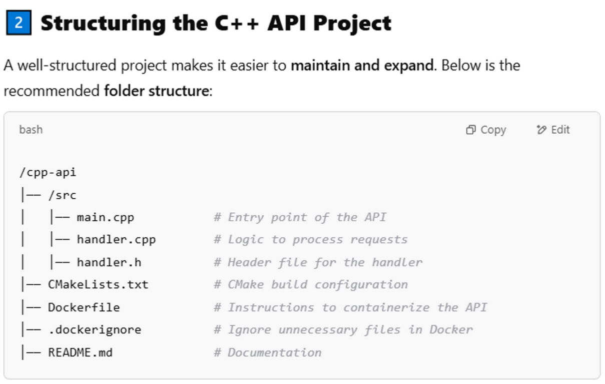
API

terça-feira, 1 de abril de 2025 07:32

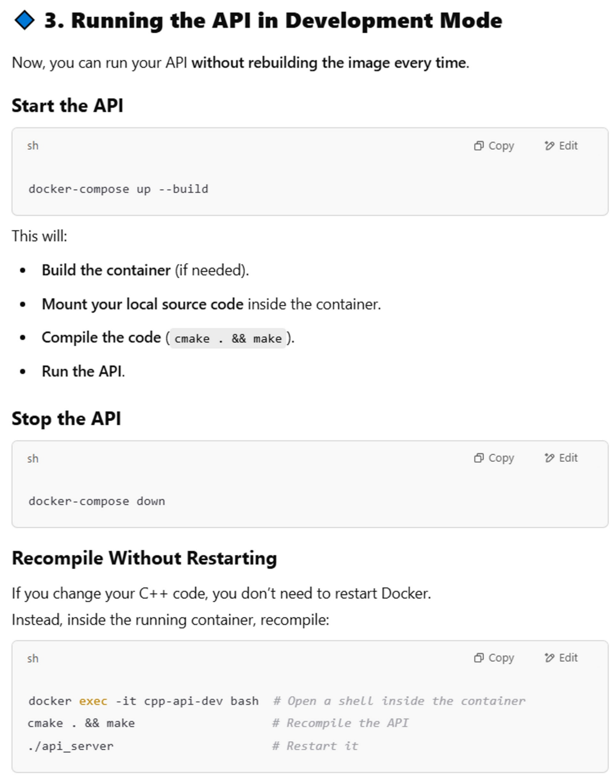
- CppRestSDK is not an option anymore, no support for years and the github page recommend not to use it
- Pistache
  - May lack documentation.... but it's something I can contribute... they are looking for voluntiers
- **Crow**
  - Seems like a good option, has support (github page s active) https://github.com/CrowCpp/Crow
  - Looks like there are some documentation arround it https://crowcpp.org/master/getting_started/setup/linux/
  - GPT gave a solid answer... that is nice



2 Structuring the C++ API Project

A well-structured project makes it easier to **maintain and expand**. Below is the recommended **folder structure**:

```bash
/cpp-api
│── /src
│   │── main.cpp          # Entry point of the API
│   │── handler.cpp       # Logic to process requests
│   │── handler.h         # Header file for the handler
│── CMakeLists.txt        # CMake build configuration
│── Dockerfile            # Instructions to containerize the API
│── .dockerignore         # Ignore unnecessary files in Docker
│── README.md             # Documentation
```



🔷 3. Running the API in Development Mode

Now, you can run your API **without rebuilding the image every time**.

Start the API

```sh
docker-compose up --build
```

This will:

- **Build the container** (if needed).
- **Mount your local source code** inside the container.
- **Compile the code** (`cmake . && make`).
- **Run the API**.

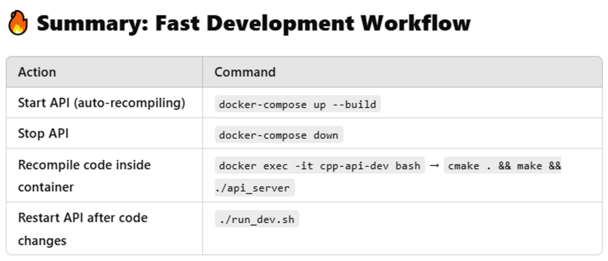
Stop the API

```sh
docker-compose down
```

Recompile Without Restarting

If you change your C++ code, you don't need to restart Docker.
Instead, inside the running container, recompile:

```sh
docker exec -it cpp-api-dev bash  # Open a shell inside the container
cmake . && make                   # Recompile the API
./api_server                      # Restart it
```



🔥 Summary: Fast Development Workflow

Action	Command
Start API (auto-recompiling)	`docker-compose up --build`
Stop API	`docker-compose down`
Recompile code inside container	`docker exec -it cpp-api-dev bash` → `cmake . && make && ./api_server`
Restart API after code changes	`./run_dev.sh`

1 Build & Start Container

docker-compose up --build

2 Manually Rebuild (if needed)

docker exec -it cpp-api-dev bash
cd build && cmake .. && make -j$(nproc)
./api_server # Run manually

3 Stop Container

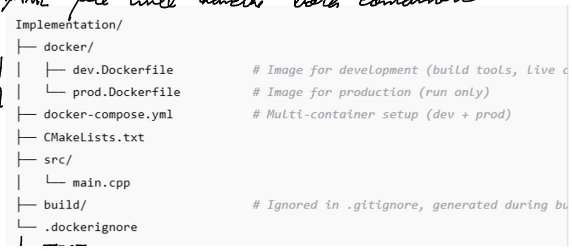
docker-compose down

Definição
- Rotas
- Entidades
- Recursos
- Dominios

Brainstorm

terça-feira, 8 de abril de 2025
20:45

• YAML file will handle both containers



```
Implementation/
├── docker/
│   ├── dev.Dockerfile          # Image for development (build tools, live c
│   └── prod.Dockerfile         # Image for production (run only)
├── docker-compose.yml          # Multi-container setup (dev + prod)
├── CMakeLists.txt
├── src/
│   └── main.cpp
├── build/                      # Ignored in .gitignore, generated during bu
└── .dockerignore
```

Just a draft

1. Development Dockerfile (docker/dev.Dockerfile)

Dockerfile
CopyEdit
FROM ubuntu:22.04
# Install build dependencies
RUN apt-get update && apt-get install -y \
   g++ cmake git libboost-all-dev libasio-dev \
   && rm -rf /var/lib/apt/lists/*
# Install Crow
RUN git clone --branch v1.2.1.2 https://github.com/CrowCpp/Crow.git /Crow \
   && cd /Crow && mkdir build && cd build \
   && cmake .. -DBUILD_SHARED_LIBS=ON && make -j$(nproc) && make install

Also need to include GTests in the installation

# 3. Instalar GoogleTest e GoogleMock
RUN git clone
https://github.com/google/googletest.git
/tmp/googletest && \

```
├── src/
│   └── main.cpp
├── build/                    # Ignored in .gitignore, generated during bu
└── .dockerignore
└ TEST
   └ my-TEST-cpp
```

yaml

```yaml
version: "3.8"
services:
  cpp-dev:
    build:
      context: .
      dockerfile: docker/dev.Dockerfile
    container_name: cpp-api-dev
    volumes:
      - .:/app
    working_dir: /app
    ports:
      - "8080:8080"
    command: >
      bash -c "mkdir -p build && cd build && cmake .. && make -j$(nproc) && .

  cpp-api:
    build:
      context: .
      dockerfile: docker/prod.Dockerfile
    container_name: cpp-api-prod
    ports:
      - "8080:8080"
    depends_on:
      - cpp-dev
```

```
   g++ cmake git libboost-all-dev libasio-dev \
   && rm -rf /var/lib/apt/lists/*
# Install Crow
RUN git clone --branch v1.2.1.2 https://github.com/CrowCpp/Crow.git /Crow \
   && cd /Crow && mkdir build && cd build \
   && cmake .. -DBUILD_SHARED_LIBS=ON && make -j$(nproc) && make install
WORKDIR /app
```

```
# 3. Instalar GoogleTest e GoogleMock
RUN git clone
https://github.com/google/googletest.git
/tmp/googletest && \
   cd /tmp/googletest && \
   cmake -S . -B build && \
   cmake --build build -j$(nproc) && \
   cmake --install build && \
   rm -rf /tmp/googletest
```

2. Production Dockerfile (docker/prod.Dockerfile)

Dockerfile
CopyEdit

```dockerfile
FROM ubuntu:22.04
# Install runtime-only dependencies
RUN apt-get update && apt-get install -y \
   libboost-all-dev libasio-dev \
   && rm -rf /var/lib/apt/lists/*
COPY build/api_server /app/api_server
WORKDIR /app
EXPOSE 8080
CMD ["./api_server"]
```

💻 4. Development Workflow

Run Dev Environment:

docker-compose up --build

bash

```bash
docker-compose up cpp-dev
```

Run tests manually in dev container:

```
cd build
ctest --output-on-failure
```

Recompile after code change (automatically with volume):

- Change code in `src/main.cpp`
- Inside the container (`cpp-dev`), rerun:

bash

```bash
cd build && make
```

docker-compose up --build

🌐 5. Running the API (Dev or Prod)

- From host (Windows or WSL), call:

bash

```bash
curl -X POST http://localhost:8080/api/analyze -d '{"code":"class A {}"}'
```

Or test with Postman:

- **POST URL:** `http://localhost:8080/api/analyze`
- **Body (JSON):**

json

```json
{
  "code": "class A {}"
}
```

✅ Summary

Goal	Solution
Dev on Linux	Use `cpp-dev` container with volume mapping
Run API	Auto-run in dev or manually in `cpp-api`
CI/CD	GitHub Actions with `docker build`
Installer	Use `cpack` or `makeself` after build

- Usar o tunder client ou postman?
- Pensar no versionamento das apis e como testar elas
  - Rodar uma bateria de testes na API a cada versão
    - Uma aplicação que comunca com uma API e faz avaliação das respostas eseradas
- Camada de segurança (não vou preocupar com isso agora
  - Permito aquela origem
- Rcebe um endereço no github, baixa os arquivos e processa eles

Notes to consider developing the API

This folder structure will be very important to organize version of the implementations!!!!!

makefile   Copy   Edit

```
Implementation/
├── CMakeLists.txt
├── docker-compose.yml
├── Dockerfile
├── include/                    # Public headers
│   ├── api/
│   │   ├── v1/
│   │   │   ├── analyzer_route.h
│   │   │   └── ...
│   └── ...
├── src/                        # Source files
│   ├── main.cpp
│   ├── api/
│   │   ├── v1/
│   │   │   ├── analyzer_route.cpp
│   │   │   └── ...
│   └── ...
├── tests/                      # Unit tests
│   └── ...
├── build/                      # Generated by cmake
├── data/                       # Optional, for test inputs or configs
└── scripts/                    # Utility or build scripts
```

The APIs are going to be the entry point of the main logic. Which will be implemented later on... but this won't work... because the API is more likely to be static and the logic changes.

Ok, i may need to duplicate code here?

Ok... i may need a database...

upload a project... parse everything and populate this database

The following requests on query them from database.... A CRUD... I would need to do it at some point... because or i create a database or store everything in memory

Create a database it is...

How to create APIs with crow in c++

- Recommended Folder Structure
  - Follow the folder structure for includes and source files
  - Inside the V1 folder, organize by domains
    - Analyze domain, database domain, retreve information domain....
- Looks like the API structure is like lambdas functions
  - This is saying: “when someone does a GET /status, run this small anonymous function and return the result.”
- We need to define the route and the HTTP method required to be used there
  - `CROW_ROUTE(app, "/status").methods(crow::HTTPMethod::GET)`
- Why put the implementation in a .h file?
  - in **Crow**, we use .h for route setup
    - The lambdas are templates and must be visible during compilation. If you split them between .cpp and .h, the compiler won’t know the lambda's exact type in another compilation unit.
    - Crow itself is mostly a header-only library (no .so or .a files to link). Your handlers behave better when defined the same way.
- You can use classes to group APIs by domain:
  - **organizing routes by domain in classes** is a best practice to keep code clean and modular.
    - You group related endpoints (e.g., /user, /auth, /product) in their own files.
    - Each class encapsulates its setup.
    - Easier to reuse shared logic inside those classes
  - We can add to the other folder and call the implementation, there is na example ->>>>>

cpp

```cpp
#pragma once

#include <crow.h>
#include "../services/CalculatorService.h"

class CalculatorRoutes {
public:
    static void init(crow::SimpleApp& app) {
        CROW_ROUTE(app, "/v1/calc/add").methods("POST"_method)
        ([](const crow::request& req) {
            auto body = crow::json::load(req.body);
            if (!body || !body.has("a") || !body.has("b"))
                return crow::response(400, "Missing parameters");

            int a = body["a"].i();
            int b = body["b"].i();
            int result = CalculatorService::add(a, b);

            crow::json::wvalue res;
            res["result"] = result;
            return crow::response(res);
        });
    }
};
```

cpp

```cpp
#include <crow.h>
#include "routes/CalculatorRoutes.h"

int main() {
    crow::SimpleApp app;

    CalculatorRoutes::init(app);

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