project commands from inside your project:

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

### What you should remember

The basic Vix HTTP model is:

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

The minimal server is:

```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;
}
```

Use:

```
vix dev
```

during development.

Use:

```
vix run
```

when you want to start the app directly.

## What to read next

Now that you have a running HTTP server, continue with:

- The Vix Book
- Routes
- Request and Response
- Build a REST API
- Templates

# Part 2

## The Vix Book

*The Vix Book now goes deeper into the runtime model and application layers. Read it in order — each chapter builds on the previous one — to grow from a single C++ file into a complete production system.*

# 1. Introduction

Welcome to the Vix book.

This book teaches Vix step by step, like a story. You will start with the simplest idea:

```
Run C++ code quickly.
```

Then you will grow toward real backend systems:

- HTTP APIs,
- JSON,
- middleware,
- validation,
- database,
- WebSocket,
- async runtime,
- cache,
- offline-first sync,
- P2P,
- and production deployment.

The goal is not only to show commands or APIs. The goal is to help you understand the mental model behind Vix.

## What is Vix?

Vix is a modern C++ runtime for building fast and reliable applications. It gives C++ a more direct development experience:

```
vix run main.cpp
```

or:

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

Instead of forcing you to manually configure everything before writing your first line of useful code, Vix gives you a runtime-oriented workflow. You write the app. Vix handles the development loop.

## The big idea

C++ is powerful. But building real applications in C++ often requires too much setup before the developer can even begin. You may need to think about CMake, compiler flags, linker flags, dependencies, build directories, runtime arguments, logs, tests, server startup, database flags, and project structure.

Vix tries to make this smoother. The idea is simple:

```
C++ should be able to feel direct without losing its power.
```

Vix does not remove C++. Vix gives C++ a better application runtime around it.

## Why a runtime?

A language alone is not enough to build modern applications comfortably.

JavaScript became widely used for backend development not only because of the language, but because Node.js gave developers a runtime, package workflow, and fast feedback loop.

Python became popular for scripting and backend work because running a file is simple:

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

Vix brings that kind of workflow to C++:

```
vix run main.cpp
```

But with the performance, control, and explicitness of C++.

## What Vix is not

Vix is not a replacement for C++. It is not a garbage-collected language. It is not trying to hide how systems work or turn C++ into JavaScript or Python.

Vix is a runtime and toolkit that makes C++ application development more practical. It keeps the C++ philosophy — explicit, fast, deterministic, close to the system — but adds a smoother workflow.

## A tiny example

```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;
}
```

Run it:

```
vix run main.cpp
```

Then open `http://localhost:8080`.

This small example already shows the core Vix style:

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

## The development experience

```
vix new api    # Create a project
vix dev        # Run in development mode
vix build      # Build it
vix check      # Validate it
vix tests      # Run tests
vix fmt        # Format code
```

## The structure of this book

**Part 1: Understand Vix** Introduction, Why Vix, Mental model

**Part 2: Start building** Installation, Run your first file, Create your first project, First HTTP server

**Part 3: Build APIs** Routes, Request and Response, JSON API

**Part 4: Add professional layers** Middleware, Validation, Errors and logging

**Part 5: Connect real systems** Database, Real-time WebSocket, Async runtime

**Part 6: Advanced runtime features** Cache, Offline-first sync, P2P

**Part 7: Production** Production deployment, Next steps

## How to read this book

Read it in order the first time. Each chapter builds on the previous one.

You only need basic C++ knowledge:

- functions,
- classes,
- headers,
- `std::string`,
- `std::vector`,
- lambdas,
- and basic CMake awareness.

## The main mental shift

Traditional C++ development often starts with the build system. Vix starts with the application.

Instead of thinking first about how to configure, link, and build, Vix wants the first question to be:

```
What do I want to build?
```

## Vix and production

A Vix app can run as a normal Linux service:

```
browser → Nginx → Vix app → systemd
```

The production chapter will show how to deploy with a release build, systemd, Nginx, TLS, logs, and health checks.

### What you should remember

Vix is a modern C++ runtime for building fast and reliable applications. It gives C++ a direct run workflow, a project workflow, an HTTP application model, JSON APIs, middleware, validation, database access, WebSocket support, runtime-oriented tools, and a production deployment path.

The core idea: keep the power of C++, make the application workflow simpler.

## Next chapter

**Next: Why Vix**

# 2. Why Vix?

Vix exists because building real applications in C++ should be faster, clearer, and more practical.

C++ gives you: performance, control, deterministic lifetimes, low-level access, strong types, native binaries, and systems-level power. But for many developers, the experience of building applications in C++ feels heavier than it should.

## The problem

To build a simple backend, you may need to think about compiler setup, CMake configuration, dependency installation, include paths, linker flags, build directories, runtime arguments, server lifecycle, logging, JSON, HTTP routing, database configuration, tests, and deployment.

Each piece is manageable alone. The problem is that you must wire everything together before the application becomes useful.

## The first step should be simple

In many ecosystems, the first step is direct:

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

Vix gives C++ a similar starting point:

```
vix run main.cpp
```

The goal is not to copy these ecosystems. The goal is to make C++ feel more direct for application development.

## C++ does not lack power

The problem is not the language itself. The problem is the missing application workflow around the language. C++ already has great compilers, excellent performance, RAII, templates, zero-cost abstractions, native concurrency, mature tooling, and portable binaries. What developers often miss is a unified runtime experience.

## The missing layer

When you build a backend in C++, you usually combine many pieces:

- HTTP library,
- JSON library,

- build system,
- logging library,
- database layer,
- middleware logic,
- validation layer,
- WebSocket library,
- and deployment scripts.

This creates friction. Vix answers these questions with one coherent workflow.

## The Vix approach

```
vix run main.cpp          # Run a file
vix new api && vix dev    # Create and run a project
vix build                 # Build
vix check && vix tests    # Validate and test
```

## Why not just use CMake?

Vix does not replace CMake. CMake answers: "How do I configure and build this C++ project?"

Vix answers: "How do I build, run, test, develop, package, and operate this C++ application?"

Vix can use CMake underneath while giving the developer a simpler top-level experience.

## Why not just use a C++ web framework?

A web framework usually focuses on HTTP. Vix is broader.

Vix also includes the surrounding application workflow:

- CLI,
- project creation,
- direct file execution,
- build commands,
- dependency workflow,
- tests,
- formatting,
- logging,
- database access,
- WebSocket runtime,
- middleware,

- validation,
- cache,
- sync,
- P2P,
- and deployment path.

## The application comes first

Traditional C++ project setup often begins with build configuration. Vix wants the first step to be: write the application, run it.

```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
```

## Vix is explicit

A route is explicit:

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

A database query is explicit:

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

This matters because serious systems must be understandable. Vix should reduce friction without hiding behavior.

## Why reliability matters

Vix is also connected to a bigger idea: applications should keep working under real-world conditions.

Real systems face network failure, process restart, slow servers, partial writes, timeouts, offline clients, retries, duplicate requests, and lost connections.

This is why Vix includes deeper runtime ideas like cache, offline-first sync, WAL, outbox, retry, and P2P.

### What you should remember

Vix exists because C++ deserves a smoother application runtime. The core problem is not that C++ is weak. The problem is that building applications in C++ often starts with too much friction.

Vix provides:

- direct execution,
- project workflow,
- HTTP app model,
- JSON support,
- middleware,
- validation,
- database access,
- WebSocket runtime,
- production deployment path,
- and advanced reliability modules.

The core idea: keep C++ powerful, make building applications with it feel direct.

## Next chapter

**Next: Mental model**

# 3. Mental Model

This chapter explains how to think about Vix.

Before learning every command and module, you need one clear picture:

```
Vix is a runtime workflow for C++ applications.
```

It connects four layers:

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

The CLI helps you work. The runtime runs your code. The application layer exposes APIs such as `App`, `Request`, and `Response`. The modules add capabilities like JSON, database, middleware, validation, WebSocket, cache, sync, and P2P.

## The simplest mental model

```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
```

This example contains most of the Vix mental model:

- `vix run` → CLI workflow,
- `App` → application object,
- `app.get` → route registration,

- `Request` → incoming request,
- `Response` → outgoing response,
- `app.run` → runtime starts the server.

## Layer 1: The CLI

The CLI is the developer entrypoint. It provides commands such as: `vix run`, `vix new`, `vix dev`, `vix build`, `vix check`, `vix tests`, `vix fmt`, `vix doctor`.

It gives a consistent development loop: create → run → edit → reload → check → test → build → deploy.

| Command | Purpose |
|---|---|
| `vix run` | Builds and runs a file, project, or manifest. |
| `vix dev` | Starts a development loop with watch reload. |
| `vix build` | Configures, compiles, and links the project. |
| `vix check` | Validates builds, tests, and sanitizers. |

## Layer 2: The runtime

The runtime is what makes the app actually run. In a simple HTTP program:

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

For advanced apps with HTTP + WebSocket together:

```
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);
```

## Layer 3: The application

```
App app;

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

**Request** is read-only input from the client:

- path params,
- query params,
- headers,
- body,
- JSON body.

**Response** is how the route sends output back:

- text,
- JSON,
- files,
- status codes,
- headers.

### Keep `main()` small

```
int main()
{
  App app;

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

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

## Layer 4: Modules

```
#include <vix.hpp>           // core
#include <vix/db.hpp>        // database
#include <vix/websocket.hpp> // WebSocket
#include <vix/middleware.hpp>// middleware
#include <vix/validation.hpp>// validation
```

Use the smallest public module header that gives the feature you need.

### Key modules

**Middleware** runs around routes — CORS, rate limiting, authentication, security headers, static files.

**Validation** checks input before business logic:

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

**Database** gives explicit access:

```
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** adds real-time events:

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

## Configuration

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

Environment variables:

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

## Request lifecycle

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

With database: request → middleware → route → validate → query database → return JSON

## Error flow

```
validation failure   → 400 Bad Request
missing token        → 401 Unauthorized
not allowed          → 403 Forbidden
missing resource     → 404 Not Found
too many requests    → 429 Too Many Requests
```

## Application growth

```
Start:  src/main.cpp

Later:  src/
        ├── main.cpp
        ├── routes/
        ├── validation/
        ├── database/
        └── services/
```

Start small. Move code into modules when it earns it. Keep `main()` as wiring.

## Production shape

```
browser → Nginx → Vix app on localhost → systemd
```

### What you should remember

The Vix mental model has four layers: **CLI**, **Runtime**, **Application**, and **Modules**.

The **CLI** controls the developer workflow. The **Runtime** executes the application. The **Application** layer is built around `App`, `Request`, and `Response`. The **Modules** add capabilities such as JSON, middleware, validation, database, WebSocket, cache, sync, and P2P.

The best way to grow a Vix app is simple:

```
start with one file → keep main() small → register routes through functions →
add modules when needed → move logic into services
```

**Next: Routes**

# 4. Routes

Routes are the heart of a Vix HTTP application. They connect an HTTP request to C++ code:

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

## Route anatomy

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

| Part | Purpose |
|---|---|
| `app.get` | HTTP method |
| `"/"` | Path pattern |
| Lambda | Handler |
| `Request &req` | Incoming request |
| `Response &res` | Outgoing response |

## HTTP methods

```
app.get("/users", list_handler);          // read
app.post("/users", create_handler);       // create
app.put("/users/{id}", replace_handler);  // replace
app.patch("/users/{id}", update_handler); // partial update
app.del("/users/{id}", delete_handler);   // delete
```

## Path parameters

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

Matches: `/users/1`, `/users/42`, `/users/abc`

### Multiple path parameters

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

## Query parameters

Query params are NOT part of the route pattern — they come after `?`:

```
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"
```

## Path params vs query params

```
Path params  → identify the resource:  /users/42
Query params → modify how it's read:   /users?page=2&limit=10
```

## Error routes

```
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;    // always return after error
  }

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

## Organizing routes by feature

```
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;
}
```

## Route order matters

Routes are matched in registration order.

Specific routes should be registered before generic fallback routes.

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

// Wrong — wildcard catches everything
app.get("/*", fallback_handler);
app.get("/users/{id}", user_handler);

// In the wrong version, the wildcard route can catch requests before /users/{id}
gets a chance to run.
```

## Wildcard routes

A wildcard route matches many paths and is useful as a fallback.

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

Used for:

- static file fallback
- SPA fallback
- custom 404 behavior

Used for: SPA fallback, static file fallback, custom 404.

## Static file fallback

```
#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);
}
```

With this structure:

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

A request to:

```
/index.html
```

serves:

```
public/index.html
```

Run:

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

## SPA fallback

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

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

## Split routes into files

**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"});
  });
}
```

Update `CMakeLists.txt`:

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

## Complete example

```cpp
#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;
}
```

### Common mistakes

#### Missing slash

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

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

#### Confusing path params and query params

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

#### Forgetting to return after error

```
// Always return after sending an error
if (bad) {
  res.status(400).json(...);
  return;
}
```

#### Wildcard route too early

Specific routes must be registered before wildcard routes.

### What you should remember

A route connects: HTTP method + path → C++ handler. Use path params for resource identity, query params for options, grouped functions as the app grows. The core idea: routes are the public shape of your application — keep them clear, predictable, and organized.

## Next chapter

**Next: Request and Response**

# 5. Request and Response

They are the core of the Vix HTTP model:

```
Request reads what the client sent.
Response sends what your app returns.
```

## What is Request?

| Method | Purpose |
|---|---|
| `req.param("id")` | Reads a route path parameter. |
| `req.param("id", "0")` | Reads a route parameter with fallback. |
| `req.query_value("page", "1")` | Reads a query parameter with fallback. |
| `req.query()` | Reads all query parameters. |
| `req.header("Authorization")` | Reads a request header value. |
| `req.body()` | Reads the raw request body. |
| `req.json()` | Reads the parsed JSON body. |
| `req.path()` | Reads the current request path. |

## What is Response?

| Method | Purpose |
|---|---|
| `res.text("Hello")` | Sends a plain text response. |
| `res.json({"ok", true})` | Sends a JSON response. |
| `res.status(201).json(...)` | Sets status before sending. |
| `res.header("X-Foo", "bar")` | Sets a response header value. |
| `res.file("public/index.html")` | Sends a static file response. |

## Reading path parameters

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

## Reading query parameters

```
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"
```

## Reading all query parameters

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

## Reading headers

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

## Reading the raw body

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

## Reading JSON body

```
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
  });

});
```

## Setting headers

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

## Cache-Control header

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

## Download response

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

## Error helper

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

## Good response shape

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

// Error
{ "ok": false, "error": "message" }

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

## Request and Response lifecycle

```
client sends request
  ↓
Vix creates Request
  ↓
route handler reads Request
  ↓
route handler writes Response
  ↓
Vix sends response to client
```

### Common mistakes

#### Forgetting to name Request when you need it

```
// Wrong — req is unnamed but used
app.get("/users/{id}", [](Request &, Response &res){
  const std::string id = req.param("id");
});   // error!

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

#### Forgetting to return after errors

```
// Wrong
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});
```

#### Trusting JSON body without checking shape

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

### What you should remember

**Request** is the input: params, query, headers, body, json. **Response** is the output: status, headers, text, json, file.

The core route flow: read Request → validate input → write Response → return.

## Next chapter

**Next: JSON API**

# 6. JSON API

Now you will build a complete JSON API. JSON APIs are one of the most common things you will build with Vix.

```
client sends JSON → Vix reads Request → route validates input → route returns JSON Response
```

## Routes to build

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

## Recommended response shape

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

## User struct

```
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"} };
}
```

> *Data is in-memory for now. The database chapter will replace this with SQLite or MySQL.*

## JSON helpers

```
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::int6
4_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;
  }
}
```

## 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
```

## 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
```

## 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"}'
```

## Complete example

```cpp
#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::int6
4_t id)
{
  for (const auto &u : users)
    if (u.id == id)
      return u;

  return std::nullopt;
}
```

```cpp
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;
}
```

## Test the complete API

```
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 '{}'
```

## Status codes for JSON APIs

| Status | Meaning |
|---|---|
| `200` | OK, request succeeded. |
| `201` | Created, resource added. |
| `400` | Bad Request, invalid input. |
| `401` | Unauthorized, auth required. |
| `403` | Forbidden, access denied. |
| `404` | Not Found, resource missing. |
| `409` | Conflict, state mismatch. |
| `429` | Too Many Requests, rate limited. |
| `500` | Internal Server Error. |

## Route flow for JSON APIs

```
read request → parse params or body → validate input → run logic → format JSON →
send response
```

## Preparing for the next chapters

- **Database:** the in-memory vector will be replaced by SQLite or MySQL.
- **Middleware:** CORS, rate limiting, and authentication will wrap the routes.

- **Validation:** manual checks will become declarative with `vix::validation`.

### Common mistakes

#### Forgetting `Content-Type` with curl

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

#### Trusting body shape

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

#### Forgetting to return after error

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

#### Returning inconsistent errors

Use one helper: `respond_error(res, 400, "message")`.

### What you should remember

A JSON API route follows: Request → validate → logic → JSON Response. Use `res.json(...)` for responses, `req.json()` for JSON bodies, helpers for consistent errors and JSON formatting. The core idea: JSON APIs become simple when request parsing, validation, logic, and response formatting stay separate.

## Next chapter

**Next: Middleware**

# 7. Middleware

In the previous chapter, you built a JSON API. Now you will learn middleware.

Middleware is code that runs around your routes. It can inspect the request before the route handler runs, and modify the response before it is sent.

```
request → middleware → route handler → response
```

## Why middleware exists

Without middleware, shared logic like CORS, rate limiting, authentication, and security headers must be repeated in every route. Middleware lets you write common behavior once.

## Public headers

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

## Middleware order

Order matters. Configure middleware before registering routes:

```
int main()
{
  App app;

  configure_middlewares(app);
  register_routes(app);

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

A common order:

```
CORS → rate limit → security headers → body limit → authentication → routes
```

## CORS middleware

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

CORS is not authentication. It is a browser rule that answers: which browser origins are allowed to call this API?

For production, allow only your real frontend domain. Do not use open CORS unless the API is intentionally public.

## Rate limiting middleware

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

Use rate limiting to protect public APIs, login endpoints, and small VPS deployments. When the limit is exceeded, the API returns `429 Too Many Requests`.

### Test 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
```

## Stricter middleware for auth routes

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

## Static files middleware

```
#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,
      }
  )
));
```

## Manual auth check in a 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"
  });

});
```

## Complete example

```cpp
#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"); return; }
             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;
}
```

## 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"}'

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

# Middleware and route responsibility

| Middleware | Example route action |
| --- | --- |
| CORS | Allows browser API calls. |
| Rate limiting | Protects public endpoints. |
| Authentication | Guards dashboard routes. |
| Request IDs | Tracks each request in logs. |
| Logging | Records request activity. |
| Body limits | Rejects oversized requests. |

## Common mistakes

### Registering middleware after routes

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

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

### Making CORS too open in production

Development CORS can allow localhost. Production should allow your real frontend only.

### Using one rate limit for everything

Login needs a stricter limit than normal API routes.

### Returning 403 instead of 429 for rate limits

Rate limit failures should return `429 Too Many Requests`.

### What you should remember

Middleware wraps route handlers. Use it for shared behavior: CORS, rate limiting, authentication, logging, security, body limits, static files.

```
int main()
{
  App app;

  configure_middlewares(app);
  register_routes(app);

  app.run(8080);
}
```

The core idea: middleware keeps route handlers clean by moving shared request behavior into one reusable layer.

## Next chapter

**Next: Validation**

# 8. Validation

In the previous chapter, you learned middleware. Now you will learn validation.

Validation checks whether incoming data is correct before your application uses it.

```
request data → validation → business logic → response
```

## Why validation exists

A route that trusts input blindly can receive missing fields, invalid emails, weak passwords, wrong types, or unsafe data. Validation prevents bad input from reaching real application logic.

## Public header

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

## Validate one string

```
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";
  }
}
```

## Common rules

| Rule | Purpose |
|---|---|
| `required()` | Requires a present and non-empty value. |
| `email()` | Requires a valid email address format. |
| `length_min(n)` | Requires a string length of at least `n`. |
| `length_max(n)` | Requires a string length of at most `n`. |
| `min(n)` | Requires a numeric value of at least `n`. |
| `max(n)` | Requires a numeric value of at most `n`. |
| `between(a, b)` | Requires a value between `a` and `b`. |
| `in_set({...})` | Requires one of the allowed values. |

## Validate numbers

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

## Validate allowed values

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

## Parsed validation (string → number)

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

Useful for query params, route params, and form fields that arrive as strings.

## Schema validation

```
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(1
20))

        .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);
```

## 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(1
20))

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

RegisterForm form;
auto result = form.validate();            // call on object
auto result2 = RegisterForm::validate(form);  // static call
```

## Cross-field validation

```
.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");
    }
});
```

## Validation in a 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"
  });

});
```

## Structured error shape

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

### Common mistakes

#### Validating after business logic

```
// Wrong: create user then validate
// Correct: validate then create user
```

#### Forgetting body shape check

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

#### Forgetting to return after validation failure

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

#### Returning only one validation error

For forms, return all field errors at once so users can fix everything together.

### What you should remember

The normal flow: Request → validation → business logic → Response. Use single-value validation for simple fields, schemas for structs, structured errors for APIs. The core idea: bad input should stop at the boundary of your application.

## Next chapter

**Next: Errors and logging**

# 9. Errors and Logging

In the previous chapter, you learned validation. Now you will learn errors and logging.

```
request → validation → business logic → error or success → structured response →
structured logs
```

A production application should return clear responses to clients and keep useful logs for developers.

## Why error handling matters

If every route returns a different error shape, the API becomes hard to use. A Vix API should use one predictable shape.

## Recommended error shape

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

## HTTP status codes

| Status | Meaning |
|---|---|
| `200` | OK, request succeeded. |
| `201` | Created, resource added. |
| `400` | Bad Request, invalid input. |
| `401` | Unauthorized, auth required. |
| `403` | Forbidden, access denied. |
| `404` | Not Found, resource missing. |
| `409` | Conflict, state mismatch. |
| `429` | Too Many Requests, rate limited. |
| `500` | Internal Server Error. |

Do not return 200 for errors.

## Basic error helper

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

Always return after sending an error:

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

## Error codes

Use stable codes for production APIs:

```
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
```

## Do not leak internal errors

```
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"
  });
}
```

## Public logging header

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

## Basic logs

```
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");
```

## Log levels

| Level | Use |
|---|---|
| `trace` | Records very detailed debugging events. |
| `debug` | Records useful debugging information. |
| `info` | Records normal application events. |
| `warn` | Records unusual but non-fatal events. |
| `error` | Records failed operations. |
| `critical` | Records serious system failures. |

Recommended: `info` in production, `debug` or `trace` during development.

## Structured logs

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

## Log formats

```
vix::log::set_format(vix::log::LogFormat::KV);    // local development
vix::log::set_format(vix::log::LogFormat::JSON);   // production log collectors
```

## Log context

```
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();
```

## Set log level

```
vix::log::set_level(vix::log::LogLevel::Info);  // in code
// vix run main.cpp --log-level debug             // via CLI
// VIX_LOG_LEVEL=info                             // via environment
```

## What to log

Good: app started, user registered, login failed, database connection failed, unexpected exception.

Never log: passwords, tokens, private keys, sensitive personal data.

## Complete example

```cpp
#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(1
20))

        .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 &message)
{
  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.ema
il);

      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;
}
```

## 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"}'
```

### Common mistakes

#### Returning HTTP 200 for errors

Use the correct error status code always.

#### Logging secrets

Never log passwords, tokens, private keys, or authorization headers.

#### Exposing internal exceptions

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

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

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

#### Forgetting to return after an error

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

### Production config

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

### What you should remember

Errors are for clients. Logs are for developers and operators. API errors should be consistent: `{ "ok": false, "error": "stable_code", "message": "Safe message" }` Logs should keep useful internal context with structured fields. The core idea: a reliable app does not only work when everything succeeds — it also explains what happened when something fails.

## Next chapter

**Next: Database**

# 10. Database

In the previous chapter, you learned errors and logging. Now you will connect a Vix application to a database.

```
Request → validation → database query → JSON Response
```

Memory disappears when the app restarts. A database gives your application durable state.

## Public header

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

## SQLite or MySQL?

| Criteria | SQLite | MySQL |
|---|---|---|
| Best for | Local development, small apps, MVPs. | Multi-user production APIs. |
| Setup | Very simple, no server required. | Requires a database server. |
| Persistence | Stores data in a local file. | Stores data in a server database. |

Start with SQLite for learning.

## Build flags

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

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

## First SQLite connection

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

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

## First MySQL connection

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

## Database from .env

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

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

## Create a 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)");
```

## Insert data

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

Always use parameterized queries. Never build SQL with string concatenation.

## Query data

```
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";
}
```

## Prepared statements

```
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();
```

Use prepared statements for: user input, route parameters, query filters, inserts, updates, deletes.

## Connection pool

```
vix::db::PooledConn conn(db.pool());
// connection returns automatically when PooledConn is destroyed (RAII)
```

## 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();
});
```

Use transactions for: orders + items, user + profile, money transfers, any multi-step write.

# Complete database API

```cpp
#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 NOT NULL, role TEXT NOT NULL)");
}

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::s
tring &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(re
s, 500, "internal_error", "Internal server error");
    }
  });

  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(r
es, 500, "internal_error", "Internal server error");
    }
  });
}

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;
  }
}
```

## 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"}'
```

Restart the app — unlike the memory API, the new user should still exist.

## Migrations

For real projects, use migrations instead of `CREATE TABLE IF NOT EXISTS` at startup.

```
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 TEXT NOT NULL, role TEXT NOT NULL)")->exec();
  }

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

### Common mistakes

#### Building SQL with string concatenation

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

// Correct — use prepared statements
auto stmt = conn->prepare("SELECT * FROM users WHERE name = ?");
stmt->bind(1, name);
```

#### Returning raw database errors

Log internal details, return safe client errors.

#### Not validating before insert

Always validate input before database writes.

### What you should remember

The Vix DB model is explicit: connect → prepare → bind → query → read rows → commit when needed.

Use prepared statements for user input. Use transactions for multi-step writes. Routes should validate input, call database logic, and return safe JSON responses.

## Next chapter

**Next: Real-time WebSocket**

# 11. Real-time WebSocket

In the previous chapter, you connected a Vix app to a database. Now you will learn WebSocket.

HTTP is request/response — the connection ends. WebSocket stays open for real-time communication.

```
client connects → connection stays open → client sends events → server sends events
→ many clients receive updates
```

## When to use WebSocket

Use HTTP for: CRUD APIs, page loading, authentication, normal JSON APIs.

Use WebSocket for: live messages, dashboards, presence, streaming events, notifications.

## Public headers

```
#include <vix.hpp>
#include <vix/websocket.hpp>
#include <vix/websocket/AttachedRuntime.hpp>  // for HTTP + WebSocket together
```

## Vix typed message model

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

## Minimal WebSocket server

```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;
}
```

## HTTP + WebSocket together

```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};
};
```

## Register HTTP routes

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

## Register WebSocket protocol

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

## Run together

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

## Compact alternative: `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 &p
ayload) {
    if (type == "chat.message") ws.broadcast_json("chat.message", payload);
  });

});
```

## Recommended event protocol

| Event | Direction | Purpose |
|---|---|---|
| `chat.join` | Client -> server | Joins a chat room or session. |
| `chat.leave` | Client -> server | Leaves a chat room or session. |
| `chat.message` | Both directions | Sends or receives chat content. |
| `chat.error` | Server -> client | Reports a chat-related error. |
| `app.ping` | Client -> server | Sends a health check request. |
| `app.pong` | Server -> client | Returns a health check response. |
| `system.connected` | Server -> client | Confirms the client connected. |
| `system.disconnected` | Server -> client | Confirms the client disconnected. |

## Validate WebSocket payloads

```
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);
}
```

## WebSocket and Nginx

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

### Common mistakes

#### Using WebSocket for normal CRUD

Use HTTP for create/read/update/delete. Use WebSocket for live events only.

#### Broadcasting unvalidated payloads

Validate payloads before broadcasting.

#### Forgetting Nginx upgrade headers

Without upgrade headers, WebSocket fails through Nginx.

#### Not handling unknown event types

Always have a fallback for unknown types returning `app.unknown`.

### What you should remember

HTTP is request/response. WebSocket is a long-lived real-time connection.

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

The core idea: use HTTP for normal API requests and WebSocket for live events that must reach clients immediately.

## Next chapter

**Next: Async runtime**

# 12. Async Runtime

In the previous chapter, you learned WebSocket. Now you will learn the async runtime.

```
event loop → coroutines → timers, signals, networking, CPU pool → completion resumes
on event loop
```

Vix async is not a web framework. It is a low-level C++20 asynchronous runtime core — the deterministic foundation for servers, networking, timers, workers, and real-time systems.

## Why async exists

C++ has coroutines as a language feature, but not a full runtime. You still need: event loop, scheduler, timers, I/O integration, CPU worker pool, signal handling, cancellation.

Vix async provides that foundation.

## The core idea

User coroutine code resumes on the event loop. Blocking or CPU-heavy work is moved elsewhere. Completion returns to the event loop.

```
event loop thread → runs user coroutine → starts timers/I/O/CPU jobs → completion →
coroutine resumes
```

## Main components

| Component | Purpose |
|---|---|
| `io_context` | Owns the event loop and runtime services. |
| `scheduler` | Schedules coroutine resumption. |
| `task<T>` | Represents a coroutine return type. |
| `timer` | Provides non-blocking timers. |
| `thread_pool` | Offloads CPU-bound work from the event loop. |
| `signal_set` | Handles operating system signals. |
| `net::*` | Provides asynchronous networking primitives. |
| `when_all` | Waits for multiple tasks to complete. |
| `when_any` | Waits for the first completed task. |

## Minimal async example

```cpp
#include <vix/print>
#include <vix/async.hpp>

using vix::async::core::io_context;
using vix::async::core::task;

static task<void> app(io_context &ctx)
{
  vix::print("[async] hello from task");

  co_await ctx.timers().sleep_for(std::chrono::milliseconds(100));

  vix::print("[async] after timer");

  ctx.stop();
  co_return;
}

int main()
{
  io_context ctx;
  auto t = app(ctx);
  ctx.post(t.handle());  // schedule task
  ctx.run();              // start event loop
  vix::print("[async] done");
  return 0;
}
```

## Timers

Use `co_await` timers instead of blocking sleep:

```cpp
// Wrong — blocks event loop
std::this_thread::sleep_for(std::chrono::milliseconds(100));

// Correct — yields to event loop
co_await ctx.timers().sleep_for(std::chrono::milliseconds(100));

// Callback style
ctx.timers().after(std::chrono::milliseconds(150), []() { /* callback */ });
```

## CPU-bound work

CPU-heavy work belongs in the CPU pool:

```cpp
// Wrong — blocks event loop
for (int i = 0; i < 100000000; ++i) { /* heavy work */ }

// Correct — offload to worker thread
int result = co_await ctx.cpu_pool().submit([]() { return heavy_work(); });
```

## Signals

```cpp
#include <vix/async/core/signal.hpp>

auto &signals = ctx.signals();
signals.add(SIGINT);
signals.add(SIGTERM);

signals.on_signal([&](int signal) { ctx.stop(); });

int signal = co_await signals.async_wait();
ctx.stop();
```

## TCP echo server

```cpp
static task<void> handle_client(std::unique_ptr<vix::async::net::tcp_stream> client)
{
  std::vector<std::byte> buffer(4096);
  while (client && client->is_open())
  {
    std::size_t n = co_await client->async_read(std::span<std::byte>(buffer.data(),
buffer.size()));
    if (n == 0) break;
    co_await client->async_write(std::span<const std::byte>(buffer.data(), n));
  }
  client->close();
}

static task<void> server(io_context &ctx)
{
  auto listener = vix::async::net::make_tcp_listener(ctx);
  co_await listener->async_listen({"0.0.0.0", 9090}, 128);

  while (ctx.is_running())
  {
    auto client = co_await listener->async_accept();
    vix::async::core::spawn_detached(ctx, handle_client(std::move(client)));
  }
}
```

### `when_all` and `when_any`

```
// Wait for both to complete
auto values = co_await when_all(sched, a(), b());
std::get<0>(values);  // result from a()
std::get<1>(values);  // result from b()

// Wait for first to complete
auto [index, vals] = co_await when_any(sched, a(), b());
```

## Async and HTTP

For normal HTTP apps, you do not need low-level async:

```
App app;
app.get("/", handler);
app.run(8080);
```

The runtime handles execution. Understanding async helps you understand what happens underneath.

#### Common mistakes

#### Blocking the event loop

```
// Wrong
std::this_thread::sleep_for(std::chrono::seconds(1));

// Correct
co_await ctx.timers().sleep_for(std::chrono::seconds(1));
```

#### Running heavy CPU work on the event loop

```
// Wrong
auto result = heavy_work();

// Correct
auto result = co_await ctx.cpu_pool().submit([]() { return heavy_work(); });
```

#### Forgetting to schedule the task

```
// A task starts suspended — you must schedule it
ctx.post(t.handle());
```

#### Ignoring signal handling

Production services must handle `SIGINT` and `SIGTERM` for clean shutdown with systemd.

### What you should remember

The most important flow:

```
io_context ctx;
auto t = app(ctx);
ctx.post(t.handle());  // schedule
ctx.run();              // run event loop
// co_await operations inside the coroutine
```

Use timers instead of blocking sleep. Use CPU pool for heavy work. Use signals for clean shutdown.

The core idea: async code stays understandable when execution is explicit, non-blocking, and resumed through one clear runtime.

## Next chapter

**Next: Cache**

# 13. Cache

In the previous chapter, you learned the async runtime. Now you will learn cache.

```
request → cache lookup → cache hit → return cached result
                       → cache miss → compute or fetch → store → return response
```

In Vix, cache is not only about speed — it is also part of offline-first behavior. When the network fails, a cache can safely serve previously stored data.

## Public headers

```
#include <vix/cache/Cache.hpp>
#include <vix/cache/CacheContext.hpp>
#include <vix/cache/CacheEntry.hpp>
#include <vix/cache/CachePolicy.hpp>
#include <vix/cache/MemoryStore.hpp>
#include <vix/cache/FileStore.hpp>
#include <vix/cache/LruMemoryStore.hpp>
#include <vix/cache/CacheKey.hpp>
```

## Core concepts

| Concept | Purpose |
|---|---|
| `Cache` | Decides whether a cached entry is usable. |
| `CacheEntry` | Stores one cached response or value. |
| `CachePolicy` | Defines TTL, freshness, and stale behavior. |
| `CacheContext` | Describes the current network condition. |
| `MemoryStore` | Stores cached entries in memory. |
| `FileStore` | Stores cached entries on disk. |
| `LruMemoryStore` | Stores bounded entries with LRU eviction. |
| `CacheKey` | Builds stable keys for cached entries. |

## Time helper

```
static std::int64_t now_ms()
{
  using namespace std::chrono;
  return duration_cast<milliseconds>(steady_clock::now().time_since_epoch()).count()
;
}
```

## CacheEntry

```
vix::cache::CacheEntry entry;
entry.status = 200;
entry.body = R"({"users":[1,2,3]})";
entry.headers["Content-Type"] = "application/json";
entry.created_at_ms = t0;
```

## CachePolicy

```
vix::cache::CachePolicy policy;
policy.ttl_ms = 5'000;                    // fresh for 5 seconds
policy.allow_stale_if_offline = true;
policy.stale_if_offline_ms = 10'000;      // allow stale up to 10s when offline
policy.allow_stale_if_error = true;
policy.stale_if_error_ms = 5'000;         // allow stale up to 5s on network error
```

## CacheContext

```
CacheContext::Online()        // normal operation
CacheContext::Offline()       // no network
CacheContext::NetworkError()  // network request failed
```

## Minimal memory cache

```
auto store = std::make_shared<vix::cache::MemoryStore>();
vix::cache::CachePolicy policy;
policy.ttl_ms = 5'000;
vix::cache::Cache cache(policy, store);

const std::string key = "GET /api/users?page=1";
const auto t0 = now_ms();

vix::cache::CacheEntry entry;
entry.status = 200;
entry.body = R"({"users":[1,2,3]})";
entry.created_at_ms = t0;
cache.put(key, entry);

auto cached = cache.get(key, t0 + 100, vix::cache::CacheContext::Online());
if (cached) std::cout << "cache hit: " << cached->body << "\n";
```

## FileStore (persistence)

```
auto store = std::make_shared<vix::cache::FileStore>(vix::cache::FileStore::Config{
    .file_path = "./cache.json",
    .pretty_json = true});
```

Entries persist to disk and survive process restart.

## LruMemoryStore (bounded memory)

```
auto store = std::make_shared<vix::cache::LruMemoryStore>(
    vix::cache::LruMemoryStore::Config{.max_entries = 2048});
```

When capacity is reached, the least recently used entry is evicted.

## Stale data

```
// Online — fresh data only (within TTL)
cache.get(key, t0 + 50, CacheContext::Online());     // hit if within 5000ms

// Offline — allow stale up to stale_if_offline_ms
cache.get(key, t0 + 3000, CacheContext::Offline());  // hit if within 10000ms

// Network error — allow stale up to stale_if_error_ms
cache.get(key, t0 + 4000, CacheContext::NetworkError());  // hit if within 5000ms

// Too old — miss regardless of context
cache.get(key, t0 + 20'000, CacheContext::Offline());  // miss
```

## CacheKey builder

```
#include <vix/cache/CacheKey.hpp>

std::unordered_map<std::string, std::string> req_headers;
req_headers["Accept"] = "application/json";

const std::string key = vix::cache::CacheKey::fromRequest(
    "get",
    "/api/users",
    "b=2&a=1",      // query is normalized
    req_headers,
    {"Accept"});    // vary on Accept header
```

Good cache keys include: method, path, normalized query params, and selected vary headers.

## Cache in an HTTP route

```
app.get("/api/products", [&cache](Request &, Response &res){
  const auto now = now_ms();
  const std::string key = "GET /api/products";

  auto cached = cache.get(key, now, vix::cache::CacheContext::Online());
  if (cached)
  {
    res.header("X-Vix-Cache", "HIT");
    res.status(cached->status).send(cached->body);
    return;
  }

  const std::string body = R"({"ok":true,"data":[]})";

  vix::cache::CacheEntry entry;
  entry.status = 200;
  entry.body = body;
  entry.headers["Content-Type"] = "application/json";
  entry.created_at_ms = now;
  cache.put(key, entry);

  res.header("X-Vix-Cache", "MISS");
  res.header("Content-Type", "application/json");
  res.send(body);
});
```

## Policy examples

```
// Short API cache
policy.ttl_ms = 5'000;

// Offline-friendly
policy.ttl_ms = 30'000;
policy.allow_stale_if_offline = true;
policy.stale_if_offline_ms = 10 * 60 * 1000;

// Network-error fallback
policy.ttl_ms = 10'000;
policy.allow_stale_if_error = true;
policy.stale_if_error_ms = 60'000;
```

## Good vs bad cache candidates

### Good use cases

- Public `GET` responses
- Product lists

- Configuration
- Feature flags
- Read-heavy dashboards

### Be careful with

- Private user data
- Payment state
- Security decisions
- Rapidly changing values

### Common mistakes

#### Ignoring query parameters

```
// Wrong — same key for page=1 and page=2
const std::string key = "/api/products";

// Correct — use CacheKey builder
const std::string key = vix::cache::CacheKey::fromRequest("get", "/api/
users", "page=1", {}, {});
```

#### No memory limit

```
// Protect memory for long-running servers
LruMemoryStore::Config{.max_entries = 2048}
```

#### Not invalidating after writes

If a POST, PUT, PATCH, or DELETE changes data, old cached GET responses may become stale. Use short TTLs or explicit invalidation.

#### Caching error responses

A temporary 500 response should usually not be cached.

### What you should remember

Vix cache is built from explicit primitives: `Cache`, `CacheEntry`, `CachePolicy`, `CacheContext`, `store`, `CacheKey`.

For offline-first: `Online` prefers fresh, `Offline` optionally allows stale, `NetworkError` optionally allows stale.

The core idea: cache is not only an optimization — in Vix, cache is part of predictable behavior when the network is slow, unstable, or unavailable.

# Next chapter

**Next: Offline-first sync**

# 14. Offline-first Sync

In the previous chapter, you learned cache. Now you will learn offline-first sync.

```
local write → WAL → outbox → sync engine → transport → done
```

The core idea: **write locally first, persist the operation, sync when the network is available.**

## Why offline-first sync exists

Real applications do not run in perfect conditions. Networks fail, servers restart, requests timeout, devices go offline.

A normal online-first flow fails when the network is unavailable:

```
user action → network request → server response → local state updated
```

Offline-first sync uses a safer flow:

```
user action → local write → durable log → outbox → sync later
```

## The mental model

```
Never depend on the network to preserve user intent.
```

| Property | Meaning |
|---|---|
| `Durable` | The operation survives a process restart. |
| `Retryable` | Failed operations can be attempted again safely. |
| `Offline-first` | The app can continue working without network access. |

## Vix Sync architecture

```
Local Write → WAL → Outbox → SyncWorker → Transport → Done / Retry / Failed
```

| Component | Purpose |
| --- | --- |
| `Operation` | Describes one durable unit of work. |
| `Wal` | Stores an append-only operation history. |
| `Outbox` | Stores operations waiting to be synchronized. |
| `RetryPolicy` | Decides when failed operations should retry. |
| `NetworkProbe` | Checks whether network access is currently available. |
| `SyncWorker` | Processes operations that are ready to run. |
| `SyncEngine` | Orchestrates workers, retries, and recovery. |
| `ISyncTransport` | Sends operations to HTTP, P2P, or custom transport targets. |

## Public headers

```
#include <vix/sync.hpp>
#include <vix/net.hpp>
```

## Operation

```
vix::sync::Operation op;
op.kind = "http.post";
op.target = "/api/messages";
op.payload = R"({"text":"hello"})";
```

## Basic outbox lifecycle

```
// enqueue → claim → complete → Done
const std::string id = outbox->enqueue(op, t0);
auto ready = outbox->peek_ready(t0, 10);
const bool claimed = outbox->claim(id, t0 + 1);
const bool done = outbox->complete(id, t0 + 2);
```

# FileOutboxStore

```cpp
auto store = std::make_shared<vix::sync::outbox::FileOutboxStore>(
    vix::sync::outbox::FileOutboxStore::Config{
        .file_path = dir / "outbox.json",
        .pretty_json = true});

auto outbox = std::make_shared<vix::sync::outbox::Outbox>(
    vix::sync::outbox::Outbox::Config{.owner = "demo"},
    store);
```

# RetryPolicy

```cpp
vix::sync::RetryPolicy retry;
retry.max_attempts = 3;
retry.base_delay_ms = 500;
retry.factor = 2.0;

auto outbox = std::make_shared<vix::sync::outbox::Outbox>(
    vix::sync::outbox::Outbox::Config{
        .owner = "demo-retry",
        .retry = retry},
    store);
```

## Retryable vs permanent failure

```cpp
// Temporary failure — retry later
outbox->fail(id, "temporary network error", now + 2, true);

// Permanent failure — stop retrying
outbox->fail(id, "bad request", now + 2, false);
```

| Failure | Retry | Example |
|---|---|---|
| Network timeout | Yes | Temporary connection issue. |
| Server unavailable | Yes | HTTP `503` response. |
| Offline device | Yes | No active network connection. |
| Bad request | No | Invalid request payload. |
| Validation error | No | Missing required input field. |

## WAL

A WAL (Write-Ahead Log) is an append-only log of durable intent.

```
vix::sync::wal::Wal wal(vix::sync::wal::Wal::Config{
    .file_path = dir / "wal.log",
    .fsync_on_write = false});

const auto offset = wal.append(vix::sync::wal::WalRecord{
    .id = "op_1",
    .type = vix::sync::wal::RecordType::PutOperation,
    .ts_ms = t0,
    .payload = to_bytes(payload)});

wal.replay(0, [](const vix::sync::wal::WalRecord &rec)
          { std::cout << "id=" << rec.id << " type=" << static_cast<int>(rec.type)
<< "\n"; });
```

## Sync transport

```
namespace vix::sync::engine
{
  class ExampleTransport final : public ISyncTransport
  {
  public:
    SendResult send(const vix::sync::Operation &op) override
    {
      std::cout << "sending: " << op.id << " -> " << op.target << "\n";
      return SendResult{.ok = true};
    }
  };
}
```

## NetworkProbe

```
auto probe = std::make_shared<vix::net::NetworkProbe>(
    vix::net::NetworkProbe::Config{},
    [] { return true; });  // online

// Offline-first control
auto network_online = std::make_shared<std::atomic<bool>>(false);
auto probe = std::make_shared<vix::net::NetworkProbe>(
    vix::net::NetworkProbe::Config{},
    [network_online] { return network_online->load(); });

// Later when network returns
network_online->store(true);
```

## SyncEngine

```
vix::sync::engine::SyncEngine engine(
    vix::sync::engine::SyncEngine::Config{
        .worker_count = 1,
        .idle_sleep_ms = 0,
        .offline_sleep_ms = 0,
        .batch_limit = 10,
        .inflight_timeout_ms = 10'000},
    outbox,
    probe,
    transport);

const auto processed = engine.tick(now_ms());
```

## Complete local-first flow

```
local file write → WAL append → outbox enqueue → sync engine tick → transport sends
→ done
```

The critical rule: **local write happens before sync. Network is not responsible for preserving the user action.**

## Offline then recover pattern

```
1. local write succeeds
2. WAL stores intent
3. outbox stores operation
4. network is offline → engine sends nothing (processed = 0)
5. network comes back → network_online->store(true)
6. engine sends operation → operation becomes done
```

## Inflight recovery

If a worker claims an operation and then crashes:

```
claim → crash → restart → engine detects stale InFlight → requeue → worker sends →
Done
```

This is why durable outbox state matters.

## Recommended operation kinds

```
http.post, http.put, http.patch, http.delete
fs.write.sync, fs.delete.sync
db.insert.sync, db.update.sync
message.send, event.publish
```

## Design rules

### Persist before sending

Store the operation first, then send the request. Never send before persisting.

### Treat the network as optional

When the network is available, sync now. When it is unavailable, sync later.

### Make operations idempotent

Use stable operation IDs so the server can detect and ignore duplicates.

### Separate temporary and permanent failures

Retry temporary failures. Do not retry invalid data.

### Keep payloads replayable

Include enough data to replay the operation later.

#### Common mistakes

#### Sending before persisting

If the process crashes after sending but before storing, recovery becomes impossible. **Persist first.**

#### Treating offline as an error

Offline is a normal state in offline-first systems. The operation should remain pending.

#### Forgetting inflight recovery

If a worker crashes after claiming, use inflight timeout recovery to requeue.

#### Using non-idempotent remote writes

Use stable operation ids and deduplication on the receiver.

## When to use Vix Sync

Use Vix Sync when your application needs durable operations, offline execution, and safe synchronization.

Good use cases:

- Offline-first applications
- Local-first file synchronization
- Reliable message delivery
- Retry-safe background jobs
- Edge applications
- Unstable network environments
- P2P synchronization

### Production notes

- Enable fsync for stronger durability
- Use stable operation ids
- Make remote endpoints idempotent
- Separate retryable and permanent failures
- Monitor pending and failed outbox size
- Expose sync health in `/health`

### What you should remember

The core flow: `local write → WAL → outbox → sync engine → transport → done`

The WAL keeps durable history. The outbox keeps pending work. The sync worker processes ready operations. The transport sends to HTTP, P2P, or custom targets.

The core idea: **persist first, sync later, never lose user intent.**

## Next chapter

**Next: P2P**

# 15. P2P

In the previous chapter, you learned offline-first sync. Now you will learn P2P.

P2P means peer-to-peer. Instead of every node depending on a central server, nodes can discover each other, connect, exchange messages, and synchronize data.

```
node A ↔ node B
local write → WAL → outbox → P2P transport → peer → ack
```

## Why P2P exists

Real systems do not always have a perfect cloud connection. P2P gives Vix a way to build systems that communicate directly.

Use P2P when you want: local network discovery, direct node-to-node communication, peer synchronization, edge replication, offline-first data exchange, store-and-forward systems.

## The mental model

| Layer | Purpose |
|---|---|
| `Discovery` | Finds available peers on the network. |
| `Peer connection` | Connects the local node to another peer. |
| `Handshake` | Establishes peer identity and session metadata. |
| `Envelope` | Wraps messages with routing and protocol metadata. |
| `Framing` | Splits byte streams into complete protocol messages. |
| `Dispatch` | Decodes payloads and routes them to typed handlers. |
| `Sync messages` | Pushes WAL entries, sends acknowledgments, and pulls work. |

## Public headers

```cpp
#include <vix/p2p.hpp>
#include <vix/p2p_http.hpp>  // for HTTP control routes
```

## Message flow

```
typed message → payload bytes → envelope → frame → transport
→ peer → decode frame → decode envelope → dispatch typed message
```

## Envelope and framing

```cpp
#include <vix/p2p.hpp>

// Create and pack a Ping message
vix::p2p::msg::Ping ping;
ping.nonce = 42;
vix::p2p::Envelope outgoing = vix::p2p::pack::make_envelope(vix::p2p::MessageType::P
ing, ping);

// Frame it (length-prefix framing)
vix::p2p::framing::LengthPrefixVarint framer;
vix::p2p::Frame frame = framer.encode(outgoing.encode());

// Decode frame → envelope → message
vix::p2p::FrameDecodeResult decoded = framer.decode(frame.bytes);
vix::p2p::Envelope incoming = vix::p2p::Envelope::decode_or_throw(decoded.frames.fro
nt().bytes);
vix::p2p::msg::Ping roundtrip = vix::p2p::msg::Ping::decode_or_throw(incoming.payloa
d);
```

TCP is a byte stream — framing ensures the receiver knows where each message begins and ends.

## Handshake messages

```cpp
// Node A → Hello
vix::p2p::msg::Hello hello;
hello.nonce_a = 1001;
hello.node_id = "node-a";
hello.capabilities["proto"] = "1.0";

// Node B → HelloAck
vix::p2p::msg::HelloAck ack;
ack.nonce_a = 1001;
ack.nonce_b = 2002;

// Node A → HelloFinish
vix::p2p::msg::HelloFinish finish;
finish.nonce_a = 1001;
finish.nonce_b = 2002;
finish.signature = /* 64-byte signature */;
```

## Discovery announcement

```
vix::p2p::msg::DiscoveryAnnounce announce;
announce.node_id = "node-a";
announce.tcp_port = 9001;
announce.ts_ms = 17100000000000ULL;
announce.nonce = 987654321ULL;
announce.capabilities["proto"] = "1.0";

const std::string json = announce.to_json();
auto parsed = vix::p2p::msg::DiscoveryAnnounce::from_json(json);
```

## Memory router

```
vix::p2p::MemoryRouter router;
router.upsert_route("node-b", vix::p2p::Route{"edge-1", false, 8});
router.upsert_route("node-c", vix::p2p::Route{"relay-7", true, 4});

auto route = router.resolve("node-b");  // next_hop="edge-1", via_relay=false
router.remove_route("node-c");
```

## WAL sync messages

```
// Push WAL records to a peer
vix::p2p::msg::WalPush push;
push.seq_begin = 10;
push.seq_end = 12;
push.wal_bytes = make_fake_wal_bytes();

// Acknowledge applied sequence
vix::p2p::msg::WalAck ack;
ack.last_applied_seq = 12;

// Ask peer for pending operations
vix::p2p::msg::OutboxPull pull;
pull.target_node_id = "node-b";
pull.max_items = 64;
```

## Start a real P2P node

```
vix::p2p::NodeConfig cfg;
cfg.node_id = "node-a";
cfg.listen_port = 9001;
cfg.on_log = [](std::string_view line) { std::cout << line << "\n"; };

auto node = vix::p2p::make_tcp_node(cfg);
node->start();

const auto stats = node->stats();
// stats.peers_total, peers_connected, handshakes_started, handshakes_completed
```

## P2PRuntime and connect

```
vix::p2p::P2PRuntime runtime(node);
runtime.start();

vix::p2p::PeerEndpoint ep;
ep.host = "127.0.0.1";
ep.port = 9001;
ep.scheme = "tcp";

runtime.connect(ep);
// runtime.stats(), runtime.stop()
```

## UDP discovery

```
vix::p2p::DiscoveryConfig cfg;
cfg.self_node_id = "node-a";
cfg.self_tcp_port = 9001;
cfg.discovery_port = 37020;
cfg.mode = vix::p2p::DiscoveryMode::Broadcast;
cfg.announce_interval_ms = 1000;

auto on_peer = [](const vix::p2p::DiscoveryAnnouncement &a)
{
  std::cout << "discovered: " << a.node_id << " at " << a.host << ":" << a.port <<
"\n";
};

auto discovery = vix::p2p::make_udp_discovery(cfg, on_peer);
discovery->start();
auto snapshot = discovery->snapshot();
discovery->stop();
```

# P2P HTTP control surface

```cpp
#include <vix.hpp>
#include <vix/p2p.hpp>
#include <vix/p2p_http.hpp>

vix::p2p::P2PRuntime runtime(node);
runtime.start();

vix::p2p_http::P2PHttpOptions opt;
opt.prefix = "/p2p";
opt.enable_ping = true;
opt.enable_status = true;
opt.enable_peers = true;

vix::App app;
vix::p2p_http::registerRoutes(app, runtime, opt);
app.listen(8081, []() { std::cout << "HTTP API listening on 8081\n"; });
```

## HTTP control routes

| Route | Purpose |
|---|---|
| `GET /p2p/ping` | Runs a simple P2P smoke test. |
| `GET /p2p/status` | Returns current runtime statistics. |
| `GET /p2p/peers` | Lists known peers for this node. |
| `POST /p2p/connect` | Connects this node to another peer. |
| `GET /p2p/logs` | Returns recent runtime log entries. |

```bash
curl http://127.0.0.1:8081/p2p/ping
curl http://127.0.0.1:8081/p2p/status

curl -X POST http://127.0.0.1:8083/p2p/connect \
  -H "content-type: application/json" \
  -d '{"host":"127.0.0.1","port":9201,"scheme":"tcp"}'
```

**Important vix run rule**

```
# Correct — --run passes args to your program
vix run p2p_manual_connect.cpp --run server
vix run p2p_manual_connect.cpp --run client 127.0.0.1 9101

# Wrong — -- is for compiler/linker flags
vix run p2p_manual_connect.cpp -- server
```

## How P2P connects to sync

P2P can become one transport for the sync engine:

```
WAL → WalPush → peer receives → peer applies → WalAck
```

The P2P layer moves sync data between nodes. The sync layer decides what must be durable, retried, replayed, and acknowledged.

## P2P vs WebSocket vs HTTP

| Criteria | WebSocket | P2P | HTTP |
|---|---|---|---|
| Connection | Client -> server. | Node -> node. | Request -> response. |
| Best for | Browser real-time apps. | Distributed systems. | Standard APIs. |
| Discovery | Manual configuration. | UDP discovery or registry. | Manual configuration. |

**Common mistakes**

**Sending raw bytes without framing**

TCP does not preserve message boundaries. Use framing before decoding.

**Trusting unknown peers**

Use handshake, identity, and security checks before trusting a peer.

**Forgetting idempotency**

A peer can receive the same sync message more than once. Use operation ids and WAL sequence numbers to deduplicate.

**Exposing P2P HTTP control routes without auth**

Routes like `POST /p2p/connect` must be protected in production.

### Production notes

- Use stable node ids
- Protect control routes with authentication
- Use secure transport
- Make sync messages idempotent
- Monitor peer counts and handshake failures
- Log peer ids and operation ids

```
{
  "ok": true,
  "p2p": {
    "node_id": "node-a",
    "peers_total": 3,
    "peers_connected": 2,
    "handshakes_completed": 2
  }
}
```

### What you should remember

The basic message flow: `typed message → envelope → frame → transport → peer → decode → dispatch`

The runtime flow: `node starts → peer discovered → handshake → messages exchanged`

The sync flow: `WAL → WalPush → peer applies → WalAck`

P2P does not replace offline-first sync — it complements it.

The core idea: **Vix Sync preserves intent. Vix P2P moves that intent between nodes.**

## Next chapter

**Next: Production deployment**

# 16. Production Deployment

In the previous chapter, you learned P2P. Now you will learn how to deploy a Vix application in production.

```
browser → HTTPS → Nginx → Vix app → systemd
```

## Why production deployment matters

During development: `vix dev`.

In production, your application needs:

- A release build
- A stable working directory
- Environment variables
- Automatic restart
- Logs
- A reverse proxy
- HTTPS
- Health checks
- Predictable ports

## Production architecture

```
Internet → Nginx → 127.0.0.1:8080 → Vix app → systemd
```

Nginx handles public HTTP/HTTPS. The Vix app listens locally. systemd keeps the app alive.

## Development vs production

| Development | Production |
| --- | --- |
| `vix dev` | Release binary. |
| Hot reload | Stable process. |
| Terminal logs | Systemd logs. |
| Local browser | Nginx reverse proxy. |
| Debug settings | Production environment. |
| Manual restart | Automatic restart. |

## Prepare the server

```
sudo apt update
sudo apt install -y \
  build-essential cmake ninja-build pkg-config \
  nginx certbot python3-certbot-nginx \
  libssl-dev libsqlite3-dev zlib1g-dev libbrotli-dev \
  nlohmann-json3-dev libspdlog-dev libfmt-dev
```

## Create a production user

```
sudo useradd --system --create-home --shell /usr/sbin/nologin vix
sudo mkdir -p /home/vix/apps/myapp
sudo chown -R vix:vix /home/vix/apps
```

## Configure environment

```
sudo -u vix nano /home/vix/apps/myapp/.env
```

```
SERVER_PORT=8080
SERVER_HOST=127.0.0.1
SERVER_TLS_ENABLED=false
VIX_LOG_LEVEL=info
VIX_LOG_FORMAT=kv
VIX_COLOR=never
# SQLite
DATABASE_ENGINE=sqlite
DATABASE_DEFAULT_NAME=/home/vix/apps/myapp/data/app.db
# MySQL (if used)
DATABASE_ENGINE=mysql
DATABASE_DEFAULT_HOST=127.0.0.1
DATABASE_DEFAULT_PORT=3306
DATABASE_DEFAULT_USER=myapp
DATABASE_DEFAULT_PASSWORD=change-me
DATABASE_DEFAULT_NAME=myapp
```

> *When Nginx handles HTTPS, keep `SERVER_TLS_ENABLED=false` — Nginx terminates TLS and proxies to local Vix.*

## Build a release binary

```
cd /home/vix/apps/myapp
sudo -u vix vix build --preset release
# With SQLite:
sudo -u vix vix build --preset release --with-sqlite
# With MySQL:
sudo -u vix vix build --preset release --with-mysql
```

## Test the binary manually

```
cd /home/vix/apps/myapp
sudo -u vix ./build-release/myapp
```

```
# In another terminal
curl -i http://127.0.0.1:8080/health
```

Stop with `Ctrl+C` then proceed to systemd.

## Create a systemd service

```
sudo nano /etc/systemd/system/vix-myapp.service
```

```
[Unit]
Description=Vix MyApp service
After=network.target
[Service]
Type=simple
User=vix
Group=vix
WorkingDirectory=/home/vix/apps/myapp
ExecStart=/home/vix/apps/myapp/build-release/myapp
Restart=always
RestartSec=3
Environment=VIX_LOG_LEVEL=info
Environment=VIX_LOG_FORMAT=kv
Environment=VIX_COLOR=never
LimitNOFILE=65535
[Install]
WantedBy=multi-user.target
```

```
sudo systemctl daemon-reload
sudo systemctl enable vix-myapp
sudo systemctl start vix-myapp
sudo systemctl status vix-myapp
```

## Read logs

```
journalctl -u vix-myapp -f         # follow
journalctl -u vix-myapp -n 100     # last 100 lines
journalctl -u vix-myapp -b         # since boot
```

## Nginx reverse proxy

```
sudo nano /etc/nginx/sites-available/myapp
```

```
server {
    listen 80;
    server_name example.com [www.example.com](https://www.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;
    }
}
```

```
sudo ln -s /etc/nginx/sites-available/myapp /etc/nginx/sites-enabled/myapp
sudo nginx -t
sudo systemctl reload nginx
curl -i http://example.com/health
```

## Nginx for WebSocket

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

## Enable HTTPS

```
sudo certbot --nginx -d example.com -d [www.example.com](https://www.example.com)
sudo certbot renew --dry-run
```

After Certbot, your config adds SSL and a redirect from HTTP to HTTPS automatically.

## Health checks

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

Production health with more detail:

```
{
  "ok": true,
  "service": "myapp",
  "database": "ok",
  "sync": "enabled"
}
```

## Deployment flow

```
cd /home/vix/apps/myapp
sudo -u vix git pull
sudo -u vix vix build --preset release
sudo systemctl restart vix-myapp
curl -i http://127.0.0.1:8080/health
curl -i https://example.com/health
```

## Useful systemd commands

```
sudo systemctl start|stop|restart|status vix-myapp
sudo systemctl enable|disable vix-myapp
journalctl -u vix-myapp -f
```

## Useful Nginx commands

```
sudo nginx -t
sudo systemctl reload nginx
sudo systemctl status nginx
ls /etc/nginx/sites-enabled
```

## Firewall

```
sudo ufw allow OpenSSH
sudo ufw allow 80
sudo ufw allow 443
sudo ufw enable
```

## Database production notes

**SQLite:**

```
DATABASE_ENGINE=sqlite
DATABASE_DEFAULT_NAME=/home/vix/apps/myapp/data/app.db
```

**MySQL:**

```
CREATE DATABASE myapp;
CREATE USER 'myapp'@'localhost' IDENTIFIED BY 'change-me';
GRANT SELECT, INSERT, UPDATE, DELETE ON myapp.* TO 'myapp'@'localhost';
FLUSH PRIVILEGES;
```

# Common production errors

## 502 Bad Gateway

Nginx cannot reach the Vix app. Check: `sudo systemctl status vix-myapp` and `curl -i http://127.0.0.1:8080/health`.

## 504 Gateway Timeout

App accepted the connection but did not respond fast enough. Check for slow database queries or overloaded VPS.

## WebSocket closes immediately

Add Nginx upgrade headers and longer timeouts.

## App works locally but not through domain

Check: `sudo nginx -t`, DNS records, firewall, TLS certificate.

### Common mistakes

#### Running the app as root

Always use a dedicated user: `User=vix`.

#### Forgetting the working directory

```
WorkingDirectory=/home/vix/apps/myapp
```

Relative paths ( `.env` , `public/` , `data/` ) depend on this.

#### Using debug logs forever

```
VIX_LOG_LEVEL=info
```

#### Exposing P2P control routes without auth

Routes like `POST /p2p/connect` must be protected or kept internal.

# Deployment checklist

- [ ] Release build works
- [ ] `.env` exists
- [ ] App listens on localhost
- [ ] Health route works locally
- [ ] systemd service starts
- [ ] systemd restarts after failure
- [ ] Logs visible with `journalctl`

- [ ] Nginx config passes `nginx -t`
- [ ] Domain points to server
- [ ] HTTPS enabled
- [ ] Certbot renewal works
- [ ] Database credentials not hardcoded
- [ ] Admin routes protected

## Recommended production structure

```
/home/vix/apps/myapp/
├── build-release/
│   └── myapp
├── data/
│   └── app.db
├── public/
├── src/
├── .env
└── vix.json
/etc/systemd/system/vix-myapp.service
/etc/nginx/sites-available/myapp
/etc/nginx/sites-enabled/myapp
```

### What you should remember

A Vix production app is a normal native Linux service.

```
browser → HTTPS → Nginx → Vix app on localhost → systemd
```

```
vix build --preset release   # build
sudo systemctl start vix-myapp  # run
sudo nginx -t && sudo systemctl reload nginx  # expose
sudo certbot --nginx -d example.com  # HTTPS
```

The core idea: **development uses `vix dev`, production runs a release binary.**

## Next chapter

**Next: Next steps**

# 17. Next Steps

You have reached the end of the Vix book.

You started with one simple idea:

```
Run C++ code quickly.
```

Then you built the mental model step by step:

```
CLI → Runtime → Application → Modules → Production
```

## What you now understand

The main Vix workflow:

```
vix run main.cpp
vix new api
vix dev
vix build
vix check
vix tests
```

The main application model:

```cpp
#include <vix.hpp>
using namespace vix;
int main()
{
  App app;
  app.get("/", [](Request &, Response &res){
    res.json({"message", "Hello from Vix"});
  });
  app.run(8080);
  return 0;
}
```

# The path you completed

| Stage | What you learned |
|---|---|
| Start | What Vix is and why it exists. |
| CLI | Running files, creating projects, building, testing, and checking code. |
| HTTP | Building routes with `App`, `Request`, and `Response`. |
| APIs | Building JSON APIs. |
| Layers | Adding middleware, validation, errors, and logging. |
| Data | Using SQLite, MySQL, and database access. |
| Realtime | Using WebSocket and the async runtime. |
| Reliability | Using cache and offline-first synchronization. |
| Distributed | Building P2P features. |
| Production | Deploying with Nginx, systemd, TLS, logs, and health checks. |

# What to build next

The best next step is to build one real app from start to finish.

**A good first real Vix app:**

```
GET  /
GET  /health
GET  /users
GET  /users/{id}
POST /users
POST /auth/register
POST /auth/login
GET  /auth/me
```

With:

- validation,
- SQLite storage,
- structured errors,
- logs,
- production deployment.

## Recommended project

```
vix new users-api
cd users-api
vix dev
```

Build step by step:

1. Add `/health`
2. Add `/users`
3. Add JSON responses
4. Add validation
5. Add SQLite
6. Add structured errors
7. Add logs
8. Build release
9. Deploy behind Nginx and systemd

## Use the Guides section

The book teaches the story. The guides help you solve specific problems:

- Build a REST API
- Validation
- Authentication
- Sessions
- CORS
- Rate limiting
- SQLite API
- MySQL API
- WebSocket chat
- Static files
- Templates
- Production: Nginx + systemd

## Use the CLI reference

When you need command details:

```
vix run main.cpp --run --port 8080   # runtime args
vix run main.cpp -- -O2 -DNDEBUG    # compiler flags
vix build --preset release
vix build --with-sqlite
```

> Remember: `--` = compiler/linker flags, `--run` = runtime arguments to your program.

## Learn by layers

| Layer | What to learn |
|---|---|
| 1 | `vix run main.cpp` — run C++ files |
| 2 | `App`, routes, `Request`, `Response` — HTTP APIs |
| 3 | Middleware, validation, errors, logging |
| 4 | SQLite or MySQL |
| 5 | WebSocket for realtime |
| 6 | Cache and sync for reliability |
| 7 | P2P for distributed behavior |
| 8 | Release build, systemd, Nginx, HTTPS |

## Recommended learning order after the book

1. Build a REST API
2. Add validation and structured errors
3. Add SQLite
4. Add authentication
5. Deploy with Nginx and systemd
6. Add WebSocket
7. Add cache
8. Add offline-first sync
9. Add P2P
10. Study internals and performance

# Production checklist

**App:**

- [ ] Health route exists
- [ ] Errors use consistent JSON shape
- [ ] Input is validated
- [ ] Logs are structured
- [ ] Secrets are not logged

**Build:**

- [ ] Release build works
- [ ] Required flags enabled (`--with-sqlite`, `--with-mysql`)
- [ ] Dependencies installed

**Runtime:**

- [ ] App runs as non-root user
- [ ] systemd restarts it
- [ ] Working directory is correct

**Network:**

- [ ] App listens locally
- [ ] Nginx proxies public traffic
- [ ] HTTPS enabled
- [ ] WebSocket upgrade headers configured if needed

**Data:**

- [ ] Database path is stable
- [ ] Credentials in environment
- [ ] Backups exist

**Security:**

- [ ] Admin routes protected
- [ ] P2P control routes protected
- [ ] CORS configured correctly
- [ ] Rate limiting enabled where needed

# What makes a good Vix application

Keep `main()` small:

```cpp
int main()
{
  config::Config cfg{".env"};
  App app;
  configure_middlewares(app);
  register_public_routes(app);
  register_user_routes(app);
  app.run(cfg.getServerPort());
  return 0;
}
```

Use predictable response shapes:

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

## When to use each runtime feature

| Feature | Use when |
|---|---|
| `vix run` | You need to run one file quickly. |
| `vix new` | You want to start a real project. |
| HTTP | You are building APIs or web routes. |
| JSON | You need structured API responses. |
| Middleware | You need shared request behavior. |
| Validation | You are accepting user input. |
| Database | You need durable application state. |
| WebSocket | You need realtime client updates. |
| Async | You need timers, signals, or non-blocking I/O. |
| Cache | You need speed or stale data under failure. |
| Sync | You must not lose local operations. |
| P2P | You need nodes to discover, connect, or replicate. |
| Production | Your app must run as a service. |

# A final example direction

**Reliable Notes API:**

```
POST   /notes
GET    /notes
GET    /notes/{id}
PATCH  /notes/{id}
DELETE /notes/{id}
GET    /health
```

Grow it: authentication → WebSocket updates → offline-first outbox → P2P sync → production deployment.

This kind of project uses almost everything you learned.

### What you should remember

The full Vix path:

```
one C++ file → Vix project → HTTP API → professional API layers
→ database → realtime → async runtime → cache → offline-first sync
→ P2P → production deployment
```

The most important command is still:

```
vix run main.cpp
```

The most important production command is:

```
vix build --preset release
```

The most important mental model is:

```
Vix is a modern C++ runtime for building fast and reliable applications.
```

The final idea: **start simple, build progressively, deploy for real.**

---

*End of the Vix Book. You are ready to build real applications with Vix. Choose one project and build it completely.*

---