

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ ДНР

**ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ**   
**«ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ»**

Факультет Физико-технический

Кафедра Компьютерных технологий (КТ)

|  |  |  |
| --- | --- | --- |
|  | | |
| Зав. кафедрой | | КТ |
| \_\_\_\_\_\_\_\_\_\_\_\_ | | А. М. Аверин |
| (подпись) | |  |
| «\_\_\_» | \_\_\_\_\_\_\_\_\_\_\_\_ 2022 г. | |

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе бакалавра 2 курса

на тему:

РАЗРАБОТКА МИКРО-СЕРВИСА ДЛЯ ПРЕОБРАЗОВАНИЯ HTML РАЗМЕТКИ В ИЗОБРАЖЕНИЕ

|  |  |  |  |
| --- | --- | --- | --- |
| Автор работы |  |  | А. В. Горбань |

подпись

|  |  |  |
| --- | --- | --- |
| Направление | 09.03.01 | Информатика и вычислительная техника |

Руководитель работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ст. преподаватель Мартыненко А.М.

подпись

Консультанты по разделам:

Техническое задание \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ доцент Т.В. Шарий

подпись

Нормоконтроль \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ст. лаборант В.Г. Медведева

подпись

Курсовая работа защищена \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

дата итоговая оценка комиссия

Подписи членов комиссии: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Донецк  
2022

ГОУ ВПО «Донецкий национальный университет»

кафедра Компьютерных технологий

Утверждаю

Зав. кафедрой КТ

\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись

\_\_\_\_\_\_\_\_\_\_\_\_\_

дата

**ЗАДАНИЕ**

на курсовую работу студента 2 курса Горбань А. В.

*Тема курсовой работы*: Разработка микро-сервиса для преобразования HTML разметки в изображение

*Краткая постановка задачи*: 1. Изучить и проанализировать предметную область по преобразовании HTML разметки в изображение. 2. Ознакомиться с программными продуктами для создания приложений по преобразованию HTML разметки в изображение. 3. Разработать техническое задание на создание приложения. 4. Разработать проект программного обеспечения системы. 5. Разработать средствами языка TypeScript приложение, которое позволяет преобразовать HTML разметку в изображение. 6. Протестировать программное обеспечение. 7. Оформить отчёт.

*Исходные данные*: 1. Документация по преобразованию HTML разметки в изображение. 2. Документация по языку программирования TypeScript.

*Ожидаемые результаты*: серверное приложение — микро-сервис для преобразования HTML разметки в изображение

*Календарный план работы:*

|  |  |  |
| --- | --- | --- |
| Даты консультаций | Этапы выполнения работы | Отметки о выполнении |
| 11.02.2022 | Постановка задачи и обсуждение литературы | выполнено |
| 01.03.2022 | Предварительное утверждение содержания отчёта | выполнено |
| 17.04.2022 | Утверждение проекта, алгоритмов, методов, технологий | выполнено |
| 22.04.2022 | Ход реализации проекта | выполнено |
| 23.04.2022 | Обсуждения организации тестирования программы | выполнено |
| 10.05.2022 | Демонстрация программного продукта руководителю | выполнено |
| 17.06.2022 | Оформление отчёта | выполнено |
| 27.06.2022 | Предоставление отчёта руководителю | выполнено |

Дата выдачи задания 01.02.2022 года

Студент И. И. Иванов

Руководитель А. М. Мартыненко

**АННОТАЦИЯ**

1. Отчет о курсовой работе: 53 c., 5 рис., 0 табл., 2 приложения, 11 источников.
2. Объект исследования – браузер, сервер.
3. Предмет исследования – серверное приложение для управления браузером.
4. Цель работы – разработка серверного приложения для преобразования HTML разметки в изображение, которая позволяла бы генерировать приложение по HTML разметке, присланной в XHR запросе.
5. Метод исследования – анализ возможностей языка TypeScript на платформе Node.js для создания и обработки серверного приложения.
6. В работе были использованы возможности платформы Node.js, языка программирования TypeScript, а так же express, express-fileupload, fs-extra, puppeteer, ajv, lodash и uuid пакетов.
7. В результате решения задачи было разработано приложение для преобразования HTML разметки в изображение. Приложение позволяет генерировать изображение по присланным HTML, CSS, изображениям и опциям, посредством XHR запроса.
8. Главный модуль приложения работает с браузером, запущенном на сервере. Главной операцией приложения является основной контроллер.
9. Приложение html2img-service может использоваться back-end и front-end разработчиками, web-мастерами и дизайнерами для генерации изображений по HTML разметке.

TypeScript, NODE.JS, WEB-SERVICE, MICRO-SERVICE, GNU/LINUX, XHR, CHROMIUM, HTML, CSS, JSON-SCHEMA, FILE SYSTEM, PUPPETEER, OBJECT-ORIENTED PROGRAMMING, EXPRESS

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 5](#__RefHeading___Toc12566_3889738379)

[1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 7](#__RefHeading___Toc494639103)

[1.1 Состояние вопроса 7](#__RefHeading___Toc494639104)

[1.2 Актуальность и цель работы 7](#__RefHeading___Toc494639105)

[2 ТЕХНИЧЕСКОЕ ЗАДАНИЕ 9](#__RefHeading___Toc494639106)

[2.1 Описание области применения и исходных данных приложения 9](#__RefHeading___Toc494639107)

[2.2 Требования к пользовательским интерфейсам 10](#__RefHeading___Toc494639108)

[2.3 Требования к аппаратным, программным и коммуникационным интерфейсам 11](#__RefHeading___Toc494639109)

[2.4 Требования к пользователям продукта 12](#__RefHeading___Toc494639110)

[2.5 Требования к адаптации на месте 13](#__RefHeading___Toc494639111)

[2.6 Функции продукта 14](#__RefHeading___Toc494639112)

[2.7 Ограничения 15](#__RefHeading___Toc494639113)

[3 ОБОСНОВАНИЕ ВЫБОРА ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ 17](#__RefHeading___Toc494639114)

[4 РАЗРАБОТКА МИКРО-СЕРВИСА ДЛЯ ПРЕОБРАЗОВАНИЯ HTML РАЗМЕТКИ В ИЗОБРАЖЕНИЕ 18](#__RefHeading___Toc494639115)

[4.1 Входные и выходные данные приложения 18](#__RefHeading___Toc494639116)

[4.2 Проектирование структуры приложения 19](#__RefHeading___Toc494639117)

[4.3 Описание алгоритмов работы скриптов системы или (Описание объектов и их взаимодействия для ООП) 21](#__RefHeading___Toc494639118)

[5 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА 29](#__RefHeading___Toc494639119)

[5.1 Аппаратные и программные средства создания и эксплуатации микро-сервиса 29](#__RefHeading___Toc494639120)

[5.2 Руководство пользователя 31](#__RefHeading___Toc494639121)

[5.3 Описание контрольных примеров 31](#__RefHeading___Toc494639122)

[ЗАКЛЮЧЕНИЕ 33](#__RefHeading___Toc494639123)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 34](#__RefHeading___Toc494639124)

[ПРИЛОЖЕНИЕ А Пример использования 35](#__RefHeading___Toc12568_3889738379)

[ПРИЛОЖЕНИЕ Б Фрагменты листинга 37](#__RefHeading___Toc494639126)

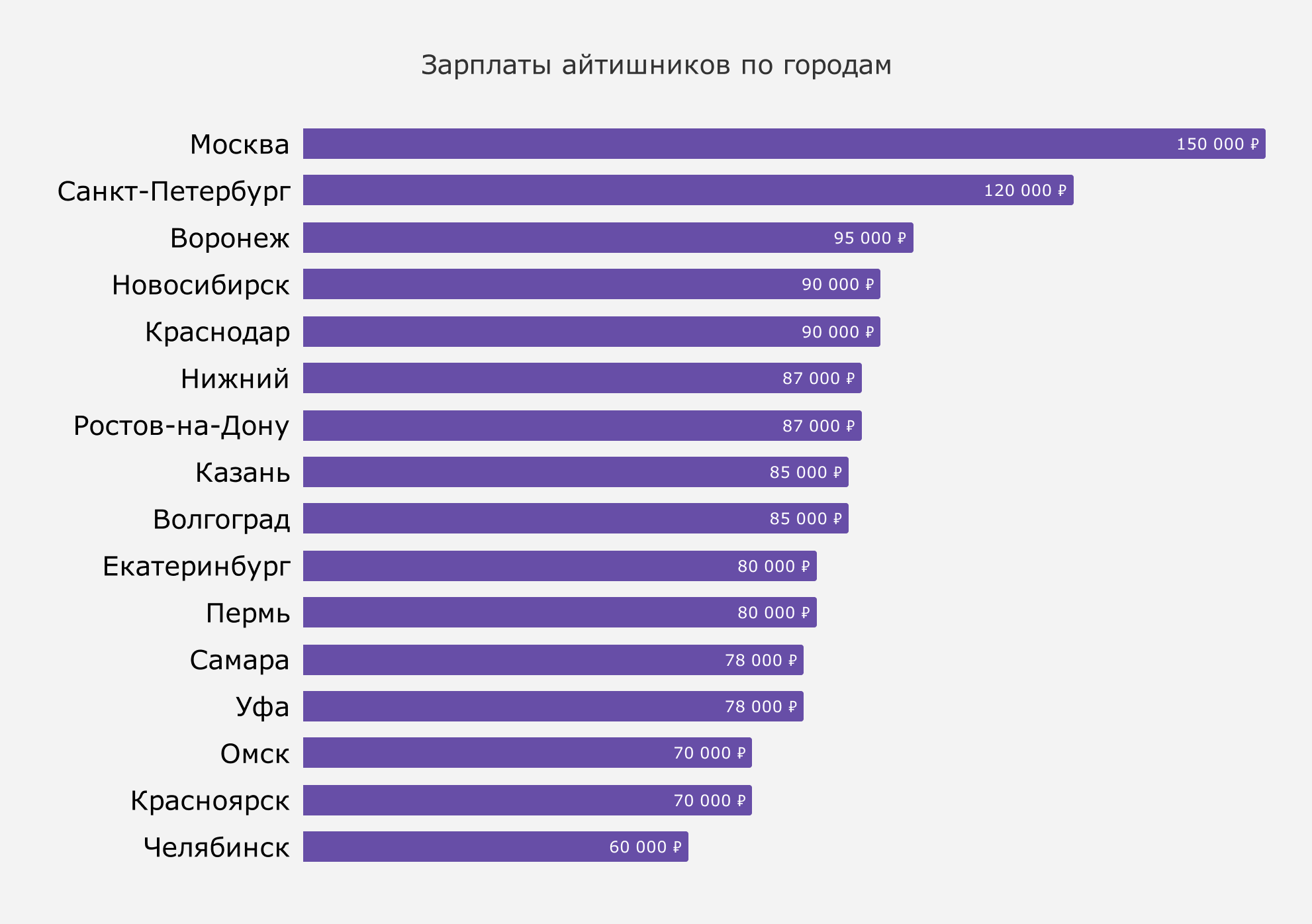
1. ВВЕДЕНИЕ
2. Современные информационные технологии в настоящее время быстрыми темпами внедряются во все сферы социально-культурной деятельности человека. Почему бы не использовать их для облегчения деятельность IT-специалистов?
3. В производимых разработчиками программных продуктах, будь то web-сайты, web-приложения, серверные приложения, сервисы, микро-сервисы, мобильные приложения, приложения для стационарных и персональных компьютеров, системы управления контентом, системы автоматизации процессов и так далее. Во всех них, периодически возникает потребность в генерации различных изображений, будь то чеки, фото с водяным знаком, билеты и тому подобное. Одним из самых удобных вариантов создавать подобный контент, это писать разметку с помощью специальных языков разметки. Самым распространенным из них является стандартизированный язык гипертекстовой разметки документов для просмотра веб-страниц в браузере, или же HTML (HyperText Markup Language). А также, для предания разметке определенной стилистической нагрузки, необходимо использовать каскадные таблицы стилей, или же CSS (Cascading Style Sheets).
4. После же написания разметки и стилей документа, необходимо перевести данный документ в формат изображения. В целях безопасности и улучшения пользовательского опыта, генерировать данные изображения в клиентской части приложения нежелательно. А значит, данные изображения необходимо генерировать на сервере.
5. Для этого необходимо написать определенную бизнес-логику в серверном приложении. Либо же, во избежание количественного разрастания кодовой базы основного серверного приложения, возможно выделить часть бизнес-логики, отвечающей за генерацию изображения на основе HTML-разметки и CSS стилей, в отдельную, подключаемую к основному серверному приложению библиотеку, или же, вынести необходимую бизнесс-логику в микро-сервис
6. Микро-сервисы — это современное представление сервис-ориентированной архитектуры, используемое для создания распределенных программных систем. Как и в SOA, модули в архитектуре микро-сервисов взаимодействуют по сети друг с другом для выполнения цели. Ещё одно сходство в том, что микро-сервисы используют протоколо-независимую технологию.
7. В данной курсовой работе будет реализовано серверное приложение микро-сервис, которое позволит front-end, back-end разработчикам, web-мастерам и дизайнерам позволит обеспечить простую автоматическую генерацию изображений в проектах.
8. 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Состояние вопроса

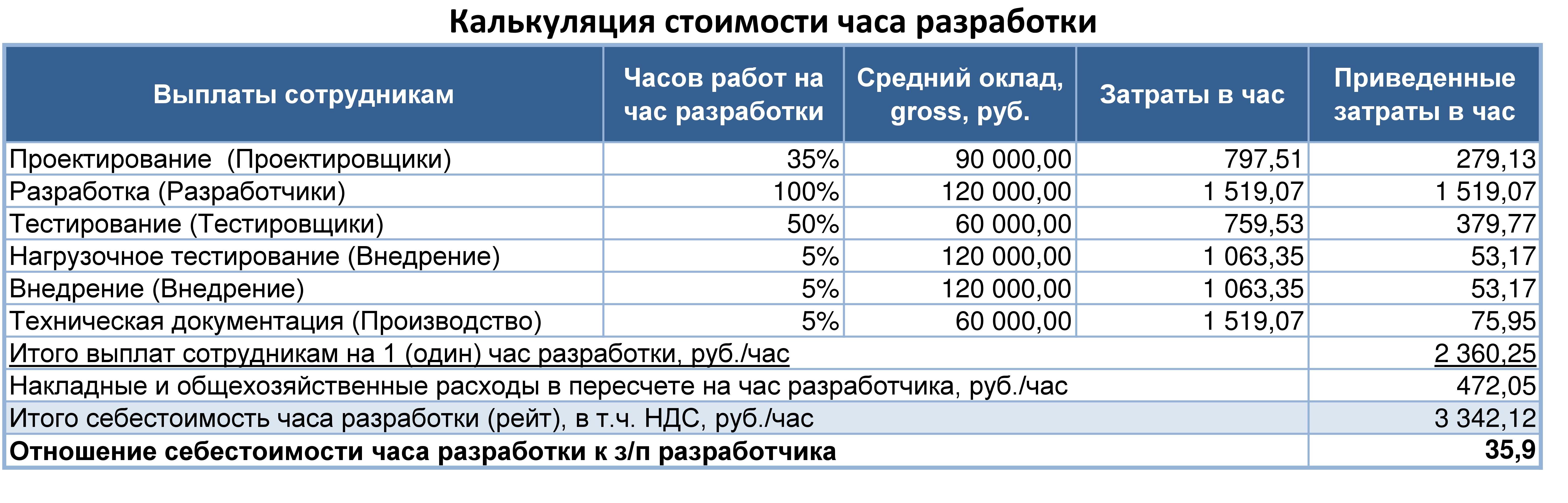
1. В настоящее время IT-индустрия охватывает все большие сферы деятельность человека. Создается все больше различных web-сайтов, приложений и сервисов, которые необходимо поддерживать и развивать. Соответственно, потребность в специалистах данной сферы постоянно растет.

1.2 Актуальность и цель работы

1. На рисунке 1.1 показан уровень средней заработной платы IT-специалистов по городам Российской Федерации на 2020 г.

Рисунок 1.1 — Средняя заработная плата IT-специалистов по городам Российской Федерации на 2020 г.

1. На рисунке 1.2 изображена таблица расчета стоимости 1 часа разработки.

Рисунок 1.2 - Таблица расчета стоимости 1 часа разработки.

1. Как видно из приведенных рисунков потребность рынка в квалифицированных IT-специалистах весьма значительна. Соответственно, и стоимость их работы высока.
2. Для снижения стоимости и сокращения сроков разработки зачастую используются готовые программные решения.
3. В процессе разработки программных продуктов периодически возникает потребность в генерации изображений. Это могут быть чеки, билеты, фирменные водные знаки, и тому подобное.
4. Для решения данной проблемы разработчик может воспользоваться микро-сервисом «html2img-service».
5. Целью данной работы будет разработка микро-сервиса, который позволит быстро и легко генерировать необходимые изображения.
6. 2 ТЕХНИЧЕСКОЕ ЗАДАНИЕ

2.1 Описание области применения и исходных данных приложения

1. Данное приложение предназначено для использования в программных продуктах по средствам API.
2. Исходными данными для приложения являются:
3. тип возвращаемого изображения;
4. ширина изображения;
5. высота изображения;
6. качество изображения;
7. флаг, нужно ли убирать задний фон изображения;
8. HTML разметка;
9. CSS стили;
10. изображения.
11. Декларация типа параметров запроса:

interface Params {

readonly returnImgType: ImgType;

}

Декларация типа GET-параметров запроса:

interface Query {

readonly width?: number;

readonly height?: number;

readonly quality?: number;

readonly omitBackground?: boolean;

}

Декларация типа тела запроса:

interface Body {

readonly html: string;

readonly css?: string;

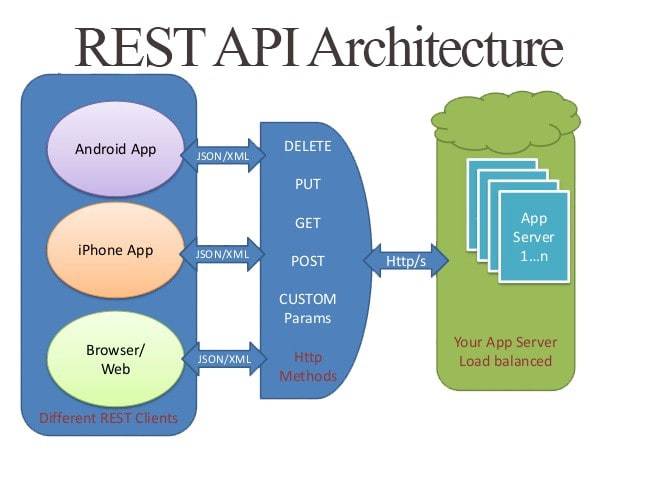
}

**2.2 Требования к пользовательским интерфейсам**

1. Так как пользователями разрабатываемого программного обеспечения являются разработчиками, пользовательский интерфейс представлен в виде RESTful API.
2. RESTful API позволяют разрабатывать все виды веб-приложений со всеми возможными операциями CRUD (create, read, update, delete).
3. Руководящие принципы REST предлагают использовать определенные методы HTTP для определенного типа действий с ресурсами на сервере (хотя технически возможно нарушить это руководство, но это крайне не рекомендуется).
4. Методы HTTP используемые в RESTful API:

* GET — для получения информации о ресурсе;
* POST — для добавления информации о ресурсе;
* PUT — для полного изменения или замены информации о ресурсе;
* PATCH — для частичного изменения информации о ресурсе;
* DELETE — для удаления информации о ресурсе.

1. Графическое представление RESTful API архитектуры представленно на рисунке 2.1.

Рисунок 2.1 — Графическое представление RESTful API архитектуры

2.3 Требования к аппаратным, программным и коммуникационным интерфейсам

1. Для установки и работы приложения необходимо иметь вычислительную систему следующей минимальной аппаратной конфигурации:
2. Процессор: 1.0 ГГц;
3. Оперативная память: 512 Мб;
4. Свободное место на жёстком диске: 150Мб;
5. Видеоадаптер: 64Мб;
6. Сетевая карта.
7. Для установки и работы приложения необходимо иметь вычислительную систему следующей минимальной программной конфигурации:
8. Операционная система: любой бистрибутив GNU/Linux, любой дистрибутив UNIX-систем, Microsoft Windows или MacOS;
9. Платформа выполнения JavaScript кода Node.js или Deno;
10. Бзаузер созданный на основе браузера Chromium, разработанного Google.
11. Для установки и работы приложения необходимо иметь вычислительную систему имеющую подключение ко глобальной сети интернет.

2.4 Требования к пользователям продукта

1. Системой должны иметь возможность пользоваться следующие категории пользователей: front-end, back-end разработчики, web-мастера и дизайнеры.
2. Front-end и back-end разработчики:

* Понимание принципов функционирования приложения;
* Понимание работы с XHR запросами;
* Умение написания программного кода или скриптов на языке программирования, поддерживающего работу с XHR запросами;
* Понимание принципов автоматизации за счет написания скриптов при помощи языков программирования.

1. Web-мастера:

* Понимание принципов функционирования приложения;
* Понимание работы с XHR запросами;
* Умение написания скриптов на языке программирования, поддерживающего работу с XHR запросами;
* Понимание принципов автоматизации за счет написания скриптов при помощи языков программирования.

Дизайнеры:

* Понимание принципов функционирования приложения;
* Понимание работы с XHR запросами;
* Умение написания скриптов на языке программирования, поддерживающего работу с XHR запросами;
* Умения написания скриптов использую no-code решения или инструменты графического программирования поддерживающие работу с XHR запросами;
* Понимание принципов автоматизации за счет написания скриптов.

2.5 Требования к адаптации на месте

Для запуска и корректной работы разрабатываемого приложения необходимо предварительно установленное программное обеспечение:

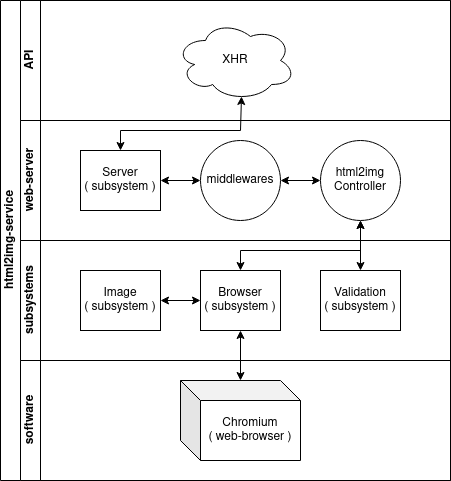
1. Операционная система: любой бистрибутив GNU/Linux, любой дистрибутив UNIX-систем, Microsoft Windows или MacOS;
2. Платформа выполнения JavaScript кода Node.js или Deno;
3. Бзаузер созданный на основе браузера Chromium, разработанного Google.

А так-же в качестве дополнительного программного обеспечения, могут быть использованы

1. Демон-менеджер процессов PM2;
2. web-сервер Nginx или Apache/
3. Еще необходима целевая директория, в которой будет находиться корень программы.

2.6 Функции продукта

1. Приложение должно обладать функционалом по работе с браузером, обработке изображений, управлении web-сервером и валидации входящих данных.
2. Приложение обладает одной, единой главной функцией, которая доступна всем классам пользователей. Эта функция является единственным и основным контроллером микро-сервиса.
3. Основной функцией продукта является функция для преобразования HTML-разметки, с соответственными ей CSS стилями, в изображение определенного переданного формата, а так же отправка этого изображения обратно в ответе запроса. Пользователь микро-сервиса сам выбирает формат итогового изображения путем передачи названия формата в запросе на микро-сервис.
4. Дабы воспользоваться сервисом, его пользователь должен отправить XHR запрос методом GET на определенный endpoint web-сервиса, которым и являеться разрабатываемое приложение.
5. На рисунке 2.2 показана схема функциональной структуры приложения html2img-service.

Рисунок 2.2 — Схема функциональной структуры приложения

2.7 Ограничения

Для корректной работы серверного приложения необходимо учитывать ограничения разрабатываемого программного продукта:

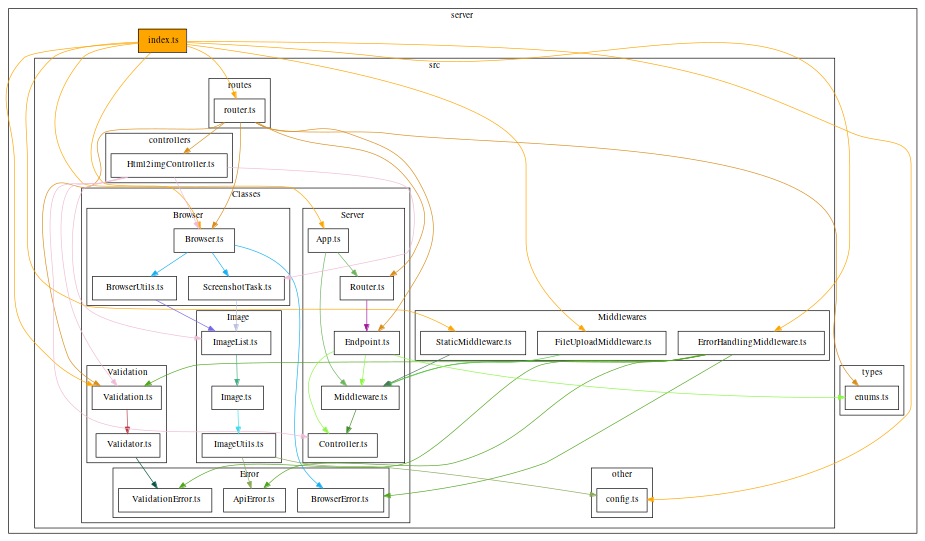
1. Приложение будет поддерживать только изображения форматов: png, jpeg и webp;
2. Приложение будет работать только с HTML-разметкой и CSS стилями;
3. Приложение будет работать только с полным набором обязательных входных данных;
4. Приложение будет работать только с полным набором установленных используемых библиотек.
5. Приложение не будет работать вне платформа выполнения JavaScript кода Node.js или Deno;
6. Приложение не будет работать в не скомпилорованном виде;
7. Приложение не будет работать без польного набора собственных исполняемых файлов;
8. Приложение должно использоваться в рамках локальной или глобальной сети;
9. Приложение не будет работать с ограничением прав на запись, чтение, изменение и удаление файлов.
10. Приложение не будет корректно работать без, установленного на машине, браузера;
11. Приложение будет некорректно работать с браузерами основанными не на chromium.
12. 3 ОБОСНОВАНИЕ ВЫБОРА ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ
13. Описание среды разработки:
14. Visual Studio Code — это редактор исходного кода, разработанный Microsoft для Windows, Linux и macOS. Позиционируется как «лёгкий» редактор кода для кроссплатформенной разработки веб- и облачных приложений.
15. Postman — это платформа API, позволяющая разработчикам проектировать, создавать, тестировать и повторять свои API.
16. Node.js — это программная платформа, основанная на движке V8, превращающая JavaScript из узкоспециализированного языка в язык общего назначения.
17. TypeScript — это язык программирования, представленный Microsoft в 2012 году и позиционируемый как средство разработки веб-приложений, расширяющее возможности JavaScript.
18. Описание производственной среды:
19. Node.js — это программная платформа, основанная на движке V8, превращающая JavaScript из узкоспециализированного языка в язык общего назначения.
20. PM2 (опционально) — это менеджер процессов для JavaScript-среды Node.js.
21. Nginx — это веб-сервер и почтовый прокси-сервер, работающий на Unix-подобных операционных системах.
22. 4 РАЗРАБОТКА МИКРО-СЕРВИСА ДЛЯ ПРЕОБРАЗОВАНИЯ HTML РАЗМЕТКИ В ИЗОБРАЖЕНИЕ

4.1 Входные и выходные данные приложения

1. Входные данные приложения приходят в URN и GET-параметрах из URI и теле XHR запроса.
2. Входными данными приложения являются 8 переменных:
3. Тип возвращаемого изображения приходит в URN;
4. Ширина возвращаемого изображения (опционально) приходит в GET-параметрах;
5. Высота возвращаемого изображения (опционально) приходит в GET-параметрах;
6. Качество возвращаемого изображения (опционально) приходит в GET-параметрах;
7. Флаг, отвечающий за наличие заднего фона возвращаемого изображения (опционально) приходит в GET-параметрах;
8. HTML разметка возвращаемого изображения приходит в теле запроса;
9. CSS стили возвращаемого изображения (опционально) приходят в теле запроса;
10. Изображения для вставки в HTML разметку и CSS стили возвращаемого изображения (опционально) приходят в теле запроса;
11. Выходными данными приложения является сгенерированное изображение или ошибка.

4.2 Проектирование структуры приложения

1. Структурно приложение состоит из следующих подсистем и  
   TypeScript-скриптов:
2. подсистема управления сервером — классы: App, Controller, Endpoint, Middleware и Router;
3. подсистема управления браузером — классы: Browser, BrowserUtils и ScreenshotTask;
4. подсистема манипуляции изображениями — классы: Image, ImageList и ImageUtils;
5. подсистема валидации — классы: Validation и Validator;
6. классы ошибок: ApiError, BrowserError и ValidationError;
7. функциональные классы: Html2ImgController, FileUploadMiddleware, ErrorHandlingMiddleware, StaticMiddleware и MainRouter;
8. настройки среды выполнения — файл config.ts;
9. декларация типов — файлы: enum.ts, index.d.ts и utils.d.ts.
10. Схема взаимодействия классов и скриптов приложения приведена   
    на рисунке 4.1.

Рисунок 4.1 – Структура микро-сервиса

4.3 Описание алгоритмов работы скриптов системы или (Описание объектов и их взаимодействия для ООП)

1. Класс Browser — модуль для взаимодействия с браузером.
2. Поля классы Browser:

* protected \_browser — экземпляр класса Browser от puppeteer.

1. Методы класса Browser:

* private \_launchBrowser — запускает браузер;
* public closeBrowser — закрывает браузер;
* private \_createPage — создает страницу в браузере;
* private сalcNativeHeight — подсчитывает нативную высоту контента на странице;
* public screenshot — делает скриншот контента страницы.

1. Класс BrowserUtils — модуль с утилитами для класса Browser.
2. Поля класса BrowserUtils:

* static htmlRootId — html id для корневого тега.

1. Методы класса BrowserUtils:

* static generateViewport — генерирует настройки viewport для страницы;
* static insertImgIntoHTML — вставляет изображения в html разметку;
* static generateContent — генерирует html разметку.

1. Класс ScreenshotTask — данные и параметры для скриншота.
2. Поля класса ScreenshotTask:

* public data — объект, в котором хранятся html и css;
* public imgs — хранится экземпляр класса ImageList:
* public opts — объект, в котором хранятся параметры для скриншота, а именно: width, height, returnImgType, quality и omitBackground.

1. Класс Image — модуль, для работы с отдельным изображением.
2. Поля класса Image:

* private \_name — наименование изображения;
* private \_path — путь к изображению;
* private \_file — файл изображения.

1. Геттеры и сеттеры класса Image:

* get uri — отдает сгенерированный uri изображения.

1. Методы класса Image:

* public move — перенести изображение из временной директории в постоянную;
* public removeTmp — удалить изображение из временной директории;
* public remove — удалить изображение из постоянной директории.

1. Класс ImageList — список изображений.
2. Поля класса ImageList:

* public imgs — массив экземпляров класса Image и их названий.

1. Методы класса ImageList:

* private \_prepareImgs — подготавливает изображения для поля imgs;
* public removeImgs — удаляет изображения из постоянной директории.

1. Класс ImageUtils - модуль с утилитами для класса Image.
2. Методы класса ImageUtils:

* private static imgMimetypeToExtension — преобразовывает mime тип изображения в расширение изображения;
* public static generateName — генерирует наименование изображения;
* public static generatePath — генерирует путь к изображению;
* public static generateUri — генерирует uri изображения.

1. Класс ApiError — класс ошибки, расширяющий базовый класс Error, предназначенный для выбрасывания ошибок из контроллеров и миддлвейров.
2. Поля класса ApiError:

* public status — поле для хранения числового HTTP статуса ошибки
* public message — поля для хранения сообщения ошибки (унаследовано от класса Error);
* public name — поля для хранения наименования ошибки (унаследовано от класса Error).

1. Методы класса ApiError:

* public static badRequest — статический метод для создания экземпляра класса ApiError с преднастройками;
* public static unauthorized — статический метод для создания экземпляра класса ApiError с преднастройками;
* public static forbidden — статический метод для создания экземпляра класса ApiError с преднастройками;
* public static notFound— статический метод для создания экземпляра класса ApiError с преднастройками;
* public static internal— статический метод для создания экземпляра класса ApiError с преднастройками;

1. Класс BrowserError — класс ошибки, расширяющий базовый класс Error, предназначенный для выбрасывания ошибок из подсистемы работы с браузером.
2. Поля класса BrowserError:

* public message — поля для хранения сообщения ошибки (унаследовано от класса Error);
* public name — поля для хранения наименования ошибки (унаследовано от класса Error).

1. Класс ValidationError — класс ошибки, расширяющий базовый класс Error, предназначенный для выбрасывания ошибок из подсистемы валидации.
2. Поля класса ValidationError:

* public info — поле для хранения исходных ошибок ajv валидации;
* public message — поля для хранения сообщения ошибки (унаследовано от класса Error);
* public name — поля для хранения наименования ошибки (унаследовано от класса Error).

1. Класс Validation — класс, предназначенный для генерации валидаторов по валидационным схемам.
2. Поля класса Validation:

* private \_ajv — экземпляр класса ajv

1. Методы класса Validation:

* private \_activateAjvPlugins — приватный метод для активации ajv плагинов;
* public generateValidator — метод для генерации валидотора по переданной JSON Schema валидации;
* public getErrorMessage — метод для получения сообщения ошибки из ajv ошибки.

1. Класс Validator — класс предоставляющий возможность валидации данных по предопределенной JSON Schema валидации.
2. Поля класса Validator:

* private \_validator — поле, хранящее ajv валидатор.

1. Методы класса Validator:

* public validate — метод для валидации переданных данных.

1. Класс App — главный класс приложения отвечающий за запуск и работы web-сервера.
2. Поля класса App:

* private \_app — поле хранящее в себе экземпляр класса Express.Application;
* private \_port — поле, хранящее в себе порт, который необходимо прослушивать web-серверу.

1. Методы класса App:

* private useMiddlewares — метод для регистрации миддлвейров в приложении;
* private useRoutes — метод для регистрации роутеров в приложении;
* public start — метод для запуска web-сервера.

1. Абстрактный класс Controller — абстрактный класс, предназначенный для создания контроллеров
2. Геттеры и сеттеры абстрактного класса Controller:

* get controller — геттер для получения функции-контроллера.

1. Методы абстрактного класса Controller:

* protected abstract \_controller — абстрактный метод функции-контроллера.

1. Абстрактный класс Middleware — абстрактный класс, расширяющий абстрактный класс Controller, предназначенный для создания миддлвейра.
2. Класс Endpoint — класс конечной точки запроса.
3. Поля класса Endpoint:

* private \_method — поле, хранящее метод конечной точки;
* private \_route — поле, хранящее путь конечной точки;
* private \_beforeMiddlewares — поле, хранящее middlewares, которые должны быть выполнены до выполнения контроллера;
* private \_ controller — поле, хранящее контроллер конечной точки;
* private \_afterMiddlewares — поле, хранящее middlewares, которые должны быть выполнены после выполнения контроллера.

1. Методы класса Endpoint:

* public use — метод для использования конечной точки в роутере.

1. Абстрактный класс Router — абстрактный класс, предназначенный для создания роутера.
2. Поля абстрактного класса Router:

* private \_router — поле, хранящее экземпляр класса ExpressCore.Express;
* private \_route — поле, хранящее путь;
* protected abstract \_endpoints — абстрактное поле, хранящее массив экземпляров класса Endpoint.

1. Геттеры и сеттеры абстрактного класса Router:

* get route — геттер для получения пути;
* get router — геттер для получения роутера.

1. Методы абстрактного класса Router:

* protected useEndpoints — метод для регистрации конечных точек в роутере.

1. Класс Html2imgController — класс, расширяющий абстрактный класс Controller, основной контроллер приложения.
2. Поля класса Html2IngController:

* private \_browser — поле, хранящее экземпляр класса Browser;
* private \_validation — поле, хранящее экземпляр класса Validation;
* private \_paramsValidator — поле, хранящее экземпляр класса Validator для валидации параметров запроса;
* private \_queryVaildator — поле, хранящее экземпляр класса Validator для валидации GET-параметров запроса;
* private \_bodyValidator — поле, хранящее экземпляр класса Validator для валидации тела запроса.

1. Геттеры и сеттеры класса Html2ImgController:

* get controller — геттер для получения функции-контроллера (унаследован от абстрактного класса Controller).

1. Методы класса Html2ImgController:

* protected \_controller — метод функция-контроллер (реализация абстрактного метода от абстрактного класса Controller).

1. Класс ErrorHandlingMiddleware — класс, расширяющий абстрактный класс Middleware, middleware — обработчик ошибок.
2. Геттеры и сеттеры класса ErrorHandlingMiddleware:

* get controller — геттер для получения функции-контроллера (унаследован от абстрактного класса Middleware).

1. Методы класса ErrorHandlingMiddleware:

* protected \_controller — метод функция-контроллер (реализация абстрактного метода от абстрактного класса Middleware).

1. Класс FileUploadMiddleware — класс, расширяющий абстрактный класс Middleware, middleware — адаптер библиотеки express-fileupload для парсинга изображений, приходящих в теле запроса.
2. Геттеры и сеттеры класса FileUploadMiddleware:

* get controller — геттер для получения функции-контроллера (унаследован от абстрактного класса Middleware).

1. Методы класса FileUploadMiddleware:

* protected \_controller — метод функция-контроллер (реализация абстрактного метода от абстрактного класса Middleware).

1. Класс StaticMiddleware — класс, расширяющий абстрактный класс Middleware, middleware — адаптер middleware Express.static для назначения статических файлов и директорий web-сервера.
2. Геттеры и сеттеры класса StaticMiddleware:

* get controller — геттер для получения функции-контроллера (унаследован от абстрактного класса Middleware).

1. Методы класса StaticMiddleware:

* protected \_controller — метод функция-контроллер (реализация абстрактного метода от абстрактного класса Middleware).

1. Класс MainRouter — класс, расширяющий абстрактный класс Router, основной роутер приложения.
2. Поля класса MainRouter:

* protected \_endpoints — абстрактное поле, хранящее массив экземпляров класса Endpoint (реализация абстрактного поля от абстрактного класса Router).

1. Геттеры и сеттеры класса MainRouter:

* get route — геттер для получения пути (унаследовано от абстрактного класса Router);
* get router — геттер для получения роутера (унаследовано от абстрактного класса Router).

1. Методы класса MainRouter:

* protected useEndpoints — метод для регистрации конечных точек в роутере (унаследовано от абстрактного класса Router).

1. 5 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА

5.1 Аппаратные и программные средства создания и эксплуатации микро-сервиса

1. Аппаратные требования для работы сервиса в среде разработки:

* Процессор с тактовой частотой от 1.3 ГГц;
* Оперативная память от 2 ГБ;
* Видеокарта;
* Монитор;
* Клавиатура;
* Компьютерная мышь;
* Сетевая карта;
* Видео-карта.

1. Программные требования для работы сервиса в среде разработки:

* Операционная система: любой бистрибутив GNU/Linux, любой дистрибутив UNIX-систем, Microsoft Windows или MacOS;
* Платформа выполнения JavaScript кода Node.js или Deno;
* Бзаузер созданный на основе браузера Chromium, разработанного Google;
* IDE.

1. Аппаратные требования для работы сервиса в производственной среде:

* Процессор с тактовой частотой от 1.0 ГГц;
* Оперативная память от 1 ГБ;
* Сетевой адаптер.

1. Программные требования для работы сервиса в среде разработки:

* Операционная система: любой бистрибутив GNU/Linux, любой дистрибутив UNIX-систем, Microsoft Windows или MacOS;
* Платформа выполнения JavaScript кода Node.js или Deno;
* Бзаузер созданный на основе браузера Chromium, разработанного Google.
* Менеджер процессов для JavaScript-среды Node.js PM2;
* Web-браузер на основе chromium.

5.2 Руководство пользователя

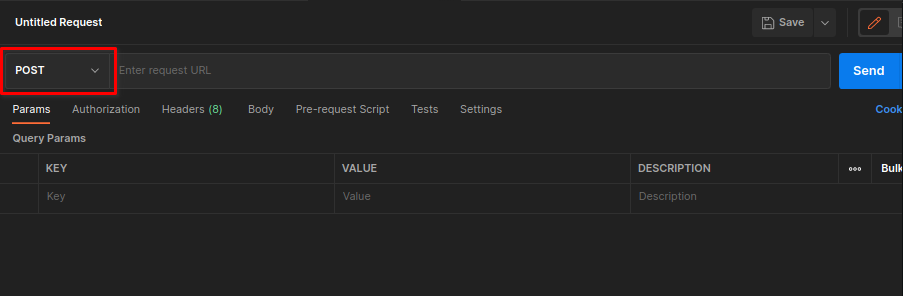
1. Для работы сервиса необходимо его инсталлировать:
2. Инсталлировать среду выполнения Node.js (стабильной или тестовой версии), следуя инструкциям по установке с официального сайта;
3. Склонируйте локально репозиторий сервиса с https://github.com/AntonGorban/html2img-service.git;
4. Установить зависимости проекта, командой npm install;
5. Соберите приложение командой npm run build;
6. Переместите собранное приложение из директории build в подготовленную директорию для приложения;
7. Запустите приложение с помощью PM2, командой pm2 start index.js;
8. Протестируйте приложение.
9. Дополнительно вы можете настроить домен для сервиса используя Nginx или Apache. Так-же вы можете настроить firewall ufw или iptables.

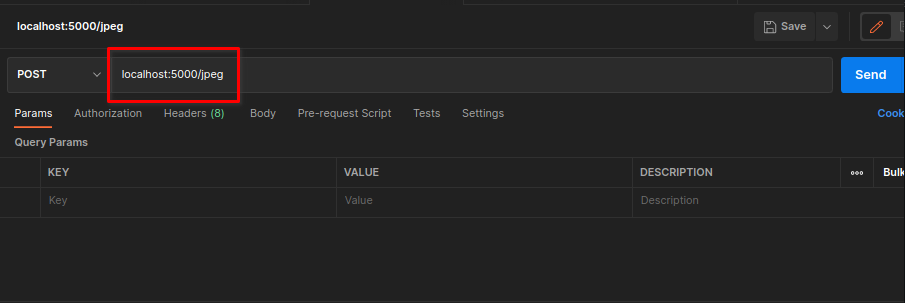
5.3 Описание контрольных примеров

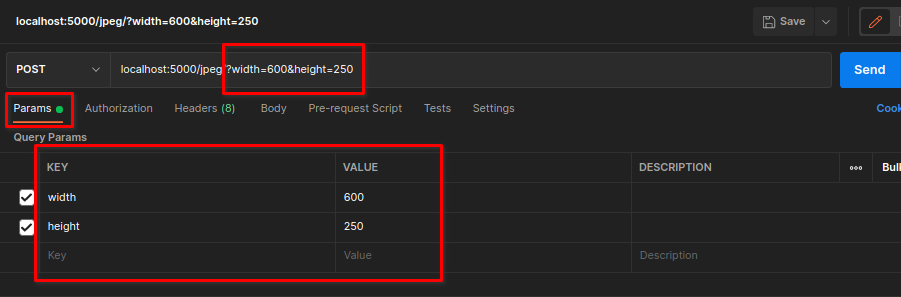
1. Для использования микро-сервиса в вашем проекте вы можете использовать fetch, axios и т.п. для JavaScript / TypeScript, или подобные библиотеки в других языках программирования.
2. Для использования микро-сервиса, вам нужно отправить XHR запрос методом POST на URI по шаблону:

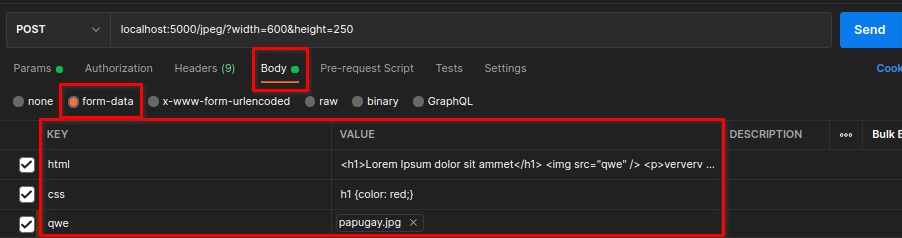
* Если вы не подключали домен: (протокол)://(ip):(port)/(тип возвращаемого изображения);
* Если вы подключали домен: (протокол)://(домен)/(тип возвращаемого изображения).

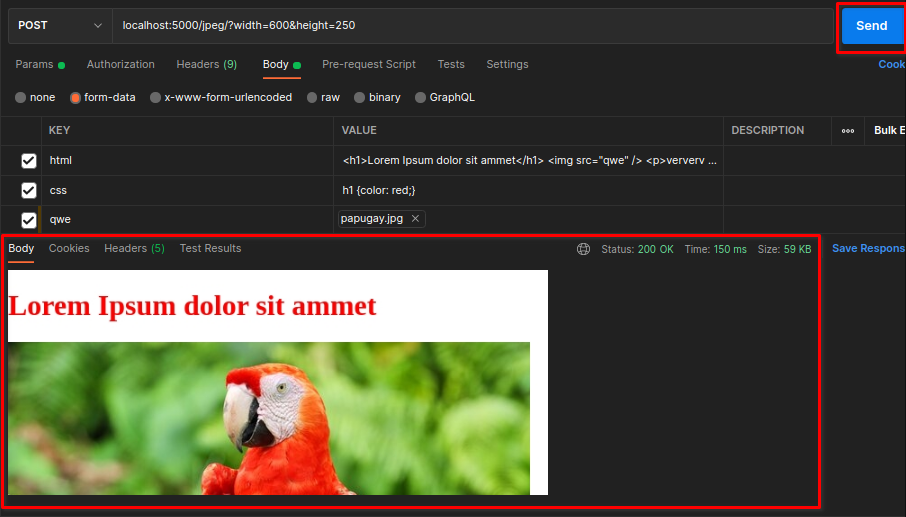
1. Тестировать микро-сервис мы будем с помощью программного обеспечения Postman.
2. Создайте новый запрос в Postman, выберите метод запроса POST (рисунок А.1). Введите URL-адрес сервиса по вышеописанному шаблону (рисунок А.2). Далее введите параметры запроса во вкладке «Params» (рисунок А.3) и тело запроса на вкладке «Body», выбрав «form-data»  
   (рисунок А.4). В тело запроса вы можете добавлять изображения с придуманным полем «KEY». В дальнейшем ваши изображения будут вставлены в HTML и CSS, на месте слов соответствующим «KEY».
3. После заполнения всех необходимых данных, вы можете нажать на кнопку «Send». Таким образом вы отправите запрос на сервер, он провалидирует ваши данные, и если все хорошо, сгенерирует и отдаст вам изображение (рисунок А.5). А если, ваши данные окажутся невалидными, тогда сервер отправит вам ошибку (рисунок А.6).
4. ЗАКЛЮЧЕНИЕ
5. В результате разработки спроектирован микро-сервис для преобразования HTML разметки в изображение.
6. Разработано приложение, которое предоставляет возможность генерации изображения на основе HTML разметки.
7. Приложение осуществляет генерацию изображения на основе HTML разметки.
8. Для решения задачи был использован язык программирования TypeScript.
9. Приложение html2img-service может использоваться front-end и back-end разработчиками, web-мастерами и дизайнерами для генерации изображений на основе HTML разметки.
10. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ
11. Express.js : официальная документация фреймворка «Express»: [сайт] — URL: https://expressjs.com/en/starter/installing.html;
12. TypeScript : официальная документация по языку программирования «TypeScript»: [сайт] — URL: https://www.typescriptlang.org/docs/;
13. Wanago : блог по написанию приложения на Express использую язык программирования TypeScript : [сайт] — URL: https://wanago.io/courses/typescript-express-tutorial/;
14. GeeksfoGeeks : статья «How to use express in typescript ?» : [сайт] — URL: https://www.geeksforgeeks.org/how-to-use-express-in-typescript/:
15. Puppeteer : Официальная документация библиотеки «Puppeteer» : [сайт] — URL: https://pptr.dev;
16. Node package manager (npm) : Официальная документация библиотеки «express-fileupload» : [сайт] — URL: https://www.npmjs.com/package/express-fileupload;
17. Node package manager (npm) : Официальная документация библиотеки «fs-extra» : [сайт] — URL: https://www.npmjs.com/package/fs-extra;
18. AJV : официальная документация библиотеки «ajv» : [сайт] — URL: https://ajv.js.org/guide/getting-started.html;
19. Lodash : Официальная документация библиотеки «Lodash» : [сайт] — URL: https://lodash.com/docs/4.17.15;
20. Хабр : статья «Зарплаты айтишников в первой половине 2020: +8% за счет Москвы» : [сайт] — URL: https://habr.com/ru/article/511700/;
21. hr-portal : Статья «Сколько стоит час работы программиста» : [сайт] — URL: https://hr-portal.ru/story/skolko-stoit-chas-raboty-programmista
22. ПРИЛОЖЕНИЕ А  
      
    Пример использования

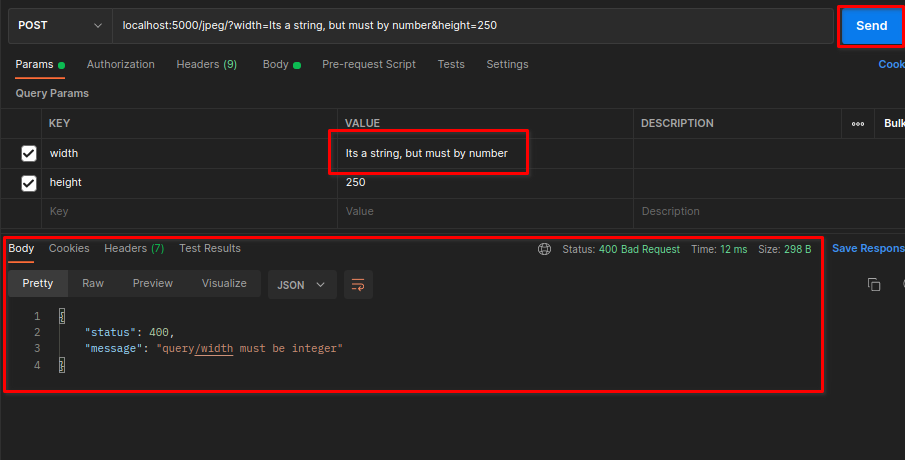
Рисунок А.1 — Создание «POST» запроса в «Postman»

Рисунок А.2 — Заполнение URI в «Postman»

Рисунок А.3 — Заполнение вкладки «Params» в «Postman»

Рисунок А.4 — Заполнение вкладки «Body» в «Postman»

Рисунок А.5 — Отправка подготовленного запроса в «Postman»

Рисунок А.6 — Отправка заведомо невалидного запроса в «Postman»

1. ПРИЛОЖЕНИЕ Б  
     
   Фрагменты листинга

Листинг Б.1 — Файл «/src/Classes/Server/App.ts»

import express from "express";

import { Middleware } from "./Middleware";

import { Router } from "./Router";

export class App {

private readonly \_app: express.Application;

private readonly \_port: number;

constructor(

port: number,

beforeMiddlewares: ReadonlyArray<Middleware>,

routers: ReadonlyArray<Router>,

afterMiddlewares?: ReadonlyArray<Middleware>

) {

this.\_app = express();

this.\_app.use(express.json());

this.\_port = port;

this.useMiddlewares(beforeMiddlewares);

this.useRouters(routers);

!!afterMiddlewares && this.useMiddlewares(afterMiddlewares);

}

private useMiddlewares(middlewares: ReadonlyArray<Middleware>) {

middlewares.forEach((middleware) => this.\_app.use(middleware.controller));

}

private useRouters(routers: ReadonlyArray<Router>) {

routers.forEach((router) => this.\_app.use(router.route, router.router));

}

public start() {

try {

this.\_app.listen(this.\_port, () => {

console.log(`Server started on port ${this.\_port}`);

});

} catch (error) {

console.error("Server does not start", error);

}

}

}

Листинг Б.2 — Файл «index.ts»

require("dotenv").config();

import fse from "fs-extra";

import { Browser } from "./src/Classes/Browser/Browser";

import { App } from "./src/Classes/Server/App";

import { Validation } from "./src/Classes/Validation/Validation";

import { ErrorHandlingMiddleware } from "./src/Middlewares/ErrorHandlingMiddleware";

import { FileUploadMiddleware } from "./src/Middlewares/FileUploadMiddleware";

import { StaticMiddleware } from "./src/Middlewares/StaticMiddleware";

import { paths, PORT } from "./src/other/config";

import { MainRouter } from "./src/routes/router";

fse.ensureDir(paths.tmp);

fse.ensureDir(paths.static.root);

fse.ensureDir(paths.static.img);

const browser = new Browser();

const validation = new Validation();

const app = new App(

PORT,

[

new StaticMiddleware(paths.static.root),

new FileUploadMiddleware({

limits: { fileSize: 20971520 },

useTempFiles: true,

tempFileDir: paths.tmp,

}),

],

[new MainRouter(browser, validation)],

[new ErrorHandlingMiddleware(validation)]

);

app.start();

/\* ---------------------------- On Exit Handling ---------------------------- \*/

let wasCleanedUp = false;

const runBeforeExiting = (callback: Function) =>

["exit", "SIGINT", "SIGUSR1", "SIGUSR2", "uncaughtException"].forEach(

(exitSignal) => {

process.on(exitSignal, async () => {

console.log("exit signal");

if (!wasCleanedUp) {

await callback();

wasCleanedUp = true;

}

process.exit();

});

}

);

runBeforeExiting(() =>

browser

.closeBrowser()

?.then(() => console.log("Browser closed"))

.then(() => console.log("App closed"))

);

/\* --------------------------- / On Exit Handling --------------------------- \*/

Листинг Б.3 — Файл «/types/index.d.ts»

import Express from "express";

import ExpressCore from "express-serve-static-core";

import { ObjectWithGivenValues } from "./utils";

export namespace Server {

interface ParamsType extends ExpressCore.ParamsDictionary {

readonly [key: string]: string | number;

}

interface QueryType extends ExpressCore.Query {

readonly [key: string]: string | number | boolean;

}

interface BodyType {

readonly [key: string]: any;

}

type ControllerType<P = ParamsType, Q = QueryType, B = BodyType, R = any> = (

request: Express.Request<

ObjectWithGivenValues<P, ParamsType[0] | undefined>,

R,

ObjectWithGivenValues<B, unknown>,

ObjectWithGivenValues<Q, QueryType[0] | undefined>

>,

response: Express.Response<R>,

next: Express.NextFunction

) => Promise<Express.Response<R> | ReturnType<Express.NextFunction>>;

type ExpressRouterType = ExpressCore.Express;

}

export type ImgType = "png" | "jpeg" | "webp";

Листинг Б.4 — Файл «/src/Classes/Browser/Browser.ts»

import lo from "lodash";

import Puppeteer from "puppeteer";

import { BrowserError } from "../Error/BrowserError";

import { BrowserUtils } from "./BrowserUtils";

import { ScreenshotTask } from "./ScreenshotTask";

export class Browser {

protected \_browser: Puppeteer.Browser | undefined;

constructor() {

this.\_launchBrowser();

}

private readonly \_launchBrowser = async () => {

this.\_browser = await Puppeteer.launch();

console.log("Browser opened");

};

public readonly closeBrowser = () =>

new Promise((resolve) => {

if (!!this.\_browser) this.\_browser.close().then(() => resolve(true));

else resolve(true);

});

private readonly \_createPage = () => {

if (!this.\_browser) throw new BrowserError("browser is not running");

return this.\_browser.newPage();

};

private readonly calcNativeHeight = async (

page: Puppeteer.Page,

height: number

) =>

page

.$(`#${BrowserUtils.htmlRootId}`)

.then(

(rootElement) =>

rootElement

?.boxModel()

.then(

(box) =>

lo.max([

...(box?.margin.map(({ y }) => y) || []),

box?.height || height,

]) || height

) || height

);

public readonly screenshot = async (task: ScreenshotTask) => {

try {

const page = await this.\_createPage();

const viewport = BrowserUtils.generateViewport({

width: task.opts.width,

height: task.opts.height,

});

await page.setViewport(viewport);

await page.setContent(

BrowserUtils.generateContent({ ...task.data, imgs: task.imgs })

);

const width = task.opts.width || viewport.width;

const height =

task.opts.height ||

(await this.calcNativeHeight(

page,

task.opts.height || viewport.height

));

const screenshot = await page.screenshot({

type: task.opts.returnImgType,

clip: { x: 0, y: 0, width, height },

encoding: "binary",

omitBackground: task.opts.omitBackground || false,

quality:

task.opts.returnImgType !== "png"

? task.opts.quality || 100

: undefined,

});

await page.close();

return screenshot;

} catch (error) {

console.error(error);

if (error instanceof Error) throw new BrowserError(error.message);

throw new BrowserError("что-то пошло не так");

}

};

}

Листинг Б.5 — Файл «/src/Classes/Browser/ScreenshotTask.ts»

import { ImgType } from "../../../types";

import { ImageList } from "../Image/ImageList";

export class ScreenshotTask {

constructor(

public readonly data: {

readonly html: string;

readonly css?: string;

},

public readonly imgs: ImageList,

public readonly opts: {

readonly width?: number;

readonly height?: number;

readonly returnImgType: ImgType;

readonly quality?: number;

readonly omitBackground?: boolean;

}

) {}

}

Листинг Б.6 — Файл «/src/Classes/Image/Image.ts»

import { UploadedFile } from "express-fileupload";

import fse from "fs-extra";

import { ImageUtils } from "./ImageUtils";

export class Image {

private readonly \_name: string;

private readonly \_path: string;

constructor(private readonly \_file: UploadedFile) {

this.\_name = ImageUtils.generateName(this.\_file.mimetype);

this.\_path = ImageUtils.generatePath(this.\_name);

}

public readonly move = () => {

return this.\_file.mv(this.\_path);

};

public readonly removeTmp = () => {

return fse.remove(this.\_file.tempFilePath);

};

public readonly remove = () => {

return fse.remove(this.\_path);

};

get uri() {

return ImageUtils.generateUri(this.\_name);

}

}

Листинг Б.7 — Файл «/src/Classes/Server/Controller.ts»

import { Server } from "../../../types";

export abstract class Controller<

P = Server.ParamsType,

Q = Server.QueryType,

B = Server.BodyType,

R = any

> {

protected abstract readonly \_controller: Server.ControllerType<P, Q, B, R>;

get controller() {

return this.\_controller;

}

}

Листинг Б.8 — Файл «/src/Classes/Server/Endpoint.ts»

import { Server } from "../../../types";

import { ApiMethods } from "../../../types/enums";

import { Controller } from "./Controller";

import { Middleware } from "./Middleware";

export class Endpoint {

constructor(

private readonly \_method: ApiMethods,

private readonly \_route: string,

private readonly \_beforeMiddlewares: ReadonlyArray<Middleware>,

private readonly \_controller: Controller<any, any, any>,

private readonly \_afterMiddlewares?: ReadonlyArray<Middleware>

) {}

public use(router: Server.ExpressRouterType) {

router[this.\_method](

this.\_route,

...this.\_beforeMiddlewares.map((middleware) => middleware.controller),

this.\_controller.controller,

...(!!this.\_afterMiddlewares

? this.\_afterMiddlewares.map((middleware) => middleware.controller)

: [])

);

}

}

Листинг Б.9 — Файл «/src/Classes/Server/Router.ts»

import Express from "express";

import { Server } from "../../../types";

import { Endpoint } from "./Endpoint";

export abstract class Router {

private readonly \_router: Server.ExpressRouterType;

private readonly \_route: string;

protected abstract readonly \_endpoints: ReadonlyArray<Endpoint>;

constructor(route: string) {

this.\_route = route;

this.\_router = Express();

}

protected useEndpoints() {

this.\_endpoints.forEach((endpoint) => endpoint.use(this.\_router));

}

get route() {

return this.\_route;

}

get router() {

return this.\_router;

}

}

Листинг Б.10 — Файл «/src/Classes/Validation/Validator.ts»

import { ValidateFunction } from "ajv";

import lo from "lodash";

import { ValidationError } from "../Error/ValidationError";

export class Validator<T extends {}> {

constructor(

private readonly \_validator: ValidateFunction,

private readonly \_dataVar?: string

) {}

public readonly validate: (rawData: unknown) => T = (rawData) => {

const data = lo.cloneDeep(rawData);

const valid = this.\_validator(data);

if (!valid) {

if (!this.\_validator.errors) throw new Error("что-то пошло не так");

throw new ValidationError(this.\_validator.errors, this.\_dataVar);

}

return data as T;

};

}

Листинг Б.11 — Файл «/src/Controller/Html2ImgController.ts»

import { JSONSchemaType } from "ajv";

import { ImgType, Server } from "../../types";

import { Browser } from "../Classes/Browser/Browser";

import { ScreenshotTask } from "../Classes/Browser/ScreenshotTask";

import { ImageList } from "../Classes/Image/ImageList";

import { Controller } from "../Classes/Server/Controller";

import { Validation } from "../Classes/Validation/Validation";

export class Html2imgController extends Controller<

Params,

Query,

Body,

Response

> {

constructor(

private readonly \_browser: Browser,

private readonly \_validation: Validation

) {

super();

}

\_controller: ControllerType = async (req, res, next) => {

try {

const { returnImgType } = this.\_paramsValidator.validate(req.params);

const opts = this.\_queryValidator.validate(req.query);

const data = this.\_bodyValidator.validate(req.body);

const imgs = new ImageList(req.files || {});

const screenshotTask = new ScreenshotTask(data, imgs, {

...opts,

returnImgType,

});

const returnImg = await this.\_browser.screenshot(screenshotTask);

// \* side effect

imgs.removeImgs();

return res.end(returnImg);

} catch (error) {

return next(error);

}

};

private readonly \_paramsValidator = this.\_validation.generateValidator(

paramsVS,

"params"

);

private readonly \_queryValidator = this.\_validation.generateValidator(

queryVS,

"query"

);

private readonly \_bodyValidator = this.\_validation.generateValidator(

bodyVS,

"body"

);

}

/\* ------------------------------- Validation ------------------------------- \*/

const paramsVS: JSONSchemaType<Params> = {

type: "object",

properties: {

returnImgType: { type: "string", enum: ["png", "jpeg", "webp"] },

},

required: ["returnImgType"],

};

const queryVS: JSONSchemaType<Query> = {

type: "object",

properties: {

width: { type: "integer", nullable: true, default: 1024 },

height: { type: "integer", nullable: true },

quality: { type: "integer", nullable: true, minimum: 1, maximum: 100 },

omitBackground: { type: "boolean", nullable: true },

},

};

const bodyVS: JSONSchemaType<Body> = {

type: "object",

properties: {

html: { type: "string" },

css: { type: "string", nullable: true },

},

required: ["html"],

};

/\* ------------------------------ / Validation ------------------------------ \*/

/\* ---------------------------------- Types --------------------------------- \*/

interface Params {

readonly returnImgType: ImgType;

}

interface Query {

readonly width?: number;

readonly height?: number;

readonly quality?: number;

readonly omitBackground?: boolean;

}

interface Body {

readonly html: string;

readonly css?: string;

}

interface Response {}

type ControllerType = Server.ControllerType<Params, Query, Body, Response>;

/\* --------------------------------- / Types -------------------------------- \*/

Листинг Б.12 — Файл «/src/Middlewares/ErrorHandlingMiddleware.ts»

import express from "express";

import { Server } from "../../types";

import { ApiError } from "../Classes/Error/ApiError";

import { BrowserError } from "../Classes/Error/BrowserError";

import { ValidationError } from "../Classes/Error/ValidationError";

import { Middleware } from "../Classes/Server/Middleware";

import { Validation } from "../Classes/Validation/Validation";

export class ErrorHandlingMiddleware extends Middleware {

constructor(private readonly \_validation: Validation) {

super();

}

protected \_controller: Server.ControllerType = ((

error: unknown,

req: express.Request,

res: express.Response,

next: express.NextFunction

) => {

if (error instanceof ApiError) {

return res

.status(error.status)

.json({ status: error.status, message: error.message })

.end();

}

if (error instanceof ValidationError) {

return res

.status(400)

.json({

status: 400,

message:

this.\_validation.getErrorMessage([error.info[0]], error.dataVar) ||

"что-то пошло не так",

})

.end();

}

if (error instanceof BrowserError) {

return res

.status(400)

.json({ status: 400, message: error.message })

.end();

}

return res

.status(500)

.json({ status: 500, message: "Internal Server Error" })

.end();

}) as unknown as Server.ControllerType;

}

Листинг Б.13 — Файл «/src/routes/router.ts»

import { ApiMethods } from "../../types/enums";

import { Browser } from "../Classes/Browser/Browser";

import { Endpoint } from "../Classes/Server/Endpoint";

import { Router } from "../Classes/Server/Router";

import { Validation } from "../Classes/Validation/Validation";

import { Html2imgController } from "../controllers/Html2imgController";

export class MainRouter extends Router {

constructor(

private readonly \_browser: Browser,

private readonly \_validation: Validation

) {

super("/");

this.useEndpoints();

}

protected \_endpoints: readonly Endpoint[] = [

new Endpoint(

ApiMethods.POST,

"/:returnImgType",

[],

new Html2imgController(this.\_browser, this.\_validation)

),

];

}