**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук

Образовательная программа бакалавриата «Программная инженерия»

|  |  |  |
| --- | --- | --- |
| СОГЛАСОВАНО  Преподаватель департамента программной инженерии ФКН, кандидат компьютерных наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_С.А.Виденин  \_\_\_\_\_\_\_\_\_\_\_10\_марта 2024 г. |  | УТВЕРЖДАЮ  Академический руководитель образовательной программы «Программная инженерия»  профессор департамента программной инженерии, кандидат технических наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н.А. Павлочев  \_\_\_\_\_\_\_\_\_\_\_\_10\_марта\_2024 г. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | ***Подп. и дата*** |  | | ***Инв. № дубл.*** |  | | ***Взам. Инв. №*** |  | | ***Подп. и дата*** |  | | ***Инв. № подл.*** |  | | **ВЫСОКОПРОИЗВОДИТЕЛЬНЫЙ НАСТРАИВАЕМЫЙ HTTP СЕРВЕР**  **Пояснительная записка**  **ЛИСТ УТВЕРЖДЕНИЯ**  **RU.17701729.04.04-01 01-1-ЛУ**  Исполнитель  студент группы БПИ214  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Е.К.Фортов/  \_\_\_\_\_\_\_\_\_\_\_\_\_10\_марта\_2024 г. | |
|  |  |

УТВЕРЖДЕН

RU.17701729.04.04-01 01-1-ЛУ

|  |  |
| --- | --- |
| ***Подп. и дата*** |  |
| ***Инв. № дубл.*** |  |
| ***Взам. инв. №*** |  |
| ***Подп. и дата*** |  |
| ***Инв. № подл*** |  |

**ВЫСОКОПРОИЗВОДИТЕЛЬНЫЙ НАСТРАИВАЕМЫЙ HTTP СЕРВЕР**

**Пояснительная записка  
  
RU.17701729.04.04-01 01-1-ЛУ**

**Листов 33**

Оглавление

[1. ВВЕДЕНИЕ](#_Toc72106850) 4

[1.1. Наименование программы](#_Toc72106851) 4

[1.2. Краткая характеристика области применения](#_Toc72106852) 4

[2. ОСНОВАНИЕ ДЛЯ РАЗРАБОТКИ](#_Toc72106853) 5

[2.1. Документы, на основании которых ведётся разработка](#_Toc72106854) 5

[2.2. Наименование темы разработки](#_Toc72106855) 5

[3. НАЗНАЧЕНИЕ РАЗРАБОТКИ](#_Toc72106856) 6

[3.1. Функциональное назначение](#_Toc72106857) 6

[3.2. Эксплуатационное назначение](#_Toc72106858) 6

[4. ТРЕБОВАНИЯ К ПРОГРАММЕ](#_Toc72106859) 7

[4.1. Требования к функциональным характеристикам](#_Toc72106860) 7

[4.1.1. Требования к составу выполняемых функций](#_Toc72106861) 7

[4.1.2. Требования к организации входных данных](#_Toc72106862) 8

[4.1.3. Требования к организации выходных данных](#_Toc72106863) 8

[4.2. Требования к интерфейсу](#_Toc72106864) 9

[4.3. Требования к надежности](#_Toc72106865) 9

[4.3.1. Требования к обеспечению надежного (устойчивого) функционирования программы](#_Toc72106866) 9

[4.3.2. Время восстановления после отказа](#_Toc72106867) 9

[4.3.3. Отказы из-за некорректных действий оператора](#_Toc72106868) 9

[4.4. Условия эксплуатации](#_Toc72106869) 10

[4.5. Требования к составу и параметрам технических средств](#_Toc72106870) 10

[4.6. Требования к информационной и программной совместимости](#_Toc72106871) 10

[4.6.1. Требования к информационным структурам и методам решения](#_Toc72106872) 10

[4.6.2. Требования к программным средствам, используемым программой](#_Toc72106873) 10

[4.6.3. Требования к исходным кодам и языкам программирования](#_Toc72106874) 10

[4.7. Требования к маркировке и упаковке](#_Toc72106875) 10

[4.8. Требования к транспортировке и хранению](#_Toc72106876) 11

[4.8.1. Требования к транспортировке и хранению программных документов, предоставленных в электронном виде 1](#_Toc72106877)1

[4.8.2. Требования к транспортировке и хранению программных документов, представленных в печатном виде 1](#_Toc72106878)1

[5. ТРЕБОВАНИЯ К ПРОГРАММНОЙ ДОКУМЕНТАЦИИ 1](#_Toc72106879)2

[5.1. Предварительный состав программной документации 1](#_Toc72106880)2

[5.2. Специальные требования к программной документации 1](#_Toc72106881)2

[6. ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ](#_Toc72106882) 13

[6.1. Ориентировочная экономическая эффективность 1](#_Toc72106883)3

[6.2. Предполагаемая потребность 1](#_Toc72106884)3

1. [6.3. Экономические преимущества разработки по сравнению с отечественными или зарубежными аналогами 1](#_Toc72106885)3

[7. СТАДИИ И ЭТАПЫ РАЗРАБОТКИ](#_Toc72106886) 15

[7.1. Необходимые стадии разработки, этапы и содержание работ](#_Toc72106887) 15

[7.2. Сроки разработки и исполнители](#_Toc72106888) 16

[8. ПОРЯДОК КОНТРОЛЯ И ПРИЁМКИ](#_Toc72106889) 17

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ](#_Toc72106890) 18

[ПРИЛОЖЕНИЕ 1……………………………………………………………………………….](#_Toc72106770)20

ПРИЛОЖЕНИЕ 2……………………………………………………………………………….27

# 1. ВВЕДЕНИЕ

## 1.1. Наименование программы

Наименование программы – «Высокопроизводительный Настраиваемый HTTP Сервер» («High Perfomance Customizable HTTP Server»).

## 1.2. Краткая характеристика области применения

Данный IT продукт представляет из себя высокоуровневую C++ библиотеку, которая дает возможность быстро проектировать и разворачивать REST API на языке С++, минуя такие низкоуровневые детали, как сокеты, потоки, контексты и т.д.

# 2. ОСНОВАНИЕ ДЛЯ РАЗРАБОТКИ

**2.1. Документы, на основании которых ведётся разработка**

Основанием для разработки является учебный план подготовки бакалавров по направлению 09.03.04 «Программная инженерия» и утвержденная академическим руководителем тема курсового проекта».

**2.2. Наименование темы разработки**

Наименование темы разработки – «Высокопроизводительный Настраиваемый HTTP Сервер».

Программа выполняется в рамках темы курсового проекта — «Высокопроизводительный Настраиваемый HTTP Сервер», в соответствии с учебным планом подготовки бакалавров по направлению 09.03.04 «Программная инженерия».

# 3. НАЗНАЧЕНИЕ РАЗРАБОТКИ

## 3.1. Функциональное назначение

Функциональным назначением программы является предоставление программисту возможности быстро и удобно проектировать REST API на языке С++, не погружаясь в детали реализации REST API. Данный продукт является отличным решением для команд разработки, пишущих на С++, так как позволит им не менять стек разработки, в том числе и ЯП, при появлении требования в необходимости наличия REST API для каких бы то ни было целей.

## 3.2. Эксплуатационное назначение

Многие существующие IT продукты написаны на С++. Этот ЯП славится своей производительностью и универсальностью, однако многие более новые языки (например, Джава, С#, Го) позволяют разрабатывать такие же продукты в разы быстрее.

Зачастую командам невозможно поменять стек разработки по разным причинам (например, слишком большое наследние, функционал которого нельзя перенести на современный технологический стек с точки зрения бизнес-value и затраченного на разработку времени). Чтобы не оказаться вне рынка с текущим «устаревшим» продуктом, его разработчикам приходится имплементировать современные features на старом технологическом стеке. Для одной из таких features, а именно REST API, которое присутствует во многих промышленных системах, и предназначена данная библиотека. Она призвана значительно сокращать время на разработку REST API и добиваться наибольшего значения соотношения «бизнес-value / затраченное на разработку время».

Для подключения данной библиотеки достаточно импортировать один заголовочный файл, который, в свою очередь, будет подключать другие заголовочные файлы. Настраивать зависимости будет система автоматизации сборки проектов CMake. Такой способ подключения является наиболее современным и простым, и именно поэтому многие разработчики придерживаются такого подхода при написании собственных библиотек, фреймворков и модулей.

# 4. ТРЕБОВАНИЯ К ПРОГРАММЕ

## 4.1. Требования к функциональным характеристикам

### 4.1.1. Требования к составу выполняемых функций

Программа должна давать пользователю возможность выполнять следующие функции:

* инстанцировать объект http сервера
* базово конфигурировать http сервер
* наследовать класс http сервера под свои нужды
* создавать status line http ответа из готовых шаблонов: определять методы GET и POST, код возвращаемого значения
* создавать заголовки http ответа из готовых шаблонов: content-type, content-length и т. д.
* создавать тела http ответа из готовых шаблонов, отдельных html файлов
* настраивать кастомное логирование с разными уровнями в отдельный файл
* настраивать кастомное логирование с разными уровнями в syslog
* кешировать http ответы
* обрабатывать ошибки
* читать комментарии в коде сервера, которых будет достаточно для использования всех возможностей сервера
* подключать сервер через .hpp файл с отдельной папкой (где будут все остальные файлы-зависимости лежать), настройка необходимых зависимостей идет через готовый CmakeLists.txt

### 4.1.2. Требования к организации входных данных

Входные данные — это исходный код на С++, в который портируется http сервер (исходный код может быть представлен в файлах с расширениями .cpp, .h, .hpp). В качестве системы автоматизации сборки проекта рекомендуется использовать CMake, так как в таком случае будет намного проще настроить зависимости, необходимые для подключаемого http сервера. Исходный код проекта до подключения данного фреймворка должен компилироваться успешно и проект должен собираться корректно.

В свою очередь, после подключения фреймворка (после успешного подключения всех необходимых для его работы файлов и успешной настройки необходимых зависимостей) в исходном коде создается объект http сервера. При проектировании REST API для создания очередного эндпоинта необходимо воспользоваться соответствующим методом созданного http сервера.

Касательно требований к входным данным, программисту необходимо ознакомиться с внутренней справкой / документацией http сервера, которая исчерпывающе описывает, как с помощью него проектировать REST API.

### 4.1.3. Требования к организации выходных данных

Результат работы сервера должен представлять собой действующий REST API, а также файлы с логированием. Отследить корректность работы можно с помощью логов, настроенных на максимально возможный уровень — DEBUG 5, а также с помощью непосредственно функционального тестирования написанного REST API. Если в логах была обнаружена хоть одна ошибка, сервер имеет неопределенное поведение.

## 4.2. Требования к интерфейсу

Графический интерфейс у данного сервера фактически отсутствует, так как все команды прописываются именно в исходном файле.

**4.3. Требования к надежности**

### 4.3.1. Требования к обеспечению надежного (устойчивого) функционирования программы

Для устойчивой работы программы необходимо соблюдать ряд организационно-технических мер:

1) При компиляции тестируемой программы необходимо включить следующие флаги: -fsanitize=address,undefined -fno-sanitize-recover=all -Wall -Wextra -Werror -std=c++14 -pedantic; Компилировать необходимо компилятором gcc версии не ниже 14 или компилятором clang версии не ниже 3.4;

2) Иметь правильно скомпилированные и находящиеся в нужном месте библиотеки, который использует данный ;

3) Компиляция исходного кода должна производиться с флагами оптимизации (-O2, -O3 или -Ofast);

**4.3.2. Время восстановления после отказа**

Если отказ был спровоцирован внешними факторами (например, поломка энергоблока компьютера или неисправность других его внутренних компонентов), то время исправления ситуации не регламентируется.

Если отказ был спровоцирован внутренними факторами (например, пользователь случайно удалил системный файл и ОС теперь работает некорректно), то время восстановления не должно быть больше времени, необходимого для исправления ошибки с ОС.

### 4.3.3. Отказы из-за некорректных действий оператора

Отказ программы возможен также вследствие некорректных действий пользователя при неправильном использовании (например, исходный тестируемый код отрабатывает с ошибкой или предупреждением или в runtime возникло неопределенное поведение). Чтобы такого не допускать, необходимо ознакомиться с пунктом 4.3.1;

Также отказ возможен при некорректном пользовании операционной системой. В таком случае время на восстановления сервера не должно превышать времени, необходимого для устранения поломки ОС.

**4.4. Условия эксплуатации**

Компьютер предназначен для эксплуатации в помещениях с искусственно-регулируемыми климатическими условиями, например, в отапливаемых и вентилируемых помещениях категории 4.1 согласно ГОСТ 15150-69 [4].

Программа не требует специального обслуживания.

Программа может быть использована как одним человеком, так и группой лиц. Необходимая квалификация – пользователь (ознакомившийся с краткой справкой сервера).

## 4.5. Требования к составу и параметрам технических средств

Для бесперебойной работы программного продукта требуется компьютер с:

* установленной версией компилятора gcc - 14, clang — 3.4
* операционной системой со стабильной сборкой, выпущенной не позднее 2015 года
* объемом свободной встроенной памяти не меньше 55 МБ,
* объёмом оперативной памяти не меньше 1 ГБ.

## 4.6. Требования к информационной и программной совместимости

### 4.6.1. Требования к информационным структурам и методам решения

Требования к информационным структурам и методам решения не предъявляются.

### 4.6.2. Требования к программным средствам, используемым программой

Для работы программного продукта требуется gcc компилятор версии не ниже 14 или clang компилятор версии не ниже 3.4; необходимые флаги см. в пункте 4.3.1

**4.6.3. Требования к исходным кодам и языкам программирования**

Программа должна быть написана на языке программирования С++ версии не выше 14. В качестве среды разработки программы может быть использован любой редактор кода. Допускается писать код только в .cpp, .h и .hpp файлах.

**4.7. Требования к маркировке и упаковке**

Требования к маркировке и упаковке не предъявляются.

**4.8. Требования к транспортировке и хранению**

**4.8.1. Требования к транспортировке и хранению программных документов, предоставленных в электронном виде**

Программные документы загружаются в электронном виде в информационную образовательную среду LMS (Learning Management System) НИУ ВШЭ. Требования к хранению и транспортировке не предъявляются.

**4.8.2. Требования к транспортировке и хранению программных документов, представленных в печатном виде**

Программные документы, предоставляемые в печатном виде, должны соответствовать общим правилам учета и хранения программных документов, предусмотренных стандартами Единой системы программной документации и соответствовать требованиям ГОСТ 19.602-78 [13].

# 5. ТРЕБОВАНИЯ К ПРОГРАММНОЙ ДОКУМЕНТАЦИИ

## 5.1. Предварительный состав программной документации

1. «Высокопроизводительный Настраиваемый HTTP Сервер». Техническое задание (ГОСТ 19.201-78 [8])
2. «Высокопроизводительный Настраиваемый HTTP Сервер». Программа и методика испытаний (ГОСТ 19.301-79 [9])
3. «Высокопроизводительный Настраиваемый HTTP Сервер». Текст программы (ГОСТ 19.401-78 [10])
4. «Высокопроизводительный Настраиваемый HTTP Сервер». Пояснительная записка (ГОСТ 19.404-79 [11])
5. «Высокопроизводительный Настраиваемый HTTP Сервер». Руководство оператора (ГОСТ 19.505-79 [12])

## 5.2. Специальные требования к программной документации

1. Все документы к программе должны быть выполнены в соответствии с ГОСТ 19.106-78 [7] и ГОСТами к каждому виду документа (см. п. 5.1.);
2. Пояснительная записка должна быть загружена в систему Антиплагиат через LMS (Learning Management System) НИУ ВШЭ.
3. Вся документация и программа также сдаются в электронном виде в формате .pdf или .docx. в архиве формата .rar или .zip.
4. За три дня до защиты комиссии все материалы курсового проекта:

* техническая документация,
* программный проект,
* исполняемый файл,
* отзыв руководителя,
* лист Антиплагиата

должны быть загружены одним или несколькими архивами в проект дисциплины «Курсовой проект, 3 курс ПИ» в личном кабинете в информационной образовательной среде LMS (Learning Management System) НИУ ВШЭ.

# 6. ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ

## 6.1. Ориентировочная экономическая эффективность

В рамках данной работы расчет экономической эффективности не предусмотрен.

## 6.2. Предполагаемая потребность

Данный сервер могут использовать все разработчики с компилятором gcc версии не ниже 14 или компилятором clang версии не ниже 3.4, которым нужно быстро добавить REST API в свою программу. Данный сервер предлагает простое, быстрое, легковесное и одновременно высокопроизводительное решение данной проблемы, упрощая жизнь разработчикам.

## 6.3. Экономические преимущества разработки по сравнению с отечественными или зарубежными аналогами

На момент создания программы наиболее используемыми аналогами в области http серверов-микрофреймворков являются: Crow, Pistache, Beast.

Общий недостаток всех этих продуктов — недостаточный уровень абстракции для использования их возможностей в условиях ограниченных временных ресурсов. Данные фреймворки предоставляют более сложный интерфейс для создания REST API, нежели текущий http сервер. В итоге в большинстве случаев они требуют больше времени для имплементации той же функциональности, которую предлагает мой сервис.

Если рассматривать найденных «конкурентов» по отдельности, то можно выявить следующие особенности.

Pistache:

1. Сложность и документация: начальная настройка и использование Pistache может потребовать много времени из-за относительно сложной структуры и ограниченной документации.

2. Ограниченные возможности масштабирования: хотя Pistache предлагает хорошую производительность, в некоторых случаях могут возникать ограничения в масштабировании и обработке больших нагрузок, особенно в сравнении с другими более распространенными фреймворками.

Crow:

1. Ограничения масштабируемости: В некоторых случаях Crow может иметь ограничения по масштабируемости и производительности в сравнении с иными фреймворками.

2. Отсутствие полной стандартной поддержки: иногда Crow может не поддерживать все стандарты и спецификации, которые могут потребоваться для конкретных задач, требуя дополнительной настройки и расширений.

Beast:

1. Сложность использования: Beast является частью библиотеки Boost, которая является довольно низкоуровневой абстракцией для работы с сетью. С помощью этого инструмента будет сложно разрабатывать REST API «с нуля».

2. Обширная документация: из-за того, что Beast является низкоуровневой основой для построения REST API, то и документация его значительно больше других рассмотренных REST API решений.

Подытожив, можно сказать, что такие инструменты разработки, как данный фреймворк, являются передовыми средствами разработки на современном С++, так как имеют достаточный уровень абстракции, что позволяет разработчикам данного языка повысить свой perfomance, затрачивая существенно меньше времени на создание таких популярных решений, как REST API.

# 7. СТАДИИ И ЭТАПЫ РАЗРАБОТКИ

## 7.1. Необходимые стадии разработки, этапы и содержание работ

Стадии и этапы разработки были выявлены с учетом ГОСТ 19.102-77 [6]:

Таблица 1 – Стадии разработки, этапы и содержание работ

|  |  |  |
| --- | --- | --- |
| **Стадии разработки** | **Этапы работ** | **Содержание работ** |
| I.Техническое задание | Обоснование необходимости разработки программы | Постановка задачи |
| Сбор исходных материалов |
| Выбор и обоснование критериев эффективности и качества разрабатываемой программы |
| Разработка и утверждение технического задания | Определение требований к программе |
| Определение стадий, этапов и сроков разработки программы и документации на нее |
| Определение необходимости проведения научно-исследовательских работ на последующих стадиях |
| Согласование и утверждение технического задания |
| II.Рабочий проект | Разработка программы | Программирование и отладка программы |
| Разработка программной документации | Разработка программных документов в соответствии с требованиями ГОСТ 19.101-77 [5] |
| Испытания программы | Разработка, согласование и утверждение порядка и методики испытаний |
| Проведение предварительных испытаний |
| Корректировка программы и программной документации по результатам испытаний |

Продолжение Таблицы 1

|  |  |  |
| --- | --- | --- |
| **Стадии разработки** | **Этапы работ** | **Содержание работ** |
| III. Внедрение | Подготовка и защита программного продукта. | Утверждение даты защиты программного продукта. |
| Подготовка программы и программной документации для презентации и защиты. |
| Представление разработанного программного продукта руководителю и получение отзыва. |
| Загрузка Пояснительной записки в систему Антиплагиат через LMS (Learning Management System) НИУ ВШЭ |
| Загрузка материалов курсового проекта в LMS (Learning Management System) НИУ ВШЭ, проект дисциплины «Курсовой проект, 3 курс ПИ» (см. п. 5.2) |
| Защита программного продукта (курсового проекта) комиссии. |

## 7.2. Сроки разработки и исполнители

Разработка должна закончиться к 25 апреля 2024 года.

Исполнитель: Фортов Егор Кириллович, студент группы БПИ214 факультета компьютерных наук НИУ ВШЭ.

# 8. ПОРЯДОК КОНТРОЛЯ И ПРИЁМКИ

Проверка программного продукта, в том числе и на соответствие техническому заданию, осуществляется заказчиком совместно с исполнителем согласно «Программе и методике испытаний», а также пункту 5.2

Защита выполненного проекта осуществляется комиссии, состоящей из преподавателей департамента программной инженерии, в утверждённые приказом декана ФКН сроки.

**СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ**

1. Pistache — REST API framework [Электронный ресурс] Режим доступа: https://github.com/pistacheio/pistache, свободный. (дата обращения: 15.02.2024)
2. Crow – REST API framework [Электронный ресурс] Режим доступа: https://github.com/CrowCpp/Crow, свободный. (дата обращения: 15.02.2024)
3. Beast - REST API framework [Электронный ресурс] Режим доступа: <https://www.boost.org/doc/libs/master/libs/beast/doc/html/index.html>, свободный. (дата обращения: 15.02.2024)
4. ГОСТ 15150-69 Машины, приборы и другие технические изделия. Исполнения для различных климатических районов. Категории, условия эксплуатации, хранения и транспортирования в части воздействия климатических факторов внешней среды. – М.: Изд-во стандартов, 1997.
5. ГОСТ 19.101-77 Виды программ и программных документов. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
6. ГОСТ 19.102-77 Стадии разработки. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
7. ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
8. ГОСТ 19.201-78 Техническое задание. Требования к содержанию и оформлению. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
9. ГОСТ 19.301-79 Программа и методика испытаний. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
10. ГОСТ 19.401-78 Текст программы. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
11. ГОСТ 19.404-79 Пояснительная записка. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
12. ГОСТ 19.505-79 Руководство оператора. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
13. ГОСТ 19.602-78 Правила дублирования, учета и хранения программных документов, выполненных печатным способом. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
14. Статья про REST API в целом [Электронный ресурс]. Режим доступа: https://habr.com/ru/articles/483202/, свободный. (дата обращения: 15.02.2024)
15. Статья про REST API в целом [Электронный ресурс]. Режим доступа: https://aws.amazon.com/ru/what-is/restful-api/, свободный. (дата обращения: 15.02.2024)
16. Статья про REST API в целом [Электронный ресурс]. Режим доступа: https://blog.skillfactory.ru/glossary/rest-api/, свободный. (дата обращения: 15.02.2024)
17. Статья про REST API [Электронный ресурс] / Wikipedia. Режим доступа: https://ru.wikipedia.org/wiki/REST, свободный. (дата обращения: 15.02.2024)
18. Статья RESTful APIs [Электронный ресурс]. Режим доступа: [https://www.astera.com/type/blog/rest-api-definition/](https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Testing/Profiling), свободный. (дата обращения: 15.02.2024)
19. Видео про REST API [Электронный ресурс]. Режим доступа: https://www.youtube.com/watch?v=-mN3VyJuCjM свободный. (дата обращения: 15.02.2024)
20. Статья IBM про REST API [Электронный ресурс] / IBM; Режим доступа: https://www.ibm.com/topics/rest-apis свободный. (дата обращения: 15.02.2024)

# ПРИЛОЖЕНИЕ 1

**ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ КЛАССОВ**

Таблица 1.1 – Описание и функциональное назначение классов/структур в файле ServeMe.hpp

|  |  |
| --- | --- |
| **Класс** | **Назначение** |
| Level | Класс необходим для категоризации уровней логирования. Доступные уровни: Debug, Info, Warning, Error, Critical. |
| Method | Класс необходим для категоризации методов HTTP-запросов. Доступные методы: GET, POST. |
| HttpServerInterface | Класс представляет собой интерфейс для любого объекта, который претендует на роль HttpServer-а. |
| LoggerInterface | Класс представляет собой интерфейс для любого объекта, который претендует на роль Logger-а. |
| HttpSessionInterface | Класс представляет собой интерфейс для любого объекта, который претендует на роль HttpSession. |
| RESTAPIAPPInterface | Класс представляет собой интерфейс для любого объекта, который претендует на роль RESTAPIAPP – входной точки фреймворка. |
| Logger | Класс предоставляет возможность логирования в файл и внутренний лог ОС с учетом текущего времени и установленного уровня логирования. Класс реализует интерфейс LoggerInterface. |
| HttpSession | Класс нужен для реализации http сервера. Отвечает за принятие http-запросов и отправку на них ответов. Класс реализует интейрфейс HttpSessionInterface. |
| HttpServer | Класс является главным в данном фреймворке. Отвечает за создание сокета, открытия его, добавления новых эндпоинтов и прослушивания соединений по протоколу TCP. Класс реализует интерфейс HttpServerInterface. |
| RESTAPIAPP | Класс является входной точкой подприложения. Предоставляет возможность создать сам HTTP сервер, запустить его, добавить в него обработку новых эндпоинтов и завершить его. Класс реализует интерфейс RESTAPIAPPInterface. |

# Также в добавок к краткому описанию функционального назначения классов приведу ответы на FAQ:

Вопрос: Зачем нужны разные уровни логирования? (класс Level)

Ответ:

Различные уровни логирования важны для того, чтобы обеспечить гибкость и контроль над сообщениями, которые записываются в логи. Каждый уровень логирования представляет собой определенный уровень важности сообщений, их значимость и контекст использования, что позволяет программистам и системным администраторам выбирать, какие сообщения нужно сохранять в логах в зависимости от ситуации.

Преимущества разных уровней логирования:

1. Отладочный (Debug): Используется для вывода детальной отладочной информации о процессе работы программы. Этот уровень полезен для выявления ошибок и анализа процессов.

2. Информационный (Info): Логирование основной информации о работе программы, такой как старт, остановка, ключевые события. Эти сообщения полезны для мониторинга работы приложения.

3. Предупреждения (Warning): Для сообщений о потенциальных проблемах или исключительных ситуациях, которые не критичны, но требуют внимания.

4. Ошибки (Error): Логирование ошибок и исключений, которые не позволяют программе работать правильно.

5. Фатальные ошибки (Fatal): Для критических ошибок, которые приводят к аварийному завершению работы программы.

Использование разных уровней логирования позволяет эффективно управлять объемом информации в логах, обеспечить быструю диагностику проблем, анализировать работу программы и обеспечивать необходимую информацию для поддержки и отладки.

Вопрос: Зачем нужен отдельный класс для типов http запросов?

Ответ:

Использование отдельного класса для типов HTTP запросов может иметь несколько преимуществ:

1. Чистый и понятный код: Отдельный класс для типов HTTP запросов помогает упростить код и сделать его более читаемым и понятным. Это улучшает поддерживаемость кода и облегчает работу с HTTP запросами.

2. Увеличение переиспользуемости: Класс для типов HTTP запросов может содержать члены данных и методы, специфичные для запросов, что упрощает их использование и повторное использование в различных частях приложения.

3. Улучшенная безопасность: Использование класса может способствовать контролю за данными, передаваемыми в запросах, и обеспечивать определенный уровень проверки корректности данных, что способствует улучшению безопасности приложения.

4. Поддержка модели объектно-ориентированного программирования: Отдельный класс для типов HTTP запросов соответствует принципам ООП, таким как инкапсуляция, наследование и полиморфизм, что упрощает разработку, тестирование и поддержку кода.

5. Расширяемость: Класс для типов HTTP запросов может быть легко расширен и дополнен дополнительными функциями и свойствами, что упрощает добавление нового функционала в рамках работы с HTTP запросами.

Таким образом, использование отдельного класса для типов HTTP запросов может улучшить структуру и организацию кода, обеспечить повышенную переиспользуемость, безопасность и поддерживаемость приложения.

Вопрос: Зачем нужны отдельные интерфейсы для каждого класса?

Ответ:

Использование отдельных интерфейсов для каждого класса позволяет создавать абстрактные контракты, определяющие набор методов, которые класс должен реализовать. Это способствует разделению интерфейса и реализации, облегчает взаимодействие между компонентами программы и обеспечивает гибкость при добавлении новых классов.

Основные преимущества:

1. Разделение интерфейса и реализации: Интерфейсы определяют только методы, не затрагивая их внутреннюю реализацию. Это упрощает изменения в реализации классов, не затрагивая их внешний интерфейс, что является важным аспектом при разработке масштабируемых приложений.

2. Улучшенная читаемость и понимание кода: Использование интерфейсов делает код более читаемым и легким для понимания. Разработчики могут предварительно оценить, какие методы предоставляют классы, без необходимости изучения их реализации.

3. Поддержка множественного наследования: Интерфейсы позволяют классу реализовать несколько контрактов одновременно, обеспечивая гибкость и возможность работать с разными типами объектов через общие интерфейсы.

4. Поддержка полиморфизма: Использование интерфейсов упрощает использование полиморфизма, что позволяет передавать и хранить различные типы объектов, реализующих общий интерфейс, в общих коллекциях.

Таким образом, создание отдельных интерфейсов для каждого класса способствует улучшению гибкости, читаемости и архитектуры программы, что способствует разработке качественного и легко поддерживаемого кода.

Вопрос: Зачем нужен отдельный логгер?

Ответ:

Отдельный логгер - это специальный компонент программы, предназначенный для записи информации о работе приложения в журнал событий. Использование отдельного логгера позволяет разделить процессы ведения журнала и бизнес-логики приложения, что обеспечивает ряд преимуществ:

1. Четкая раздельность обязанностей: Логгер отвечает исключительно за регистрацию событий и ошибок, позволяя другим компонентам приложения сконцентрироваться на своей основной функциональности. Это способствует улучшению модульности и структурированности приложения.

2. Гибкая настройка: Отдельный логгер позволяет поддерживать разные уровни важности событий, фильтрацию по категориям, выбор метода вывода информации (например, в консоль, файл, базу данных), что обеспечивает гибкую настройку журнала под нужды разработчиков и системных администраторов.

3. Улучшенная отладка: Наличие отдельного логгера упрощает процесс отладки кода, поскольку он позволяет выводить разнообразную информацию о ходе выполнения программы, включая сообщения об ошибках, предупреждения и детали выполнения различных операций.

4. Удобство мониторинга и анализа: Благодаря логгеру можно вести подробный мониторинг работы приложения, создавать отчёты о произошедших событиях, анализировать поведение программы в различных сценариях и улучшать качество приложения.

Таким образом, использование отдельного логгера является важной частью современного программного обеспечения, способствуя улучшению разделения обязанностей, отладки и мониторинга, а также обеспечивая гибкую настройку и анализ работы приложения.

Вопрос: зачем нужен отдельный класс httpSession?

Ответ:

Класс HttpSession используется для управления сеансом связи между HTTP-сервером и клиентом. Он обычно содержит логику для обработки входящих HTTP-запросов, отправки HTTP-ответов, управления состоянием сеанса и взаимодействия с клиентскими запросами.

Отдельный класс HttpSession является разумным выбором, так как он позволяет отделить логику работы с конкретным сеансом связи от других частей серверной системы. Это способствует модульности и улучшает читаемость и поддерживаемость кода. Кроме того, класс HttpSession можно настроить для управления сеансами, аутентификации, авторизации и другими аспектами безопасности, что обеспечивает безопасность и надежность всей системы.

Вопрос: зачем разделять rest api на классы server и session?

Ответ:

Разделение REST API на классы Server и Session позволяет улучшить модульность, читаемость и поддерживаемость кода, а также облегчает расширение и тестирование функциональности.

Класс Server обычно отвечает за обработку входящих HTTP-запросов, маршрутизацию запросов к соответствующим методам и управление жизненным циклом сервера в целом. Он предоставляет интерфейс для настройки сервера, добавления маршрутов, управления подключениями и других аспектов общей логики сервера.

Класс Session, с другой стороны, фокусируется на управлении индивидуальными сеансами связи между сервером и клиентом. Он обрабатывает входящие запросы, управляет состоянием сеанса, включает в себя механизмы аутентификации и авторизации, а также работает с конкретными данными и контекстом, связанным с текущим запросом.

Разделяя ответственности между классами Server и Session, мы получаем возможность более ясного определения обязанностей различных частей нашего REST API. Это снижает связность, упрощает распределение задач и позволяет каждому классу фокусироваться на своей конкретной роли, делая код более ясным, удобным для поддержки и развития.

Вопрос: зачем выделяется отдельный класс для запуска приложения?

Ответ:

Выделение отдельного класса для запуска приложения может иметь несколько преимуществ, включая:

1. Ясность и модульность: Разделяя логику инициализации и запуска приложения в отдельный класс, вы повышаете ясность и модульность кода. Это обеспечивает лучшее разделение ответственности и упрощает понимание и поддержку программы.

2. Гибкость конфигурации: Отдельный класс для запуска приложения облегчает различные виды конфигурации и настройки, такие как загрузка настроек, инициализация ресурсов и запуск различных компонентов приложения.

3. Тестирование: Изоляция логики запуска приложения делает тестирование кода более простым. Это позволяет проводить модульное тестирование и мокирование без необходимости запуска всего приложения.

4. Расширение и поддержка: Выделение запуска в отдельный класс облегчает добавление новых функциональностей в процесс запуска приложения, а также обеспечивает легкость сопровождения приложения.

Таким образом, выделение отдельного класса для запуска приложения способствует улучшению архитектуры, гибкости, тестируемости и поддерживаемости приложения.

Вопрос: какие ключевые сущности есть у любого http сервера?

Ответ:

У любого HTTP сервера есть несколько ключевых сущностей, которые обеспечивают его функционирование:

1. Прослушиватель (Listener) - прослушивает указанный сетевой порт и ожидает входящие HTTP запросы от клиентов.

2. Маршруты (Routes) - определяют, какие запросы отправляются на какие обработчики (handler) в зависимости от URL и метода запроса (GET, POST, PUT, DELETE и т. д.).

3. Обработчики (Handlers) - логика для обработки конкретного HTTP запроса, и генерации соответствующего HTTP ответа. Обработчики могут также взаимодействовать с хранилищем данных или другими сервисами для выполнения бизнес-логики.

4. Контекст (Context) - содержит информацию о текущем HTTP запросе, такую как параметры запроса, заголовки, тело запроса и другие атрибуты, которые могут быть использованы обработчиками для принятия решений.

5. Логгеры (Loggers) - обеспечивают журналирование различных событий и действий, проводимых сервером, для отслеживания и мониторинга работы приложения.

Эти ключевые сущности помогают организовать работу HTTP-сервера, обеспечивая прием запросов от клиентов, их обработку и отправку соответствующих ответов, управление состоянием и данных запросов, а также обеспечивают безопасность и журналирование работы сервера.

Вопрос: какая стандартная архитектура фреймворка rest api?

Ответ:

Стандартная архитектура фреймворка REST API состоит из нескольких ключевых компонентов. В общем, архитектура RESTful API является базовым каркасом для создания веб-сервисов, который следует принципам REST, используя HTTP протокол в качестве основы для обработки запросов и отправки ответов. Основные компоненты включают:

1. Маршрутизация: Фреймворк обычно предоставляет механизм маршрутизации, который определяет, какие запросы отправляются на какие обработчики в зависимости от URL и метода HTTP запроса.

2. Обработчики (Controller/Handler): Обработчики представляют логику для обработки входящих HTTP запросов и формирования соответствующих ответов. Они обычно взаимодействуют с бизнес-логикой и доступом к данным для выполнения задач, определенных в API.

3. Модели данных (Data Models): Фреймворк может предоставлять средства для определения общих моделей данных, которые облегчают обработку входящих запросов и формирование ответов в нужном формате.

4. Middleware: Middleware представляет собой цепочку промежуточных обработчиков, которые выполняются перед обработкой основного запроса и позволяют выполнять такие задачи, как аутентификация, логгирование, обработка ошибок и другие операции.

5. Сервисы и Бизнес-логика: Фреймворк может предоставлять средства для организации и управления бизнес-логикой и сервисами, которые обрабатывают функциональные запросы API.

6. Контекст запроса (Request Context): Предоставляет доступ к информации о текущем HTTP запросе, такой как заголовки, параметры, аутентификационные данные, что позволяет обработчикам легко получить доступ к этой информации.

Эти компоненты обеспечивают базовую структуру и функциональность для построения RESTful API, чтобы упростить разработку, тестирование, поддержку и масштабируемость веб-сервисов.

Вопрос: почему методы get и post самые популярные?

Ответ:

Методы GET и POST являются самыми популярными методами HTTP из-за своего распространенного использования и специфических свойств, обусловленных их назначением:

1. GET: Метод GET используется для запроса данных ресурса с веб-сервера. Он широко используется для получения информации с сервера, включая HTML страницы, изображения, видео, данные в формате JSON и другие ресурсы. Метод GET прост в использовании и позволяет передавать ограниченное количество данных через URL-параметры, что делает его удобным для использования в адресной строке браузера и для создания гиперссылок.

2. POST: Метод POST используется для отправки данных на сервер для обработки. Он часто применяется при отправке форм и передаче сложных данных, таких как данные из формы авторизации, файлы, или большие объемы информации, не помещающиеся в URL. Метод POST также широко используется в AJAX-запросах и при создании, обновлении или удалении ресурсов.

Популярность методов GET и POST обусловлена их широким спектром применения, простотой использования и понимания, а также широкой поддержкой со стороны различных клиентов и серверов. Эти методы обеспечивают базовый набор функций для обеспечения взаимодействия между клиентами и серверами, что делает их неотъемлемой частью протокола HTTP.

Вопрос: Зачем нужен acceptor в boost asio?

Ответ:

В библиотеке Boost.Asio acceptor предназначен для прослушивания входящих соединений на серверной стороне в сетевом программировании.

Когда клиентское приложение пытается соединиться с сервером, сервер создает acceptor, который прослушивает указанный порт на входящие соединения. Когда входящее соединение установлено, acceptor создает новый сокет для взаимодействия с клиентом. Этот новый сокет передается дальше для обработки в рамках серверной программы.

Использование acceptor позволяет серверной части программы эффективно обрабатывать входящие соединения, управлять ими и создавать новые сокеты для взаимодействия с клиентами.

Пример с использованием acceptor в Boost.Asio:

// Создание и настройка io\_context и acceptor

boost::asio::io\_context io\_context;

tcp::endpoint endpoint(tcp::v4(), 8080);

tcp::acceptor acceptor(io\_context, endpoint);

// Принятие входящего соединения

tcp::socket socket(io\_context);

acceptor.accept(socket);

// Теперь можно взаимодействовать с новым сокетом "socket"

В этом примере acceptor настраивается для прослушивания входящих соединений на порту 8080, и когда такое соединение устанавливается, создается новый сокет socket для взаимодействия с клиентом.

Вопрос: что такое io в boost?

Ответ:

В библиотеке Boost.Asio термин "io" обычно относится к "io\_context". `io\_context` - это центральный компонент многих операций ввода-вывода (I/O) в Boost.Asio и играет важную роль в организации асинхронных операций ввода-вывода.

`io\_context` представляет собой общий интерфейс для управления операциями ввода-вывода, такими как сетевые операции, операции файлового ввода-вывода и другие асинхронные операции ввода-вывода. Он обеспечивает асинхронное и многозадачное взаимодействие с различными ресурсами (например, сокетами) в программе.

`io\_context` управляет циклом обработки событий (event loop) и позволяет организовывать асинхронные операции ввода-вывода, отслеживать состояние операций и управлять потоками выполнения для обработки асинхронных событий.

Основные операции связанные с `io\_context` включают:

- Организация асинхронных операций ввода-вывода.

- Управление потоками выполнения для обработки асинхронных операций.

- Централизация обработки и отслеживания асинхронных событий.

В целом, `io\_context` представляют собой ключевой компонент для организации асинхронных операций ввода-вывода в Boost.Asio, используемый для управления и координации асинхронных конструкций в сетевом программировании.

Вопрос: Как boost реализует асинхронность под капотом?

Ответ:

Boost.Asio, основанный на стандарте Networking Technical Specification (NTS) и представленный в стандартной библиотеке C++ в версии 23, предоставляет удобные средства для работы с сетевым вводом-выводом (I/O) и асинхронными операциями. Реализация асинхронности Boost.Asio строится на основе многозадачной, неблокирующей модели и событийно-ориентированного программирования.

Ниже приведены основные аспекты реализации асинхронности в Boost.Asio:

1. Использование системных вызовов: Boost.Asio использует системные вызовы операционной системы для реализации асинхронных операций ввода-вывода, таких как чтение/запись из сокетов.

2. Многозадачность: Boost.Asio использует многозадачные аспекты среды выполнения языка C++, включая многопоточность и кроссплатформенные механизмы асинхронного ввода-вывода, такие как epoll в Linux, kqueue в BSD и IOCP в Windows.

3. Callstack & Event-driven: Реализация включает в себя использование функций обратного вызова (callback) и цикла обработки событий, который позволяет обрабатывать асинхронные операции при возникновении событий.

4. Декомпозиция: Boost.Asio разделяет асинхронные операции на более низкоуровневые конструкции, позволяя разработчику управлять асинхронными операциями на более детальном уровне и определять свою собственную многозадачность.

Как следствие, реализация асинхронности в Boost.Asio использует широкий спектр средств и подходов для обеспечения эффективного и портируемого асинхронного ввода-вывода в C++ при работе с сетевым программированием.

Вопрос: зачем нужна многопоточность при работе с сетью?

Ответ:

Многопоточность играет важную роль в разработке сетевых приложений по нескольким причинам:

1. Параллелизм:

- Работа сетевых приложений, особенно серверных, часто предполагает одновременное обслуживание нескольких клиентов. Запуск каждого соединения в отдельном потоке позволяет обрабатывать запросы клиентов параллельно, что повышает производительность и улучшает отзывчивость приложения.

2. Отдельные потоки для ввода/вывода:

- Ввод-вывод в сетевых приложениях может быть блокирующим, особенно когда происходит медленная операция, такая как чтение/запись большого объема данных из/в сеть. Использование отдельных потоков для ввода/вывода позволяет избежать блокировки основного потока и продолжать обработку других запросов.

3. Управление состоянием соединений:

- Многопоточность может использоваться для управления состоянием соединений, таким как мониторинг активности клиентов, обновление данных, запуск и остановка обработки запросов и т.д.

4. Распределенные вычисления:

- Многопоточность может использоваться для выполнения распределенных вычислений, обработки данных и выполнения других задач, связанных с обработкой и обменом информацией.

5. Улучшение отказоустойчивости:

- Потоки могут использоваться для создания резервных копий, контроля за состоянием сервера, обнаружения и устранения ошибок, что способствует повышению отказоустойчивости сетевого приложения.

Таким образом, использование многопоточности в сетевых приложениях позволяет эффективно управлять ресурсами, повысить производительность, улучшить отзывчивость системы и уместно обрабатывать параллельные задачи.

Вопрос: Зачем в приницпе применять многопоточность?

Ответ:

Применение многопоточности в программировании предлагает несколько значимых преимуществ и предоставляет ряд возможностей для улучшения производительности, отзывчивости и распределенности приложений. Вот некоторые причины, почему многопоточность часто применяется:

1. Параллелизм: Многопоточность позволяет выполнять несколько частей программы параллельно, что может значительно ускорить обработку данных и выполнение вычислений.

2. Улучшение отзывчивости: Работа с многозадачностью позволяет отвечать на одновременные запросы пользователя в реальном времени, предотвращая блокировку интерфейса пользователя и улучшая пользовательский опыт.

3. Использование многопроцессорных систем: Многопоточность позволяет программам более полно использовать ресурсы многопроцессорных систем и многоядерных процессоров.

4. Асинхронное программирование: Многопоточность позволяет эффективно обрабатывать асинхронные операции ввода-вывода и сетевого взаимодействия, увеличивая отзывчивость приложений.

5. Разделение задач: Разделение задач и обработка каждой из них в отдельном потоке может упростить разработку и повысить модульность программного обеспечения.

6. Улучшение производительности обработки данных: Многопоточность позволяет обрабатывать большие объемы данных или выполнить вычисления, распределяя их между несколькими потоками, что повышает производительность и эффективность вычислений.

7. Работа с пользователями: Многопоточные приложения могут эффективно управлять множеством пользователей, не блокируя поток выполнения на обработке одного задания.

В целом, многопоточность представляет собой мощный инструмент для повышения производительности, улучшения отзывчивости приложений и распределения задач, что обеспечивает более эффективное использование ресурсов компьютера.

Вопрос: зачем писать кастомные реализации фреймворков? В чем benefit?

Ответ:

Написание кастомных реализаций фреймворков или библиотек может приносить несколько преимуществ и принести ценность в различных областях, включая:

1. Специализированные требования: Подход, разработанный для удовлетворения специальных требований и особенностей проекта, может лучше соответствовать уникальным потребностям разработчика или конкретного проекта. Это позволяет создавать оптимизированные, адаптированные к специфике бизнеса или технических особенностей фреймворки.

2. Гибкость и контроль: Создание кастомных реализаций позволяет разработчику полностью контролировать архитектуру, интерфейсы и функциональность фреймворка, обеспечивая гибкость, отсутствие избыточности и настройку под конкретные требования.

3. Обучение и развитие: Разработка своих реализаций может служить ценной практикой для развития навыков программирования и глубокого понимания архитектуры систем и платформы, способствуя лучшему пониманию технологий и концепций разработки.

4. Интеграция и совместимость: Кастомные реализации могут лучше сочетаться и интегрироваться с другими компонентами и средами разработки, такими как существующие системы, инфраструктура и технологические стеки.

5. Инновации и эксперименты: Разработка собственных фреймворков позволяет экспериментировать с новыми идеями и подходами, вносить инновации и пытаться улучшить существующие решения, что может привести к новым открытиям и улучшениям в решении задач.

Однако, несмотря на потенциальные выгоды, стоит учитывать, что написание кастомных реализаций также может требовать значительных усилий, времени и ресурсов, особенно в случае, если существующие решения способны эффективно решить поставленную задачу. Прежде чем приступать к разработке кастомных реализаций, важно проанализировать риски и выгоды данного подхода.

Вопрос: на каких языках можно сделать rest api?

Ответ:

RESTful API могут быть реализованы на множестве языков программирования и технологий. Ниже приведен список наиболее распространенных языков, популярно используемых для создания RESTful API:

1. JavaScript/Node.js: Node.js является популярным выбором для создания RESTful API. Существует ряд фреймворков, таких как Express.js, Koa.js, и Hapi.js, которые облегчают разработку backend'а для RESTful API на Node.js.

2. Python: Python с библиотеками Flask, Django и FastAPI широко используется для создания RESTful API благодаря своей читаемости, масштабируемости и богатой экосистеме.

3. Java: С использованием фреймворков, таких как Spring Boot, Java EE, и JAX-RS, Java также обеспечивает мощные средства для создания RESTful API.

4. Ruby: Ruby с фреймворками Rails и Sinatra предоставляет удобные инструменты для создания RESTful API.

5. C#/.NET: Среда .NET и фреймворк ASP.NET также позволяют легко создавать RESTful API с помощью Web API.

6. Go: Go (или Golang) становится все более популярным из-за своей производительности и масштабируемости, и широко используется для создания RESTful API с использованием фреймворков, таких как Gin и Echo.

7. PHP: PHP с помощью фреймворков, таких как Laravel, Symfony и Lumen предоставляет инструменты для быстрого создания RESTful API.

8. Rust: Rust, из-за своей производительности и безопасности, также используется для создания RESTful API.

Это лишь несколько из множества языков, на которых можно создать RESTful API. Каждый из них имеет свои сильные стороны, поэтому выбор языка зависит от потребностей проекта, опыта разработчиков, экосистемы и других факторов.

# ПРИЛОЖЕНИЕ 2

**ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ ПОЛЕЙ, МЕТОДОВ, СВОЙСТВ КЛАССОВ, А ТАКЖЕ ПЕРЕМЕННЫХ И ФУНКЦИЙ В ФАЙЛАХ**

Таблица 2.1.1 – Описание полей и свойств класса Level

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Назначение** |
| Debug | public | Enum type | Данный уровень предназначен для максимального уровня логирования. |
| Info | public | Enum type | Данный уровень предназначен для вывода какой-либо информации, которая носит не критический, но довольно важный характер. |
| Warning | public | Enum type | Для вывода предупреждений (по аналогии с предупреждениями компилятора). |
| Error | public | Enum type | Для вывода ошибок, из-за которых отдельная компонента HTTP-сервера не может корректно завершить свою работу. |
| Critical | public | Enum type | Для вывода ошибок, при которых работа HTTP-сервера невозможна. |

Таблица 2.2.1 – Описание полей и свойств класса Method

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Назначение** |
| GET | public | Enum type | Для ответа на GET-запросы. |
| POST | public | Enum type | Для ответа на POST-запросы. |

Таблица 2.3.1 – Описание методов абстрактного класса HttpServerInterface

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Аргументы** | **Назначение** |
| addEndpoint | public | virtual void | const std::string &path, const std::string &response, Method method | Добавляет в HTTP-сервер новый эндпоинт. |

Таблица 2.4.1 – Описание методов абстрактного класса LoggerInterface

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Аргументы** | **Назначение** |
| log | public | virtual void | Level level, const std::string &message | Логирует переданное ей сообщение в файл и syslog с переданным уровнем логирования. |

Таблица 2.5.1 – Описание методов абстрактного класса HttpSessionInterface

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Аргументы** | **Назначение** |
| start | public | virtual void | - | Считывать HTTP-запрос и отвечать на него |

Таблица 2.6.1 – Описание методов абстрактного класса RESTAPIAPPInterface

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Аргументы** | **Назначение** |
| AddEndpoint | public | virtual void | const std::string &path, const std::string &response, const std::string &method) | Добавляет новый эндпоинт в сервис |
| RunServer | public | virtual void | - | Дает команду серверу начинать прослушивать входящие соединения |
| StopServer | public | virtual void | - | Останавливает сервер, закрывая сокет. |

Таблица 2.7.1 – Описание полей и свойств класса Logger

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Назначение** |
| logFile | private | std::ofstream | Нужен для записи в файл. |
| syslogEnabled | private | const bool | Нужна ли выгрузка в syslog помимо файла. |

Таблица 2.7.2 – Описание методов класса Logger

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Аргументы** | **Назначение** |
| Logger | public | constructor | const std::string &program\_name = "HTTPServer", const std::string &log\_file\_name = "log.txt", bool syslog\_enabled = true | Создает объект, открывает файл логов и системный лог. |
| ~Logger() | public | destructor | - | Разрушает объект, закрывает файл логов и системный лог. |
| log | public | void | Level level, const std::string &message | Логирует в файл и системный лог |
| writeToSyslog | private | void | Level level, const std::string &message | Логирует в системный лог. |
| writeToFile | private | void | Level level, const std::string &message | Логирует в файл. |

Таблица 2.8.1 – Описание полей и свойств класса HttpSession

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Назначение** |
| socket\_ | private | boost::asio::ip::tcp::socket | Нужен для открытия сокета. |
| request\_ | private | boost::asio::streambuf | Нужен для чтения http-запроса. |
| endpoints\_ | private | сonst std::unordered\_map<std::string, std::pair<std::string, Method>>& | Для ответов на соответствующие запросы. |
| enable\_cache | private | const bool | Флаг, который включает кеширование ответов. |
| logger | private | std::shared\_ptr<Logger> | Нужен для логирования. |
| cache | private | std::unordered\_map<std::string, std::string>& | Нужен для кеширования. |

Таблица 2.8.2 – Описание методов класса HttpSession

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Аргументы** | **Назначение** |
| HttpSession | public | constructor | boost::asio::ip::tcp::socket socket, const endpoint &endpoints, Logger::Ptr logger,  CACHE& cache, bool enable\_cache = true | Создает объект, начинает читать входящий запрос. |
| ~HttpSession | public | destructor | - | Разрушает объект. |
| do\_read | private | void | - | Читает входящий запрос и вызывает функцию для ответа на него. |
| do\_write | private | void | const std::string &response | Отвечает на входящий запрос. |

Таблица 2.9.1 – Описание полей и свойств класса HttpServer

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Назначение** |
| socket\_ | private | boost::asio::ip::tcp::socket | Нужен для открытия сокета. |
| request\_ | private | boost::asio::streambuf | Нужен для чтения http-запроса. |
| endpoints\_ | private | const std::unordered\_map<std::string, std::pair<std::string, Method>>& | Для ответов на соответствующие запросы. |
| enable\_cache | private | const bool | Флаг, который включает кеширование ответов. |
| logger | private | std::shared\_ptr<Logger> | Нужен для логирования. |
| cache | private | std::unordered\_map<std::string, std::string>& | Нужен для кеширования. |

Таблица 2.9.2 – Описание методов класса HttpServer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Аргументы** | **Назначение** |
| HttpServer | public | constructor | boost::asio::io\_context &io\_context,  Logger::Ptr logger, CACHE& cache, short port = 8080, bool enable\_cache = true | Создает объект, вызывает функцию, которая случает соответствующий порт. |
| ~HttpServer | public | destructor | - | Разрушает объект. |
| addEndpoint | public | void | const std::string &path, const std::string &response, Method method | Добавляет новый эндпоинт. |
| do\_accept | private | void | - | Открывает сокет и начинает слушать соответствующий порт. |

Таблица 2.10.1 – Описание полей и свойств класса RESTAPIAPP

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Назначение** |
| io\_context | private | boost::asio::io\_context | Нужна для запуска HTTP-сервера (внутренняя вещь либы boost). |
| server | private | std::shared\_ptr<HttpServer> | Сам HTTP-сервер. |
| logger | private | std::shared\_ptr<Logger> | Инструмент логирования. |
| cache | private | std::unordered\_map<std::string, std::string> | Инструмент кеширования. |

Таблица 2.10.2 – Описание методов класса RESTAPIAPP

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя** | **Модификатор доступа** | **Тип** | **Аргументы** | **Назначение** |
| RESTAPIAPP | public | constructor | uint32\_t port = 8080, const std::string& logfileName="log.txt" | Создает объект, инициализирует его поля. |
| ~RESTAPIAPP | public | destructor | - | Разрушает объект. |
| AddEndpoint | public | void | const std::string &path, const std::string &response, Method methodconst std::string &path, const std::string &response, const std::string &method="GET" | Добавляет новый эндпоинт. |
| RunServer | public | void | - | Запускает HTTP-сервер. |
| StopServer | public | void | - | Останавливает HTTP-сервер. |

# ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Изм. | Номера листов (страниц) | | | | Всего листов (страниц) в документе | № документа | Входящий № сопроводительного документа и дата | Подпись | Дата |
| измененных | замененных | новых | аннулированных |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |