B5 - Year-End Project

B-YEP-500

Zia

Communication is key

Zia

binary name: zia
language: C++11 (or more)
build tool: mkdir build && cd build && conan install .. --build=missing && cmake .. -G "Unix Makefiles" && cmake --build .

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.

The goal of the **Zia** project is to create an HTTP server. This server will be able to serve typical HTTP documents and page requests, as well as CGI execution and more. The server **MUST** be written in C++, with support for interoperable modules.

Introduction

All the knowledge you've acquired from previous C++ knowledge units will be put to use in this final, large scale project. To finish this project in time, you'll need to make clever use of all the abstractions you've created up until now. This project will be your last object-oriented design and implementation at **Epitech** (except for your end of studies project, of course).

During this project, every design diagram and any code you write must be of **professional** quality. Think! Write code you'll be proud of!

However, design and implementation are not at the heart of the **Zia** project. The core part of this project is to overcome a challenge none of you have faced during your scholarship: **teaming up with your city-wide class** to use the same API! More on this will come.

Project lifetime

The **Zia** project is split into 3 big steps:

- The creation of the API
- The local API election
- The implementation/creation of all the modules

These items will all be discussed in details below.

Common parts

Protocol

The **Zia MUST** be an exhaustive implementation of the **HTTP/1.1** protocol, as described in RFC 2616, with the exception of proxy support.

If you wrote your own RFC during the **R-Type** project, reading this big one shouldn't be a problem. Refer to the **R-Type**'s subject for more information about RFCs.

Here is a **NON-EXHAUSTIVE** list of what we consider mandatory to know about **RFC 2616**:

- The request structure
- The response structure
- Http methods
- Http response codes

Server Configuration

The server must be fully configurable by means of a configuration file. Any software configuration done by re-compiling the program (such as macros) will **NOT** be accepted and have severe consequences.

You an use any format for your configuration files, such as XML, Json, INI…

Some notes about the configuration file:

- You **MAY** use a parsing library specific to XML, Json or any other language you choose. Before you ask, **Boost::Spirit** is **NOT** authorized.
- You **MUST** implement a regular and coherent parser if you choose to not use any parsing library.
- You **MUST NOT** use XML or Json libraries for any other purpose than parsing the configuration file. This will be considered as cheating.
- The server **MUST NOT** crash if it the configuration file is corrupt or missing: you **MUST** set default values.
- The configuration file **MUST NOT** be opened with an absolute path.
- It **MUST** be possible to reload the configuration file without restarting nor recompiling the server.

Modules

As a modular server, the **Zia MUST** be able to handle modules. The **Zia executable** could be seen as an empty shell that must be filled with **modules** in order to work.

A module is an atomic processing unit that can receive input from other modules, and send output to them. It **MUST** be possible to use any number of modules together to create a processing line that will handle an HTTP request and create an appropriate HTTP response.

You **MUST** design a complete Application Programming Interface to handle you **Zia**'s modules.

Although submitting your API to the election is optional, we greatly encourage you to do design one and present it in the first follow-up.

Once the election is finished, your modules **MUST** conform to the locally choosen API.

No matter what API is elected, each group **MUST** provide two mandatory modules: the **secure connection module** and the **PHP CGI** module. Once these two modules work perfectly, you **MUST** add as many other modules as you like in order to raise your final grade.

As always, when designing your API, question its flexibility. How easy would it be to add a **log** module that would keep a log file of all incoming requests, and that would let other modules send it log messages? How easy would it be to add a **video game** module that would be a **Snake** clone in which each incoming request spawns a food item?

Having a flexible API will make it possible for you to add any custom behavior to your **Zia**, which could then serve as a generic modular program. Take a look at the **Mediator** design pattern for inspiration.

Interoperability

Modules from different groups **MUST** be interoperable. A module from a given group **MUST** run seamlessly in another group's server. It **MUST** be possible to add or remove modules **without recompiling or restarting the server**.

This task is particularly complex and requires some reflection, research and discussion among groups. Note that once the API has been chosen, every group will have to use it, whether it is good or bad.

Here are a few rules:

- Modules **MUST** be able to hook up to any stage of request processing, from connection establishment to page rendering.
- The server **MUST** be able to load and unload modules dynamically.

To make it possible for your modules to run with any group's server citywide, you **MUST** conform to the elected API.

Secure Connection Module

The server **MUST** let clients establish secure connections using **SSL** or **TLS**. This feature **MUST** be a module. You are allowed to use **OpenSSL**.

PHP CGI Module

This module **MUST** make it possible for the server to execute **PHP** scripts. The scripts **MUST** run as CGI.

Development constraints

General

Your **Zia** **MUST** compile and run on at least one **Windows** distribution **AND** one **Unix** distribution.

It **MUST** be built using a [CMake] and dependencies **MUST** be handled using [conan].
These, and only these, conan repositories may be used:

- conancenter: `https://center.conan.io`
- bincrafters: `https://bincrafters.jfrog.io/artifactory/api/conan/public-conan`

The build of your project will be done in the following fashion (for Unix systems):

Terminal

```
~/B-YEP-500> mkdir build && cd build && conan install .. --build=missing && cmake
.. -G "Unix Makefiles" && cmake --build .
```

For Windows, it must generate a Visual Studio solution file.

Conan should be set to build the requirements using C++11 (see `compiler.libcxx` config).
Your project must also be built using C++11 at the very least.

Your **Zia** **MUST** be multi-threaded.

You **SHOULD** use all the abstractions you've designed and implemented in C++ up until now.

You **SHOULD** write **unit tests**.

You **MUST** write **C++** code. **C** and **C+** will **NOT** be tolerated.
You **MAY** use Boost:Asio.

Testing modules and servers

During your final defense, your server must be usable with these client programs:

- Telnet
- Lynx
- Google Chrome
- Mozilla Firefox
- Opera
- MS Edge
- Siege

Project steps

API creation

Create an API capable of handling modules and think wisely about the architecture of it.
You're greatly encouraged to come and present your API to the first follow-up to get feedbacks on it and improve your architecture skills.

API Election

The election will take place after the first follow-up, talk with your pedagogical team to know the details of submission. But here are some general rules:

- Submitting your API is optional but greatly encouraged
- If the elected group ignores help requests from other groups or cities, it will face severe consequences.
- Having your API elected is rewarded during the unit validation.

Once the election is finished, you will start implementing your **Zia** by **CONFORMING TO** the elected API.

Implementation follow-ups

These follow-ups let us assert your progress on the project and evaluate its quality. It is very important that you be very dynamic during these follow-ups, as they are the last steps before the final defense.

General setpoints

You are (more or less) free to implement the project any way you please. However, here are a few restrictions:

- The only authorized functions from the **libc** are the ones that wrap system calls (and don't have C++ equivalents!)
- Any solution to a problem **MUST** be object-oriented.
- Any not explicitly authorized library is explicitly forbidden.
- Any value passed by copy instead of reference or pointer MUST be justified.
- Any member function or method that does not modify the current instance and is not **const** MUST be justified.
- Any code that is deemed unreadable, unmaintainable or with unnecessary performance costs **WILL** be sanctioned. Be rigorous! Write code you'll be proud of!