НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«До захисту допущено»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2020 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «gRPC-клієнт для відлагодження серверного програмного забезпечення»

Виконав:
студент IV курсу, групи КП-61
Рухайло Павло Олегович _____

Керівник:
Доцент кафедри ПЗКС, к.т.н., доцент
Онай Микола Володимирович _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент
Онай Микола Володимирович _____

Рецензент:
Доцент кафедри СПіСКС, к.т.н., доцент
Клятченко Ярослав Михайлович _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

«ЗАТВЕРДЖУЮ»

Науковий керівник кафедри

_____ Іван ДИЧКА

«___»_____2019 р.

ЗАВДАННЯ

на дипломний проєкт студенту

Рухайлу Павлу Олеговичу

1. Тема проєкту «gRPC-клієнт для відлагодження серверного програмного забезпечення», керівник проєкту Онай Микола Володимирович, доцент кафедри ПЗКС, к.т.н., доцент, затверджені наказом по університету від «25» травня 2020 р. № 1181-с.

2. Термін подання студентом проєкту «16» червня 2020 р.

3. Вихідні дані до проєкту: див. Технічне завдання.

4. Зміст пояснювальної записки:
   - аналіз існуючих рішень;
   - аналіз мов програмування та технологій розроблення;
   - розроблення програмних засобів;
   - аналіз розроблених програмних засобів.

5. Перелік обов'язкового графічного матеріалу:
   - схема композиції інтерфейсу користувача (креслення);
   - діаграма прецедентів (креслення);
   - архітектура прикладного програмного забезпечення (плакат);
   - структура вікна проведення запитів (плакат).

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент		

7. Дата видачі завдання «31» жовтня 2019 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення особливостей технології gRPC	21.10.2019	
2.	Аналіз існуючих рішень	18.11.2019	
3.	Аналіз технологій розробки	09.12.2019	
4.	Розроблення та узгодження технічного завдання	23.12.2019	
5.	Розроблення архітектури програмного забезпечення	06.01.2020	
6.	Розроблення дизайну інтерфейсів	27.01.2020	
7.	Програмна реалізація дипломного проєкту	23.03.2020	
8.	Тестування програмного забезпечення	20.04.2020	
9.	Підготовка матеріалів текстової частини проєкту	04.05.2020	
10.	Оформлення технічної документації проєкту	18.05.2020	

Студент Павло РУХАЙЛО

Керівник проєкту Микола ОНАЙ

АНОТАЦІЯ

Даний дипломний проєкт присвячений розробленню клієнта для відлагодження серверного програмного забезпечення, що використовує технологію gRPC для транспортування даних.

У роботі виконано аналіз існуючих рішень для відлагодження серверного програмного забезпечення, що побудовано на базі технології gRPC, порівняння можливих типів програмного забезпечення до розробки та обґрунтування вибору прикладного програмного забезпечення. Також наведено порівняльний аналіз технологій розробки прикладного програмного забезпечення та фреймворків для створення графічного інтерфейсу користувача з використанням web-технологій, обґрунтування їх вибору. Розглянуто можливі методи та стратегії використання технології gRPC, а також аналіз доступних для цього бібліотек. Розроблений клієнт надає розробникам та тестувальникам серверного програмного забезпечення на базі технології gRPC проводити процеси аналізу коректності роботи системи та її відлагодження. Для роботи програми відбувається аналіз вмісту обраного користувачем proto-файлу задля пошуку реалізованих віддаленим сервером сервісів та отримання інформації про типи даних, що використовуються, для налаштування подальших процесів валідації даних, введених користувачем, їх серилізації та десерилізації під час комунікації. Користувачеві також надається механізм для підтримки існуючого каналу зв’язку та його автоматичного поновлення при виникненні помилок.

В даному проєкті розроблено: архітектуру програмного забезпечення, механізм аналізу proto-файлів, механізм виклику віддаленого методу без прив’язки до сервісу,механізм автоматичного відновлення з’єднання з віддаленим сервісом, а також графічні елементи та дизайн графічного середовища програмного забезпечення.

ABSTRACT

This diploma project is dedicated to the development of client for debugging server-side software that uses gRPC technology to transport data.

The paper analyzes the existing solutions for debugging server-side software, built on the basis of gRPC technologies, comparing possible types of software for development and justification of the choice of application software. There is also a comparative analysis of application software development technologies and frameworks for creating a graphical user interface using web-technologies, justification of their choice. Possible methods and strategies for using gRPC technology are considered, as well as an analysis of available libraries. The developed client provides developers and testers of server software based on gRPC technology to conduct processes of analysis of the correctness of the system and its debugging. The program analyzes the contents of the user-selected proto-file to search for services implemented by the remote server and obtain information about the types of data used to configure further processes of validation of data entered by the user, their serialization and deserialization during communication. The user is also provided with mechanisms for keeping alive the existing communication channel and automatically reconnection in case of any errors.

During the work on this project the next parts has been developed: software architecture, mechanism for analyzing proto-files, mechanism for calling a remote method without binding to the service, mechanism for automatically reconnecting to a remote service, as well as graphic elements and design of the graphical software environment.

АННОТАЦИЯ

Данный дипломный проект посвящён разработке клиента для отладки серверного программного обеспечения, использующего технологию gRPC для транспортировки данных.

В работе выполнен анализ существующих решений для отладки серверного программного обеспечения на базе технологий gRPC, сравнение возможных типов программного обеспечения к разработке и обоснование выбора прикладного программного обеспечения. Также приведён сравнительный анализ технологий разработки прикладного программного обеспечения и фреймворков для создания графического интерфейса пользователя с использованием web-технологий, обоснование их выбора. Рассмотрены возможные методы и стратегии использования технологии gRPC, а также анализ доступных для этого библиотек. Разработанный клиент предоставляет разработчикам и тестировщикам серверного программного обеспечения на базе технологии gRPC проводить процессы анализа корректности работы системы и ее отладки. Для работы программы происходит анализ содержания выбранного пользователем proto-файла для поиска реализованных удаленным сервером сервисов и получения информации о типах данных, используемых для настройки дальнейших процессов валидации данных, введенных пользователем, их серилизации и десерилизации во время коммуникации. Пользователю также предоставляется механизм для поддержания существующего канала связи и его автоматического обновления при возникновении ошибок.

В данном проекте разработаны: архитектуру программного обеспечения, механизм анализа proto-файлов, механизм вызова удаленного метода без привязки к сервису, механизм автоматического восстановления соединения с удаленным сервисом, а также графические элементы и дизайн графической среды программного обеспечения.

ДП.045440-01-90 gRPC-клієнт для відлагодження серверного програмного забезпечення. Відомість проєкту

Позначення	Найменування	Кіл-ть	Примітка
	Документація проєкту		
ДП.045440-02-91	gRPC-клієнт для	4	
	відлагодження серверного		
	програмного забезпечення.		
	Технічне завдання		
ДП.045440-03-81	gRPC-клієнт для	64	
	відлагодження серверного		
	програмного забезпечення.		
	Пояснювальна записка		
ДП.045440-04-51	gRPC-клієнт для	4	
	відлагодження серверного		
	програмного забезпечення.		
	Програма та методика		
	тестування		
ДП.045440-05-34	gRPC-клієнт для	11	
	відлагодження серверного		
	програмного забезпечення.		
	Керівництво користувача		
ДП.045440-06-99	gRPC-клієнт для	1	
	відлагодження серверного		
	програмного забезпечення.		
	Інтерфейс користувача.		
	Схема композиції		

Позначення	Найменування	Кіл-ть	Примітка
ДП.045440-07-99	gRPC-клієнт для	1	
	відлагодження серверного		
	програмного забезпечення.		
	Взаємодія користувача з		
	програмою. Діаграма		
	прецедентів		
ДП.045440-08-98	gRPC-клієнт для	1	
	відлагодження серверного		
	програмного забезпечення.		
	Компакт-диск		

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2019 р.

GRPC-КЛІЄНТ ДЛЯ ВІДЛАГОДЖЕННЯ СЕРВЕРНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Технічне завдання

ДП.045440-02-91

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Микола ОНАЙ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Павло РУХАЙЛО

2019

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: gRPC-клієнт для відлагодження серверного програмного забезпечення.

Галузь застосування: інформаційні технології.

2. ПІДСТАВА ДЛЯ РОЗРОБЛЕННЯ

Підставою для розроблення є завдання на дипломне проєктування, затверджене кафедрою програмного забезпечення комп’ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» (КПІ ім. Ігоря Сікорського).

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для використання розробниками та тестувальниками серверного програмного забезпечення, що побудоване на базі технології gRPC для використання у процесах аналізу коректності роботи сервісів та їх відлагодження.

4. ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

Програмний продукт повинен забезпечувати такі основні функції:

1) аналіз вмісту proto-файлів;
2) надання користувачеві детальної інформації про типи даних та сервіси, що визначені у proto-файлах;
3) пакування даних у відповідні бінарні формати залежно від їх типу та визначень;
4) надання користувачеві можливості інтерактивної роботи з усіма видами запитів, що існують в рамках технології gRPC, а саме унарними, односторонньо-стрімінгові як зі сторони клієнта, так і зі сторони серверу, а також двосторонньо-стрімінговимими;
5) підтримка «well-known» типів даних;
6) автоматичне поновлення з’єднання з віддаленим сервером.

5. ВИМОГИ ДО ПРОЄКТНОЇ ДОКУМЕНТАЦІЇ

У процесі виконання проєкту повинна бути розроблена наступна документація:

1) пояснювальна записка;

2) програма та методика тестування;

3) керівництво користувача.

6. ЕТАПИ ПРОЄКТУВАННЯ

Вивчення особливостей технології gRPC.....21.10.2019

Аналіз існуючих рішень.....18.11.2019

Аналіз технологій розробки.....09.12.2019

Розроблення та узгодження технічного завдання.....23.12.2019

Розроблення архітектури програмного забезпечення.....06.01.2020

Розроблення дизайну інтерфейсів.....27.01.2020

Програмна реалізація дипломного проєкту.....23.03.2020

Тестування програмного забезпечення.....20.04.2020

Підготовка матеріалів текстової частини проєкту.....04.05.2020

Оформлення технічної документації проєкту.....18.05.2020

7. ПОРЯДОК ТЕСТУВАННЯ РОЗРОБКИ

Тестування розробленого програмного продукту виконується відповідно до «Програми та методики тестування».

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2020 р.

GRPC-КЛІЄНТ ДЛЯ ВІДЛАГОДЖЕННЯ СЕРВЕРНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Пояснювальна записка

ДП.045440-03-81

«ПОГОДЖЕНО»

Керівник проєкту:

_____ Микола ОНАЙ

Нормоконтроль:

_____ Микола ОНАЙ

Виконавець:

_____ Павло РУХАЙЛО

2020

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

ПЗ – програмне забезпечення.

Web-браузер – програмне забезпечення для перегляду та взаємодії з гіпертекстовими сторінками в мережі Інтернет.

Парадигма програмування – система ідей і понять, що задають правила написання програм, їх структурування та організацію обчислень.

HTTP – Hyper Text Transfer Protocol – протокол передачі гіпер-текстових документів.

HTTP/2 – друга версія протоколу HTTP.

CLI – Command-Line Interface – текстовий інтерфейс користувача, при якому інструкції пристрою надаються шляхом вводу певних команд.

GUI – Graphical User Interface – тип інтерфейсу, при якому користувач може взаємодіяти з пристроєм за допомогою графічних забражень та вказівок.

Фреймворк – програмна платформа, яка визначає структуру програмної системи.

TLS – Transport Layer Security – криптографічний протокол передачі даних, що використовує асиметричне шифрування інформації.

JSON – JavaScript Object Notation – текстовий формат передачі інформації.

JAR – Java ARchive – спеціальний формат архівації даних, що використовується для пакування програм мовою програмування Java.

HTML – HyperText Markup Language – мова розмітки гіпертекстових документів.

Хук – спеціальні функції, що дозволяють зберігати та взаємодіяти зі станом функціональних компонентів при використанні фреймворку React без написання класів.

XAML – eXtensible Application Markup Language – декларативна мова програмування, що використовується для налаштування GUI-інтерфейсів.

CSS – Cascading Style Sheets – спеціальна мова, що використовується для налаштування зовнішнього вигляду елементів.

QML – Qt Meta Language або Qt Modeling Language – декларативна мова програмування, що призначена для розробки програм з графічним інтерфейсом користувача. Є частиною середовища розробки Qt Quick, що входить до фрейморку Qt.

JSX – спеціальний синтаксис, схожий на XML. Використовується для анотації React-компонентів. Компілюється у текст мовою програмування JavaScript.

XML – Extensible Markup Language – стандарт побудови мов розмітки ієрархічно структурованих даних; розширювана мова розмітки даних.

ВСТУП

Історично склалося, що типовою архітектурою для розробки веб-сервісів є монолітна. Цьому сприяє певний ряд факторів, таких як прості процеси розробки, тестування та розгортання.

Впродовж останніх десятиліть вимоги до веб-сервісів зі сторони користувачів стають все більшими, що спричиняє розширення їх функціональних можливостей. Використання монолітної архітектури стає менш доцільним, адже перш за все підтримка такого ПЗ стає все більш складним завданням. Причиною цього є необхідність для розробників розуміння принципів роботи всієї системи загалом, адже певні, навіть досить незначні, зміни в одній частині ПЗ можуть спричинити непередбачені зміни поведінки в інших його частинах. Також однією з проблем, які можуть вплинути на процес розробки ПЗ, є можливе нерівномірне використання апаратних ресурсів окремими модулями. Масштабування всієї системи вирішує проблему нестачі ресурсів для більш ресурсомістких модулів, проте спричиняє простій ресурсів, виділених для менш ресурсомістких модулів.

Одним із можливих рішень проблем, пов’язаних із ростом монолітних систем, є використання мікросервісної архітектури. Розробка веб-сервісів з використанням даної архітектури полягає у розбитті бізнес-логіки сервісу на окремі частини, кожна з яких розробляється, розгортається та підтримується окремо.

Проте не дивлячись на очевидні переваги даного підходу розробники ПЗ стикаються з певною кількістю нових проблем. Однією з них є комунікація між окремими частинами системи.

Існує велика кількість технологій, створених для вирішення конкретно цієї проблеми. Одна з таких – gRPC.

gRPC (gRPC Remote Procedure Call) – це фреймворк з відкритим сирцевим кодом для виклику віддалених процедур, розроблений

компанією Google. Використовує протокол HTTP/2 як транспорт. В наш час даний фреймворк активно використовується саме для налагодження комунікацій між сервісами однієї системи.

При передачі даних використовуються бінарні формати. При виконанні запиту для того, щоб правильно серилізувати дані з однієї сторони та правильно десерилізувати з іншої, використовується технологія Protocol Buffers. Технологія використовує спеціальні proto-файли для опису типів даних, а також для опису методів, що можна використовувати.

Даний дипломний проєкт присвячено розробці клієнта для відлагодження та тестування розробником серверного програмного забезпечення, побудованого на базі технології gRPC.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

На сьогоднішній день існує певна кількість gRPC-клієнтів, призначених для інспектування та відлагодження серверного програмного забезпечення.

Знайдене програмне забезпечення є продуктами з відкритим сирцевим кодом, які ліцензовані відповідно до «Apache License 2.0», «BSD 3-Clause «New» or «Revised» License», «MIT License» або «GNU Lesser General Public License v3.0». Дані ліцензії дозволяють вільно використовувати (як персонально, так і в комерційних цілях), модифіковувати та розповсюджувати програмне забезпечення в цілому, або ж певні його частини, за умови повідомлення про ліцензування та авторські права (а також посилання на сирцевий код у випадку останньої). Проте вони не надають жодних гарантій щодо коректної роботи ПО, оскільки звільняють розробників від будь-якої відповідальності за результати своєї роботи [1, 2].

Знайдені існуючі рішення можна розбити виходячи з типу наданого інтерфейсу користувача на 2 категорії:

- CLI-орієнтовані клієнти;
- GUI-орієнтовані клієнти.

Розбиття саме за таким критерієм є суттєвим, оскільки різні типи інтерфейсу надають кінцевому користувачеві принципово різний досвід користування, а також певним чином визначають область використання програмного забезпечення.

Нижче буде розглянуто та проаналізовано знайдені існуючі рішення.

1.1. Аналіз CLI-орієнтованих клієнтів

gcall – дуже простий клієнт. Надає мінімальний набір функціональних властивостей для виконання окремих запитів.

Переваги:

- інформація для передачі може надаватися як у списку аргументів, так і через канал читання програми;
- відображення відповіді на запити в різних форматах.

Недоліки:

- немає підтримки захищеного протоколом TLS з’єднання;
- не надає можливості переглядати методи сервісів та типи вхідних і вихідних даних;
- потокові відповіді з сервера можна переглянути лише після завершення запиту, а не в міру надходження окремих частин;
- обмежена підтримка «well-known» типів даних;
- обмежена робота з зовнішніми імпортами в proto-файлах.

grpcurl – більш розширений в порівнянні з gcall клієнт. Крім виконання окремих запитів, надає певні функціональні можливості для аналізу proto-файлів.

Переваги:

- інформація для передачі може надаватися як у списку аргументів, так і через канал читання програми;
- надає можливість переглядати методи сервісів та типи вхідних та вихідних даних;
- можливість задання базових директорій для пошуку імпортованих proto-файлів.

Недоліки:

- потокові відповіді з сервера можна переглянути лише після завершення запиту, а не в міру надходження окремих частин;
- обмежена підтримка «well-known» типів даних;
- вхідні та вихідні дані можуть бути лише в форматі JSON.

polyglot – клієнт, дуже схожий до попереднього за функціональними можливостями.

Переваги:

- повна підтримка «well-known» типів даних;
- підтримка відкритого стандарту авторизації OAuth.

Недоліки:

- потокові відповіді з сервера можна переглянути лише після завершення запиту, а не в міру надходження окремих частин;
- програмне забезпечення розповсюджується як JAR-файл. Для запуску необхідно встановлювати середовище виконання Java.

grpcc – клієнт, що має вигляд інтерактивного середовища програмування. При запуску користувач потрапляє у середовище програмування Node.js, в якому через глобальні наперед визначені та створені під час аналізу proto-файлів змінні він може взаємодіяти з сервісами.

Переваги:

- інтерактивна робота з потоковими запитами та відповідями;
- повна підтримка «well-known» типів даних;
- автоматичне відновлення з’єднання з сервером;
- можливість підключення через локальний сокет.

Недолік: обмежена робота з зовнішніми імпортами в proto-файла.

evans – клієнт з найбільш широкими функціональними можливостями в порівнянні з попередніми. Може бути запущений як для виконання конкретного запиту, так і для інтерактивної роботи з сервісом.

Переваги:

- повна підтримка «well-known» типів даних;
- інтерактивна робота з потоковими запитами та відповідями;
- надає функціональні можливості для глибокого аналізу сервісів та типів даних;
- можливість задання базових директорій для пошуку імпортованих proto-файлів.

В порівнянні з наведеними вище клієнтами не має вагомих недоліків.

1.2. Аналіз GUI-орієнтованих клієнтів

Yodelay – програмне забезпечення, що має клієнт-серверну архітектуру. Для свого функціонування програма запускає 2 процеси: сервіс інтерфейсу користувача та сервіс комунікації з віддаленим сервером. Доступ до інтерфейсу користувача відбувається через Інтернет-браузер. Другий сервіс запускається як фоновий процес. Комунікація між сервісами відбувається за протоколом HTTP. Також під час ініціалізації програми запускається ще один сервіс, який є демонстраційним сервером, що побудований на базі технології gRPC.

Переваги:

- повна підтримка «well-known» типів даних;
- є можливість відміни запитів.

Недоліки:

- програма запускається як три окремі сервіси, при цьому один з них є лише демонстраційним і не несе жодного корисного навантаження;
- доступ до інтерфейсу користувача відбувається через Інтернет-браузер;
- робота з потоковими методами не є інтерактивною;
- в proto-файлах підтримується лише імпорт «well-known» типів даних.

grpcox – клієнт, що має схожу з попереднім архітектуру. Проте, на відміну від Yodelay, запускається як один процес. Програмне забезпечення розповсюджується як образ для середовища Docker.

Переваги:

- автоматичне відновлення з’єднання з сервером;
- повна підтримка «well-known» типів даних.

Недоліки:

- підтримка лише унарних запитів;

- для запуску клієнта необхідні певні знання та досвід роботи з середовищем Docker;
- в proto-файлах підтримується лише імпорт «well-known» типів даних.

letmegrpc – ПО, що не є повноцінним клієнтом для відлагодження, а представляє з себе розширення для утиліти генерації коду з proto-файлів. Під час роботи зазначеної утиліти ПО генерує HTML сторінки, за допомогою яких можна викликати окремі методи.

Переваги:

- повна підтримка «well-known» типів даних;
- згенеровані HTML сторінки надають досить багато інформації про окремі запити;
- широкі можливості для роботи з імпортованими файлами;
- інтерактивна робота з потоком запитів.

Недоліки:

- потокові відповіді з сервера можна переглянути лише після завершення запиту, а не в міру надходження окремих частин;
- для користування програмою необхідні навички роботи з утилітою для генерації коду з proto-файлів.

gRPC UI – програмне забезпечення, що надає найбільш широкі серед розглянутих клієнтів функціональні можливості для відлагодження серверного програмного забезпечення. Має клієнт-серверну архітектуру.

Переваги:

- інтерфейс, що надає можливість широкого аналізу усіх типів даних, що використовуються;
- повна підтримка «well-known» типів даних.

Недоліки:

- доступ до інтерфейсу користувача відбувається через Інтернет-браузер;
- робота з потоковими методами не є інтерактивною.

Delivery for gRPC та BloomRPC – єдині зі знайдених існуючих рішень клієнти, що представляють з себе повноцінне прикладне програмне забезпечення. Мають між собою подібні функціональні можливості. Є досить зручним програмним забезпеченням для відлагодження серверного програмного забезпечення, оскільки надають користувачеві можливість працювати з потоковими запитами та відповідями в інтерактивному форматі. Проте, в свою чергу, не надають достатньо можливостей для аналізу типів даних, оскільки всі дані відображаються в форматі JSON.

Переваги:

- повноцінне прикладне програмне забезпечення.
- зручна робота з налаштуваннями зовнішніх імпортів в proto-файлах.
- повністю інтерактивна робота з потоковими методами.
- повна підтримка «well-known» типів даних.

Недолік: робота з типами даних відбувається у форматі JSON.

1.3. Обґрунтування основних функціональних властивостей програмного продукту проєкту

Виходячи з аналізу існуючих рішень для налагодження серверного програмного забезпечення, побудованого на базі технології gRPC, можна виділити певну множину функціональних властивостей, обов’язкову до реалізації та підтримки в рамках програми проєкту.

Клієнт повинен мати функціональні можливості для читання proto-файлів з файлової системи, а також їх парсингу задля отримання опису сервісів та типів даних, що використовуються. При цьому, оскільки proto-файли можуть містити в собі імпортування інших файлів, необхідно надати користувачеві можливість вибору базових директорій для пошуку імпортованих даних.

Користувач програмного забезпечення повинен мати можливість перегляду інформації про сервіси, що містяться в обраних proto-файлах, та

їх методи, а також детальну інформації про типи даних, що використовуються.

Інформація про типи даних, описана вище, повинна використовуватися для валідації усіх даних, введених користувачем програми, для їх подальшого успішного пакування у бінарний формат та відправки на сервер. А також використовуватися для розпакування даних, отриманих від серверу.

Задля повної підтримки можливостей технології, необхідно, щоб користувач мав можливість налагоджувати всі види запитів, що існують в рамках технології gRPC, а саме унарні запити, односторонньо-стрімінгові запити як зі сторони клієнта, так і зі сторони серверу, а також двосторонньо-стрімінгові запити. При цьому дуже важливим є інтерактивна робота із стрімінговими запитами.

Оскільки однією з переваг даної технології є використання одного з’єднання для виконання запитів, необхідно, щоб клієнт мав функціональні можливості для підтримання життя з’єднання з сервером, а також його автоматичного повторного створення у разі виникнення проблем.

1.4. Висновки

Аналіз існуючих рішень показав, що клієнти для відлагодження серверного програмного забезпечення, яке побудоване на базі технології gRPC, загалом являють собою проєкти з відкритим сирцевим кодом та розповсюджуються на безоплатній основі.

Встановлено, що таке програмне забезпечення може надавати користувачеві для роботи як інтерфейс командного рядку, так і графічне середовище. В якості CLI-орієнтованих клієнтів було розглянуто такі програми, як gcall, grpcurl, polyglot, grpcc та evans. В свою чергу, розглянуто наступні клієнти з GUI-орієнтованим інтерфейсом користувача: Yodelay, grpcox, letmegrpc, gRPC UI, Delivery for gRPC та BloomRPC.

Аналіз переваг та недоліків кожної з цих програм показав, що загалом існує певний перелік ключових функціональних можливостей, необхідних кінцевому користувачеві для процесу відлагодження. Проте встановлено, що відсутнє програмне забезпечення, яке реалізовує принаймні половину цих функціональних можливостей. Також визначено, що недоліки деяких клієнтів пов’язані не з переліком їх функціональних можливостей, а з особливостями розповсюдження та запуску.

Таким чином встановлено перелік основних вимог до клієнта, що розробляється:

- аналіз вмісту proto-файлів;
- надання користувачеві детальної інформації про типи даних та сервіси, що визначені у proto-файлах;
- пакування даних у відповідні бінарні формати залежно від їх типу та визначень;
- надання користувачеві можливості інтерактивної роботи з усіма видами запитів, що існують в рамках технології gRPC, а саме унарними, односторонньо-стрімінгові як зі сторони клієнта, так і зі сторони серверу, а також двосторонньо-стрімінговимими;
- підтримка «well-known» типів даних;
- автоматичне поновлення з’єднання з віддаленим сервером;
- простий процес запуску;
- розповсюдження як єдиного програмного забезпечення, що не залежить від середовища роботи.

2. АНАЛІЗ МОВ ПРОГРАМУВАННЯ ТА ТЕХНОЛОГІЙ РОЗРОБЛЕННЯ

2.1. Обґрунтування вибору прикладної програми як типу програмного забезпечення

Як було вже зазначено в попередньому розділі, знайдені існуючі рішення можна розбити на дві категорії виходячи з типу інтерфейсу користувача, що надається, а саме:

- CLI-орієнтовані клієнти.
- GUI-орієнтовані клієнти.

Програмне забезпечення з консольним інтерфейсом користувача є корисним при використанні у процесах безперервної інтеграції та постійної доставки, або ж для автоматизації певних процесів.

Для використання у процесах відлагодження серверного програмного забезпечення більш відповідними інструментами є GUI-орієнтовані клієнти, оскільки вони можуть відображати більше корисної інформації та надавати більше функціональних можливостей кінцевому користувачеві. Також, як було зазначено вище, необхідно реалізувати інтерактивну роботу зі стрімінговими запитами, що можна більш зрозуміло реалізувати саме при використанні графічного інтерфейсу користувача.

Програмне забезпечення з графічним інтерфейсом користувача, виходячи з аналізу існуючих рішень, може бути орієнтованим на різний принцип роботи. Так, програма може мати вигляд окремого прикладного програмного забезпечення, або ж програмного забезпечення, що вимагає від користувача використання web-браузеру для доступу до інтерфейсу. Такий підхід не є оптимальним для використання в даному проєкті. Насамперед це пов’язано з тим, що однією з функціональних можливостей продукту проєкту є пошук імпортованих файлів виходячи з наданого користувачем списку базових директорій, що вимагає безпосереднього

доступу до файлової системи. Також це пов’язано з тим, що технологія gRPC побудована з використанням технології HTTP/2 [3]. Сучасні web-браузери підтримують лише HTTP 1.0 та HTTP 1.1 [4]. Для реалізації продукту проєкту з використанням такого підходу необхідно використовувати клієнт-серверну архітектуру для програмного забезпечення, що не є оптимальним.

Таким чином, типом програмного забезпечення, що розробляється, обрано прикладне програмне забезпечення з графічним інтерфейсом користувача.

2.2. Аналіз найбільш поширених технологій розроблення прикладного програмного забезпечення

Вибір технології розроблення прикладного програмного забезпечення є дуже важливим, оскільки в більшості випадків цей вибір визначає не тільки набір інструментів та бібліотек для розробки, а й мову програмування, що буде використовуватися.

В наш час існує велика кількість технологій, пов’язаних з розробкою прикладного програмного забезпечення. Кожна з них має як свої переваги, так і недоліки.

Для успішної реалізації програми проєкту необхідно, щоб обрана технологія задовільняла такі критерії:

1. Крос-платформність – програма повинна працювати на таких операційних системах, як Windows 10, MacOS, Linux.
2. Широкий набір як вбудованих, так і сторонніх компонентів.
3. Налаштування та адаптація зовнішнього вигляду компонентів.
4. Наявність окремої мови розмітки інтерфейсу користувача.
5. Велика спільнота розробників, що використовують дану технологію.
6. Можливість використання для створення продуктів з відкритим сирцевим кодом.

Розглянемо та проаналізуємо поширені технології розробки прикладного програмного забезпечення, а також їх відповідність визначеним вище критеріям.

Windows Forms – API, що відповідає за графічний інтерфейс користувача. Є частиною фреймворку .NET [5]. Інкапсулює в собі роботу з Win32 API, чим значно полегшує розробку. Надає досить мало можливостей для налаштування зовнішнього вигляду компонентів. При роботі з даною технологією використовуються мови програмування C, C++ або C#.

Windows Presentation Foundation – графічна підсистема, що входить до фреймворку .NET починаючи з версії 3.0 [6]. Використовує та розширює можливості Windows Forms. При роботі з даною технологією використовуються мови програмування C++ або C#, проте для побудови інтерфейсів користувача використовується мова розмітки XAML, що була розроблена саме для цієї технології [7]. Надає досить широкі можливості для налаштування зовнішнього вигляду компонентів.

Наведені вище технології мають досить широку спільноту, а також велику кількість готових компонентів, що є доступними через встановлення сторонніх бібліотек. Офіційно підтримуються лише в сімействі операційних систем Windows. Проте є можливість запуску ПО і в інших операційних системах за допомогою проєкту Mono [8] (коректна робота окремих компонентів не гарантується, а сам проєкт вже не розвивається [9]).

Universal Windows Platform – остання розробка компанії Microsoft. Є досить складною у вивченні. Через це, а також через те, що є відносно новою технологією, має невелику спільноту. Мовами програмування, що використовуються для побудови логіки програми, є C#, VB.Net та C++, а для побудови інтерфейсів користувача використовується мова програмування Javascript та мова розмітки XAML [10]. Програмне забезпечення, що побудоване за допомогою даної технології, може

працювати лише в операційних системах Windows 10, Windows 10 Mobile, а також на пристроях Xbox One.

Abstract Window Toolkit – частина стандартного API мови програмування Java; пакет класів, що використовуються для створення графічних інтерфейсів користувача [11]. Дане рішення є крос-платформним, оскільки запуск програм, написаних мовою програмування Java, Kotlin, Scala, Groovy або будь-якою іншою, що компілюється в машинний код Java, відбувається через віртуальну машину Java [12]. Не дивлячись на очевидні переваги, такий підхід не є оптимальним, оскільки змушує користувача встановлювати додаткове програмне забезпечення, а саме віртуальну машину Java. Abstract Window Toolkit є дуже обмеженим інструментом, оскільки оперує невеликою кількістю компонентів, а також надає досить мало можливостей для налаштування їх зовнішнього вигляду. Також не гарантує однакового вигляду та набору функціональних властивостей окремих елементів, оскільки візуальні компоненти перетворюються у відповідні їм еквіваленти платформи, на якій програма запущена [13]. Використання даної технології не є доцільним в рамках проєкту.

Для вирішення проблем, що існували в Abstract Window Toolkit, було розроблено пакет Swing [14]. Даний пакет використовує Abstract Window Toolkit, а отже і функціональні властивості, що надаються платформою, лише для побудови вікон програм. Відображення окремих компонентів у вікнах, як і самі компоненти, реалізовано безпосередньо в даному пакеті без перетворення у відповідні компоненти платформи [15]. Такий підхід надав розробникам пакету можливості для створення великої кількості візуальних компонентів та більш широкі можливості для налаштування їх зовнішнього вигляду. Технологія гарантує однакову поведінку елементів незалежно від платформи запуску. Проте налаштування зовнішнього вигляду компонентів є не досить зручним, оскільки налаштовується в тексті програми, а не через окремий інструментарій. Даний пакет також є

частиною стандартного API мови програмування Java, проте його розробка та підтримка офіційно зупинені. Через це використання даної технології також не є доцільним.

На заміну попереднього пакету було розроблено бібліотеку JavaFX. Не дивлячись на те, що в самій бібліотеці реалізовано меншу в порівнянні з пакетом Swing кількість компонентів, JavaFX було створено саме для розробки багатофункціональних графічний інтерфейсів, оскільки бібліотека надає широкі можливості для налаштування зовнішнього вигляду існуючих компонентів, а також створення нових [16]. Розмітку інтерфейсу користувача та налаштування окремих елементів можна реалізовувати як безпосередньо в тексті програми, так і за допомогою окремої мови стилізації сторінок CSS [17]. Технологією користується досить широке коло розробників, які, в свою чергу, створюють велику кількість бібліотек з готовими компонентами різних типів. Наразі бібліотека активно розвивається. Основним недоліком є необхідність кінцевому користувачеві встановлювати віртуальну машину Java для запуску програмного забезпечення.

GTK – крос-платформна бібліотека для створення графічних інтерфейсів користувача [18]. Надає розробнику широкий набір компонентів. Основною мовою програмування є C/C++, проте архітектурно бібліотека побудована так, що дозволяється використання мов програмування Python, Javascript, Perl, D, Rust та Go, оскільки реалізовані бібліотеки-зв’язки для цих мов [19]. Побудова інтерфейсу користувача може відбуватися як безпосередньо в коді програми, так і візуально, за допомогою програми-будівника інтерфейсу Glade, що значно спрощує та прискорює процес розробки програмного забезпечення [20]. Дана бібліотека дозволяє будувати як прості, так і досить складні програми. Проте технологія не надає можливостей для налаштування зовнішнього вигляду окремих компонентів. Бібліотека побудована для створення програм, що будуть мати однаковий візуальний вигляд при

певних налаштуваннях середовища, в якому вони виконуються. Такі графічні оточення робочого столу операційних систем на базі ядра Linux, як Gnome та Xfce, надають користувачам можливість налаштування системної графічної теми. В таких середовищах програми, побудовані за допомогою бібліотеки GTK будуть виглядати досить гармонічно, оскільки всі матимуть однаковий вигляд. Проте для стилізації цих додатків в інших середовищах, від користувача вимагається досить складний процес налаштування оточення [21, 22]. Дана технологія не є оптимальною для використання в рамках даного проєкту саме через неоднозначність вигляду кінцевого продукту.

Qt – набір інструментарію, що створений для розробки крос-платформного програмного забезпечення [23]. Дозволяє створювати досить широкий спектр програмного забезпечення. Загалом, існує чотири редакції фреймворку:

1. Qt Console – інструментарій для розробки додатків без графічного інтерфейсу. Прикладами таких програм є мережні демони, програмне забезпечення з консольним інтерфейсом користувача, тощо.
2. Qt Desktop – версія пакету для розробки програмного забезпечення з графічним інтерфейсом. Містить велику кількість компонентів інтерфейсу, а також певний набір модулів для роботи з базами даних різного типу, мережевими технологіями, різними форматами передачі та збереження даних і так далі.
3. Qt Desktop Light – полегшена версія попереднього пакету. На відміну від повної версій, складається лише з компонентів графічного інтерфейсу користувача.
4. Qt Open Source Edition – версія пакету для розробки відкритого програмного забезпечення.

Редакції, що використовуються для створення графічних інтерфейсів, мають велику кількість компонентів та широкі можливості для їх стилізації. Основною мовою програмування є певний діалект мови C++, який перед компіляцією додатково оброблюється через Meta Object Compiler. Даний діалект вводить до мови такі поняття як слоти та сигнали. Сам Meta Object Compiler входить до пакету інструментарію. Також до нього входять середовище програмування QtCreator та модуль для створення графічних інтерфейсів QtDesigner [24]. Графічний інтерфейс користувача можна налаштовувати як за допомогою основної мови програмування, так і за допомогою декларативної мови програмування QML з можливими вставками Javascript коду для реалізації певної логіки компонентів. Даний інструментарій є гарним вибором для створення програмного забезпечення будь-якого рівня складності. Проте існують певні нюанси при використанні Qt для розробки продуктів з відкритим сирцевим кодом. Перш за все це пов’язано з тим, що редакція пакету, що може бути використана (Qt Open Source Edition) є певною мірою обмеженою в порівнянні з комерційними версіями (Qt Desktop Light або Qt Desktop). Також від розробників проєктів з відкритим сирцевим кодом вимагається повне підпорядкування ліцензії «GNU General Public License v3» або «GNU Lesser General Public License v3», що тягне за собою певний спектр робіт. Власники технології, The Qt Company, в даній ситуації залишають за собою право на перевірку дотримання розробниками умов однієї з перелічених вище ліцензій [25].

Усі зазначені вище технології використовуються для розробки нативного програмного забезпечення. Так, текст програми компілюється окремо для кожної окремої платформи.

Останнім часом через популярність web-технологій та їх активну розробку велика кількість розробників використовують саме web-технологій для створення прикладного програмного забезпечення. Для цього використовуються проєкти NW.js та Electron. Дані проєкти мають

майже ідентичний принцип роботи та надають розробникам схожі можливості, тому розглядатимуться разом. При розробленні використовується мова програмування Javascript, або будь-яка мова програмування, що компілюється в Javascript, такі як CoffeeScript, Dart, Typescript та інші. Для побудови інтерфейсів використовується мова гіпертекстової розмітки HTML, а для стилізації компонентів – мова стилю сторінок CSS, або будь-яка інша, що інтерпретується як CSS, такі як Sass або Less. Принцип роботи полягає у запуску програмного забезпечення за допомогою спеціальної версії web-браузеру Chromium з доступом до середовища Node.js. Таким чином, побудова та відображення інтерфейсу користувача відбувається за допомогою web-технологій, а для таких операцій, як доступ до файлової системи користувача або до мережі, використовується API середовища виконання Node.js [26]. Також Node.js надає розробникам найбільшу кількість сторонніх бібліотек, доступних для використання при розробленні. Встановлення додаткових пакетів відбувається через утиліту Node Package Manager. На момент написання даного тексту доступно близько 1,25 мільйона бібліотек [27], переважна кількість яких є проєктами з відкритим сирцевим кодом. Дані технології дозволяють створювати прикладне програмне забезпечення будь-якого рівня складності, а також виконувати збірку програми для різних платформ. Єдиним недоліком є те, що при збірці до проєкту включаються web-браузер Chromium та середовище Node.js, що негативно впливає на вихідний розмір програмного забезпечення. Так, мінімальний розмір сягає 30-ти мегабайт. Проте, враховуючи характеристики сучасних комп’ютерів, даний недолік не є критичним.

Основною різницею між технологіями NW.js та Electron є точка запуску програми [28]. Так, для першої технології точкою запуску додатка є HTML-файл, який запускає скрипти логіки програми, котрі, в свою чергу, можуть створювати додаткові вікна. Усі вікна мають доступ до спільного контексту середовища Node.js. Для другої технології точкою запуску є

скриптовий файл, що відповідає за контроль роботи окремих вікон, а кожне окреме вікно вже відображає певний HTML-файл зі своїми скриптами. Процеси, що відповідають за роботу окремих вікон та початкового скрипта, мають власні контексти виконання. Виходячи з цього, технологію NW.js краще використовувати для багатовіконних програм, а проєкт Electron – для програм з невеликою кількістю окремих вікон, або для програм з незалежними вікнами, оскільки необхідно окремо налаштовувати комунікацію процесів через API, що надається проєктом.

Для реалізації програмного забезпечення в рамках дипломного проєкту обрано технологію Electron через широку спільноту розробників, велику кількость сторонніх бібліотек та можливостей, що надаються для побудови графічного інтерфейсу користувача.

2.3. Аналіз мови програмування Typescript

Мова програмування Typescript не є самостійною мовою, а є розширенням мови Javascript, яка, в свою чергу, є однією з реалізацій стандарту ECMAScript. Тому для детального аналізу мови Typescript необхідно спочатку проаналізувати ECMAScript та Javascript.

ECMAScript – це специфікація скриптової мови програмування загального призначення, визначена в стандарті ECMA-262, що розробляється некомерційною асоціацією європейських виробників комп'ютерів ECMA International [29].

Javascript – найпоширеніша реалізація специфікації ECMAScript. Класифікується як прототип-орієнтована динамічна мова, що має декілька парадигм та підтримує об'єктно-орієнтований, імперативний та декларативний (тобто функціональне програмування) стилі [30]. Однак, слід зазначити, підтримка об'єктів в даній мові програмування відрізняється від традиційних мов ООП, оскільки використовується концепція прототипів. Синтаксис мови є схожим до мов програмування C/C++ та Java.

Примітивні типи даних, що визначені в мові [31]:

- числовий: відповідає 64-бітному формату чисел з плаваючою точкою, що визначений стандартом IEEE 754-2008, за винятком спеціального значення Not-a-Number, Infinity та -Infinity;
- строковий;
- логічний;
- нульовий;
- невизначений.

Слід зазначити, що нульовий та невизначений типи даних не є тривіальними, оскільки кожен з них визначає лише одне значення.

Також в мові визначено такий непримітивний тип даних, як об’єктний.

Ключовими особливостями мови програмування Javascript є:

- різні підходи для створення об’єктів;
- функції як об'єкти першого класу;
- контроль області видимості змінних;
- замикання контексту функцій;
- два окремих механізми порівняння рівності: строга та нестрога (остання в низці випадків призводить до перетворення типів для визначення результату);
- можливість інтроспекції об’єктів;
- використання механізму прототипів для динамічної зміни типу об’єкту;
- підтримка анонімних класів та функції;
- відсутність перевантаження функцій;
- механізми обробки виняткових ситуацій та помилок виконання;
- автоматичне приведення типів;
- два шляхи створення регулярних виразів: літеральний та об’єктний;
- автоматична робота з пам’яттю.

Текст програм, написаних даною мовою програмування, інтерпретується та компілюється під час виконання. Програми, написані даною мовою, можуть бути виконані в великій кількості середовищ. Враховуючи цей факт, а також динамічну типізацію мови, слід бути уважним при роботі з мовою Javascript, оскільки існує ймовірність неочікуваної поведінки через певні конфігурації середовища виконання.

Typescript – мова програмування, що розширює можливості мови Javascript [32]. Перевагами Typescript над Javascript є:

- можливість явного визначення типів (статична типізація);
- підтримка використання повноцінних класів;
- можливість використання узагальненого програмування;
- підтримка підключення модулів.

Текст програми мовою програмування Typescript компілюється в текст мовою Javascript, що дозволяє використовувати дану мову усюди, де можна використовувати Javascript. В свою чергу, Typescript є повністю зворотно сумісним з Javascript.

Розробники мови вважають, що зазначені вище нововведення поліпшують читабельність тексту програми, а також пришвидшують процеси розробки та рефакторингу завдяки виявленню помилок, пов’язаних з динамічною типізацією, на момент компіляції тексту програми.

2.4. Аналіз фреймворків для створення інтерфейсу користувача

В наш час більшість розробників для побудови графічних інтерфейсів користувачів на базі web-технологій використовують спеціальні бібліотеки. Вони дозволяють створювати досить складні компоненти, а також перевикористовувати їх з мінімальними змінами. Також такі бібліотеки надають функціональні можливості для контролю та зміни станів як окремих компонентів, так і всієї системи.

Найпоширенішими на даний момент є такі фреймворки: Angular, React та Vue.js [33].

Розглянемо зазначені фреймворки, а також переваги та недоліки кожного з них.

Angular – фреймворк для розробклення web-додатків з використання шаблону проєктування Model-View-ViewModel [34]. Добре підходить для створення інтерактивних web-додатків.

Переваги:

- побудований з використанням мови програмування Typescript;
- наявність широко-функціональної CLI-утиліти для налаштування проєктів;
- докладна документація;
- одностороння прив'язка даних, що забезпечує виняткову поведінку додатка та зводить до мінімуму ризик можливих помилок;
- застосування шаблону Model-View-ViewModel, що дозволяє розробникам працювати окремо на видом та логікою компонентів;
- застосування механізму «Вживлення залежностей» для підтримки модульності коду, а також оптимізації роботи за рахунок одноразової ініціалізації окремих модулів;
- структура і архітектура, що вимагаються від розробників при використанні даного фреймворку, спеціально розробленні для побудови масштабованих систем;

Недоліки:

- велика кількість структурних компонентів (Injectables, Components, Pipes, Modules и т. д.);
- низька продуктивність в порівнянні з іншими двома фреймворками.

React – це бібліотека, що була розроблена для створення сучасних односторінкових додатків будь-якого масштабу та складності [35].

Переваги:

- простий дизайн фреймворку;
- використання JSX для шаблонування компонентів;
- детальна документація;
- висока продуктивність через аналіз змін до DOM завдяки симуляції Virtual DOM;
- одностороння прив'язка даних, що забезпечує виняткову поведінку додатка та зводить до мінімуму ризик можливих помилок;
- реалізація концепцій функціонального програмування, що надають можливості для створення простого в тестуванні та багаторазового перевикористання коду;
- можливість використання мови програмування Typescript.

Недоліки:

- неоднозначний, оскільки в деяких випадках надає розробнику декілька різних шляхів для реалізації певних речей без явної переваги одного з них;
- змішування шаблонів з логікою (JSX) може спантеличити деяких розробників при перших знайомствах з React;
- відходить від використання об’єктно-орієнтованого програмування на користь функціонального.

Vue.js – це Javascript-фреймворк для створення складних інтерфейсів користувача в рамках односторінкового web-додатку [36].

Переваги:

1. Посилений HTML. Це означає, що Vue.js має багато характеристик схожих з Angular, а це, завдяки використанню різних компонентів, допомагає оптимізації HTML-блоків.
2. Детальна документація.

3. Масштабування. Загалом Vue.js використовується для розробки шаблонів багаторазового використання різного розміру. Через це розробка шаблону великого розміру може бути виконана за той же час, що і більш простого.
4. Малий розмір. Vue.js важить близько 20 КБ, при цьому маючи досить гарні показники швидкості, а також надаючи гнучкість процесу розробки, що дозволяє досягти набагато кращої продуктивності в порівнянні з іншими фреймворками.

Недолік: Брак ресурсів. Vue.js займає досить невелику частку ринку в порівнянні з React або Angular, що означає, що обмін знаннями в цьому середовищі перебуває на початковій стадії.

Angular, React та Vue.js є досить багатофункціональними фреймворками. Всі три надають розробнику достатньо можливостей для створення як простих, так и складних інтерфейсів.

Для використання в рамках проєкту обрано бібліотеку React.

2.5. Висновки

Після детального аналізу можливих варіантів типу програмного забезпечення, що спирався на аналіз існуючих рішень, встановлено, що оптимальним для розробки та використання у процесі відлагодження серверного програмного забезпечення, побудованого на базі технології gRPC, є прикладне програмне забезпечення з графічним інтерфейсом користувача. Це пов’язано з тим, що графічний інтерфейс користувача надає більше можливостей для взаємодії з програмою. В свою чергу розповсюдження в якості прикладного програмного забезпечення звільнює від розробки додаткових механізмів комунікації, необхідних для надання доступу до gRPC-сервісів через браузер.

Проведено аналіз таких технологій розробки прикладного програмного забезпечення, як Windows Forms, Windows Presentation Foundation, Universal Windows Platform, Abstract Window Toolkit, JavaFX,

GTK, Qt, NW.js та Electron, за низкою критеріїв: крос-платформність; широкий набір як вбудованих, так і сторонніх компонентів; налаштування та адаптація зовнішнього вигляду компонентів; наявність окремої мови розмітки інтерфейсу користувача; велика спільнота розробників, що використовують дану технологію; можливість використання для створення продуктів з відкритим сирцевим кодом.

Через відповідність кожному з вищеперелічених критеріїв для побудови клієнта обрано технологію Electron. Використання даної технології передбачає написання сирцевого тексту програми мовою Javascript та використання мов HTML і CSS для стилізації графічного інтерфейсу.

Проте для уникнення певного роду помилок, пов’язаних з динамічною типізацією мови Javascript, було обрано мову Typescript, яка надає можливість використання статичної типізації та перевірки типів даних під час компіляції у текст мовою Javascript.

Обрана технологія надає можливість використання web-технологій для побудови графічних інтерфейсів, що надається технологією Electron, було проведено аналіз та порівняння найбільш популярних фреймворків Angular, React та Vue.js. Для використання в рамках проєкту обрано React через простий дизайн фреймворку, використання синтаксису JSX, реалізації концепцій функціонального програмування та використання віртуального DOM-дерева елементів для аналізу змін перед оновленням змісту сторінок.

3. РОЗРОБЛЕННЯ ПРОГРАМНИХ ЗАСОБІВ

3.1. Архітектура проєкту

При створенні прикладного програмного забезпечення з графічним інтерфейсом користувача за допомогою технології Electron розробники повинні підпорядковуватися певній концепції. За цією концепцією кожний додаток повинен складатися з двох окремих частим [37]:

- основного модуля;
- модуля графічного середовища.

Основний модуль відповідає за роботу з вікнами програми та обробку системних подій, що можуть статися в процесі роботи. Робота з вікнами програми передбачає створення вікон, що відображають певні HTML сторінки, при запуску програми та завершення програми, якщо всі вікна були зачинені. Слід зауважити, що при роботі у середовищах операційних систем сімейства Darwin програма повинна не завершуватися при зачинені всіх вікон, а переходити у дезактивований режим для більш швидкого повторного запуску, що буде являти собою лише створення нового вінка.

Модуль графічного середовища програми являє собою вікно спеціальної версії браузеру Chromium, що відображає певну HTML сторінку. Завдяки цьому графічний інтерфейс користувача будується за допомогою web-технологій. Використання спеціальної версії браузеру Chromium надає скриптам сторінки доступ до середовища Node.js, яке має інтерфейси для роботи з файловою системою, мережею і так далі. Додатково технологією Electron надаються інтерфейси для створення інших вікон, на кшталт діалогових та модальних, та інтерфейси для комунікації з основним модулем, якщо необхідно. При створенні вікна задається шлях до HTML-файлу, що буде відображатися.

3.2. Файлова структура проєкту

Коренева директорія проєкту має наступний вміст:

- директорія .vscode – містить налаштування для текстового редактора Visual Studio Code;
- директорія src – містить сирцевий текст програми;
- файли package.json та package-lock.json – файли, що містять опис проєкту та інформацію про автора, тексти команд для збірки та запуску програми, а також список бібліотек та інструментів, що необхідні для розробки, збірки та роботи проєкту;
- файли .eslint.js, .prettierrc.js, tsconfig.json, webpack.main.develop.js, webpack.main.production.js, webpack.renderer.develop.js та webpack.renderer.production.js – файли конфігурацій інструментів, що використовуються при розробленні та збірці проєкту; детально розглянуті в підрозділі 3.3.

Враховуючи архітектуру проєкту сирцевий текст програми в директорії «src» знаходиться в двох окремих піддиректоріях:

- main – містить сирцевий текст основного модуля програми;
- renderer – містить сирцевий текст модуля графічного середовища.

3.3. Налаштування збірки та середовища розробки програми

Програмне забезпечення, що розробляється, має два режими роботи:

- режим нормальної роботи: використовується кінцевим користувачем при роботі з програмою ;
- режим розробки: використовується в процесі розробки та тестування програмного забезпечення.

Для підготовки сирцевого тексту модулів програми до запуску в одному з цих режимів використовується інструмент webpack.

webpack – це пакувальник модулів, що використовується для збірки текстів програм, написаних мовою Javascript, та їх пакування разом із сторонніми бібліотеками, що використовуються. Надає розробникам широкі можливості для налаштування процесу пакування. Шляхом використання сторонніх розширень може бути налаштований для компіляції таких мов програмування, як Typescript, Coffescript та інших в тексти мовою програмування Javascript, а також для трансформації та пакування інших активів, що використовуються при розробленні web-додатків, таких як файлів гіпертекстової розмітки HTML, каскадних стилів CSS, файлів зі шрифтами в різних форматах, зображень і так далі. Також є можливість підключення плагінів для виконання додаткової роботи при пакуванні проєкту.

Конфігурації даного інструмента містяться в наступних файлах в кореневій директорії проєкту:

- webpack.main.production.js – конфігурація збірки сирцевого тексту основного модуля програми для роботи в нормальному режимі;
- webpack.main.develop.js – конфігурація збірки сирцевого тексту основного модуля програми в режимі розробки;
- webpack.renderer.production.js – конфігурація збірки сирцевого тексту модуля графічного середовища для роботи в нормальному режимі;
- webpack.renderer.develop.js – конфігурація запуску модуля графічного середовища в режимі розробки.

Оскільки обидва модуля програми написані мовою програмування Typescript, усі названі конфігурації мають певну спільну частину налаштувань, що відповідає за компіляцію тексту, написаного мовою програмування Typescript, в текст мовою Javascript. Для такої компіляції використовується розширення babel-loader, що являє собою прив’язку інструмента Babel до інструменту webpack.

Babel – це транслятор сирцевого тексту програм. Для компіляції Typescript у Javascript використовується разом з пакетом конфігурацій @babel/preset-typescript.

Слід зазначити, що даний інструменти не виконує перевірку типів, а лише транслює текст мови Typescript у текст мови Javascript. Для виконання перевірки типів використовується плагін «fork-ts-checker-webpack-plugin». Таким чином, при компіляції тексту програми два процеси: процес трансляції тексту та процес перевірки типів. У випадку, якщо будуть виявлені помилки у типізації, збірка завершиться помилкою.

Також до спільних налаштувань відноситься присвоєння значення змінної оточення «NODE_ENV», що визначає, в якому режимі запущена програма. Можливі значення даної змінної: «development» та «production», що позначають режими розробки та режим нормальної роботи відповідно.

Конфігурація збірки сирцевого тексту основного модуля програми для роботи в нормальному режимі компілює та пакує файли основного модуля програми. Результатом цього є файл «main.js», що створюється у директорії «dist» кореневої директорії проєкту. Процес збірки виконується за командою «build-main-prod». Запуск збірки відбувається за допомогою команди «start», що визначена у файлі package.json.

Конфігурація збірки сирцевого тексту основного модуля програми в режимі розробки повторює попередню конфігурацію. Єдиною різницею є значення змінної оточення «NODE_ENV». Запуск даної конфігурації виконується за допомогою команди «start-main-dev».

Конфігурація збірки сирцевого тексту модуля графічного середовища для роботи в нормальному режимі використовується для компіляції та пакування файлів, що відносяться до модуля графічного середовища. Дана конфігурація додає до налаштувань інструмента Babel декілька пакетів конфігурацій, а саме:

- @babel/preset-react – компіляція синтаксису JSX, що використовується при роботі з фреймворком React;

- @babel/preset-env – компіляція функціональних можливостей, які надаються в більш нових версіях стандарту ECMAScript, для роботи в середовищах, де такі нововведення ще не підтримуються.

Конфігурація задає налаштування для пакування файлів стилів SCSS за допомогою низки розширень. Так, за допомогою розширення sass-loader стилі в форматі SCSS спочатку компілюються в формат CSS, далі за допомогою розширення css-loader всі окремі файли стилів компонуються в один та за допомогою розширення style-loader вставляються безпосередньо в DOM-дерево елементів.

Для пакування файлів шрифтів використовується розширення file-loader.

Результатом збірки є файл «renderer.js» та директорія з шрифтами «fonts», що створюються у директорії «dist». Також, враховуючи той факт, що модуль графічного середовища повинен запускатися як web-додаток, в процесі збірки використовується плагін «html-webpack-plugin» для створення файлу «index.html». Призначення даного HTML-файлу – запуск скриптів файлу «renderer.js». Процес збірки виконується за командою «build-renderer-prod». Команда запуску відсутня, оскільки результати збірки використовуються основним моделем при створенні вікна програми.

Конфігурація збірки модуля графічного середовища для запуску в режимі розробки є розширенням аналогічної конфігурації для роботи в нормальному режимі. Дана конфігурація не створює жодних файлів. Слугує для запуску сервера розробки графічного середовища, котрий слідкує за змінами у файлах відповідного модуля і, якщо присутні зміни, пакує файли повторно . Доступ до актуальної збірки відбувається за протоколом HTTP.

В рамках зазначеного сервера розробки також налаштовано процеси гарячого перезавантаження програми. Це дозволяє переглядати та

тестувати зміни у налаштуванні зовнішнього вигляду та логіки компонентів без необхідності повного перезавантаження програмного забезпечення. Так, після повторного пакування файлів відбувається аналіз та доставка змін до середовища, де відображується графічний інтерфейс користувача.

Для запуску сервера розробки модуля графічного середовища використовується команда «start-renderer-dev».

В процесі розробки також використовуються такі інструменти, як ESLint та Prettier.

ESLint – це аналізатор тексту, написаного мовою програмування Javascript, на предмет підпорядкування певному набору правил та наявність синтаксичних, семантичних та певного набору логічних помилок. Є досить гнучким інструментом. Так, за допомогою вибору парсеру файлів @typescript-eslint та плагіну react-hooks реалізується підтримка інструментом мови програмування Typescript та фреймворку React з синтаксисом JSX. Налаштування даного інструменту

Prettier – інструмент для форматування сирцевого тексту. Підтримує велику кількість мов програмування, в тому числі і Typescript. Може бути інтегрований в інструмент ESLint шляхом використання плагіну prettier.

Для розробки використовується редактор Visual Studio Code. Є програмним забезпечення з відкритим сирцевим текстом, а отже розповсюджується безкоштовно. Надає широкі можливості для налаштування процесів, суміжних до розробки. Так, за допомогою розширення ESLint налаштовується інтеграція відповідного інструмента в процес написання сирцевого тексту програми. Така інтеграція надає безперервний аналіз тексту програми та виділення проблемних ділянок тексту, а також автоматичне виправлення помилок, якщо можливо, та, у разі інтегрування Prettier з ESLint, форматування сирцевого тексту при збереженні змін у певному файлі.

3.4. Принцип роботи основного модуля програми

Як було зазначено у підрозділі 3.1, основний модуль програми відповідає за роботу з вікнами програми та обробку системних подій, що можуть статися в процесі роботи.

Сирцевий текст основного модуля програми розміщений у файлі за шляхом «src/main/index.ts». Текст основного модуля програми складається з визначення та реалізації функції «createWindow» та встановлення обробників певних подій.

Функція createWindow відповідає за створення нового вікна програми. Для цього створюється об’єкт класу BrowserWindow. При створенні вікна у конструктор об’єкту класу передається параметри такі параметри, як розмір вікна за замовчуванням, граничні значення розмірів та інші. Важливим параметром є ввімкнення інтеграції з середовищем Node.js. Рекомендується не відображати вікно програми образу при створенні, а робити це в обробнику події «ready-to-show», що сигналізує про успішний процес ініціалізації вікна.

Важливим моментом є те, що в залежності від режиму запуску програми вікна відображають різний контент. Так, при роботі в нормальному режимі буде відображатися вміст файлу «index.html», що створюється в процесі пакування файлів графічного середовища. В свою чергу, при роботі в режимі розробки вікна будуть відображати контент за шляхом «http://localhost:8080», оскільки сервер розробки процесу графічного середовища працює саме на порті 8080. Це дозволяє користуватися перевагами гарячого перезавантаження програми.

Встановлюються обробники таких подій життєвого циклу програми:

- ready – подія, що сигналізує про завершення ініціалізації середовища Electron. Саме після цієї події дозволяється створювання вікон програми, що і відбувається викликом функції «createWindow». Також, в обробнику даної події при умові роботи в режимі розробки в браузер Chromium

встановлюється розширення «React Developer Tools». Дане розширення полегшує розробку додатка за допомогою фреймворка React, оскільки дозволяє переглядати стан окремих компонентів.

- window-all-closed – подія, що сигналізує про закриття усіх вікон програми. Обробник даної подіє ініціалізує процес завершення програми. Виключенням є ситуація, коли програма запущена в середовищі операційної системи сімейства Darwin. В такому випадку програма не завершується. Це пов’язано зі специфічним циклом роботи програм в названих операційних системах.
- activate – подія, що має місце лише при роботі в середовищі операційної системи сімейства Darwin. Сигналізує про повторний запуск програми. Обробник даної події у разі, якщо немає створеного вікна, ініціалізує нове.

3.5. Архітектура та файлова структура сирцевого тексту модуля графічного середовища

Фреймворк React надає різні підходи для організації архітектури програмного забезпечення, які, в свою чергу, залежать від обраної парадигми програмування. Так, існує варіант використання як об’єктно-орієнтованої парадигми програмування для побудови компонентів та реалізації логіки додатку, так і функціональної. При виборі між зазначеними парадигмами слід мати на увазі, що розробники даного фреймворку рекомендують використання концепції композиції [38], а не наслідування, для повторного використання логіки компонентів, а також той факт, що при реалізації функціональних елементів розробникам надаються більш розширені можливості для контролю стану та життєвого циклу компонентів, ніж при реалізації компонентів-класів.

Враховуючи схильність фреймворку до функціональної парадигми програмування, саме ця парадигма і буде використана при розробленні інтерфейсу користувача та логіки програмного забезпечення.

Реалізація компонентів, що розроблені в рамках проєкту знаходяться в директоріях «src/renderer/components» та «src/renderer/containers». В першій директорії знаходяться компоненти, що мають досить обмежену логіку та стан, або ж взагалі їх не мають. У другій – компоненти з великою кількістю логіки та досить широким станом. Такі компоненти зазвичай виступають в ролі контейнерів для інших компонентів. Також, слід зауважити, що при побудові компонентів графічного інтерфейсу частково використовувалися готові компоненти, що надаються бібліотекою blueprintjs. Дані компоненти не реалізовують жодної логіки, а використовуються лише для більш швидкого процесу налаштування зовнішнього вигляду інтерфейсу користувача.

Для менеджменту стану функціональних компонентів бібліотека React надає спеціальні методи, котрі називають хуками [39]. Попри наявність певної кількості таких хуків в бібліотеці, існує можливість створення власних, що дозволяє виносити певну частину стану компонента, а також логіку роботи з нею в окремі функції задля повторного використання в інших компонентах. Файли з реалізаціями таких функцій знаходяться в директорії «src/renderer/hooks».

При використанні бібліотеки React додаток набуває вигляду дерева компонентів. Так, кожен компонент має батьківський компонент, а також може мати будь-яку кількість дочірніх компонентів. Компонент може передавати в дочірні необхідні дані. Такі дані називають пропси. За допомогою механізму пропсів компонент може контролювати відображення дочірніх, а також надавати дочірнім компонентам можливість впливати на власний стан за допомогою передачі певних методів в пропсах.

Механізм пропсів не є оптимальним, якщо необхідно використовувати одні і ті ж самі дані в компонентах, що знаходяться далеко один від одного в дереві компонентів. Для вирішення таких ситуацій використовується механізм контекстів. Контексти дозволяють зробити певні дані доступними на будь-якому рівні дочірніх елементів компонента-постачальника контексту [40]. Контексти можуть використовуватися для реалізації певної логіки окремо від компонентів та надання інтерфейсів для роботи з даною логікою. Реалізації контекстів, що розроблено в рамках даного проєкту, знаходяться у відповідних піддиректоріях директорії «src/renderer/contexts».

3.6. Аналіз та використання proto-файлів

Офіційна документація технології gRPC визначає два можливих варіанти аналізу та використання структур даних, описаних в proto-файлах, в середовищі Node.js [41]:

- генерація статичного тексту мовою Javascript;
- динамічний аналіз типів даних.

Перший варіант являє собою використання інструменту з консольним інтерфейсом користувача protoc. Даний інструмент розроблений для аналізу вмісту proto-файлів та генерації класів, що реалізовують сериалізацію та десериалізацію визначених типів даних, а також класів для взаємодії з визначеними сервісами та абстрактних класів для їх реалізації для різних мов програмування, в тому числі і для Javascript. Даний варіант є оптимальним у разі, якщо заздалегідь відомо, які саме сервіси та типи даних визначаються в proto-файлах.

Даний варіант через низку причин не є бажаним до використання в рамках проєкту. Перш за все, це пов’язано з необхідністю використання стороннього програмного забезпечення, яке має або бути заздалегідь встановлено в середовищі операційної системи користувача, або пакуватися та розповсюджуватися разом з самою програмою, що

розробляється. Також, генерація тексту класів та їх динамічне імпортування під час роботи програми накладає додаткове навантаження на апаратне середовище виконання та може призвести до неочікуваних помилок. Ще однією причиною є те, що згенеровані класи сервісів не надають достатньо можливостей для контролю з’єднання з віддаленим сервісом, оскільки шлях до віддаленого сервісу можна задавати лише при ініціалізації класу, що згенерований для даного сервісу, і в подальшому неможливо змінити. Також немає можливості визначити стан з’єднання з віддаленим сервісом в конкретну точку в часі.

Динамічний аналіз типів даних досягається шляхом використання бібліотек @grpc/proto-loader та grpc. Перша бібліотека слугує для генерації певного опису вмісту proto-файлів, а друга – для безпосередньої взаємодії з віддаленим сервісом за допомогою результату роботи бібліотеки @grpc/proto-loader. Даний метод має низку переваг над описаним вище методом генерації статичного тексту. Так, названі бібліотеки дозволяють генерувати реалізацію класів для роботи з типами даних та сервісами безпосередньо при виконанні програми, що звільнює від додаткової роботи з файловою системою та динамічного імпортуванні сторонніх файлів з текстом програми. Проте згенерований бібліотекою @grpc/proto-loader опис вмісту proto-файлів не є достатнім для використання в рамках даного проєкту, оскільки містить в собі лише опис сервісів та методів цих сервісів та не надає опис типів даних, що використовуються. Також, даний метод не вирішує інших проблем, описаних вище в аналізі методу генерації статичного тексту програми. Через це даний метод також не є оптимальним для використання в рамках даного проєкту.

Після детального аналізу сирцевого тексту бібліотеки @grpc/proto-loader було встановлено, що в ній не реалізовується аналіз вмісту proto-файлів. Дана бібліотека використовує іншу, а саме

protobuf.js, для виконання цієї задачі. Сама бібліотека виконує роль рівня сумісності між protobuf.js та grpc.

Бібліотека protobuf.js використовується для аналізу типів даних та сервісів, що описані в proto-файлах. Надає дуже детальний опис присутніх даних, а також на основі цього опису може бути використана для серилізації та десерилізації інформації. Також гарантує підтримку «well-known» типів даних. Є оптимальним варіантом для використання в рамках даного проєкту.

Використання даної бібліотеки інкапсульоване в хук useProtoDefinition, який повертає об’єкт опису стану та методи для маніпулювання з цим об’єктом.

Об’єкт опису стану складається з таких полів:

- isLoading: boolean – показує, чи відбувається в даний момент часу аналіз proto-файлу;
- protoPath: null | string – шлях до обраного proto-файлу; за замовчуванням має значення null;
- includeDirs: null | Array<string> – список шляхів до директорій, які необхідно використовувати для пошуку зовнішніх імпортів; за замовчуванням має значення null;
- error: null | Error – об’єкт помилки, що сталася при аналізі proto-файлу; за замовчуванням має значення null;
- root: null | protobufjs.Root – об’єкт, що надає опис усіх типів даних та сервісів, що були визначені в обраному proto-файлі.

Методи , що реалізовані для маніпуляцій з об’єктом стану:

- setProtoPath: (protoPath: string): => void – використовується для встановлення шлях до proto-файлу;
- setIncludeDirs: (include: null| Array<string>) => void – використовується для встановлення списку шляхів до директорій, які необхідно використовувати для пошуку зовнішніх імпортів;

- load: () => void – використовується для початку процесу аналізу proto-файлу;
- clean: () => void – використовується для очищення усіх даних.

Слід зазначити, що бібліотека protobuf.js надає обмежені можливості для розв’язання шляху зовнішніх імпортів. Для реалізації більш гнучкого процесу імпортування додаткових типів даних перевизначається метод protobuf.Root.resolvePath та реалізується пошук необхідних даних відносно шляхів директорій, що зазначені у списку змінній стану includeDirs, якщо такі присутні. У разі, якщо необхідні дані не були знайдені, повертається результат виконання оригінальної реалізації даного методу.

Об’єкт класу protobuf.Root, що є значенням поля root в об’єкті стану, є об’єктом, що містить в собі повний опис сервісів і їх методів, а також типів даних, що використовуються в запитах.

3.7. Модуль контролю з’єднання з віддаленим сервером

Для підтримки з’єднання з віддаленим сервером використовується бібліотека grpc, а саме клас grpc.Client, що надається даною бібліотекою. Цей клас реалізує процеси створення та підтримки з’єднання з віддаленим сервером, надає інтерфейси для контролю його стану, певним чином сприяє реалізації процесів автоматичного поновлення з’єднання, а також надає допоміжні методи для проведення усіх типів запитів, визначених в технології gRPC. Так, для ініціювання нового з’єднання достатньо створити екземпляр цього класу. Конструктор класу приймає шлях до віддаленого серверу як параметр.

Екземпляр класу grpc.Client має певні методи для створення та налаштування запитів, а саме:

- makeUnaryRequest – для унарних запитів;
- makeClientStreamRequest – для запитів з потоками даних зі сторони клієнта;

- makeServerStreamRequest – для запитів з потоками даних зі сторони сервера;
- makeBidiStreamRequest – для запитів з двосторонніми потоками даних.

Дані методи мають спільну частину параметрів:

- метод для серилізації даних;
- метод для десерилізації даних;
- метадані запиту.

Слід зауважити, що при використанні бібліотеки @grpc/proto-loader виклики даних методів, а також створення методів для роботи з даними, інкапсульовані у певних обгортках над методами сервісів, що описані в proto-файлах.

Для отримання методів серилізації та десерилізації даних створюються обгортки над певними методами, що надаються разом з описом типів даних бібліотекою protobuf.js (див. підрозділ 3.7).

Логіка роботи зі з’єднанням з віддаленим сервером реалізована у контексті grpc. Даний контекст надає доступ до об’єкту стану та методу ініціювання змін цього об’єкту.

Об’єкт стану складається з наступних полів:

- client: null | grpc.Client – екземпляр класу grpc.Client, що слугує для роботи зі з’єднанням, або значення null, якщо немає поточного з’єднання;
- autoconnection: boolean – налаштування процесу автоматичного поновлення з’єднання при виникненні помилок;
- status: null | grpc.connectivityState – статус поточного з’єднання з віддаленим сервером або null.

Можливі значення статусу з’єднання:

- IDLE – невизначений стан; набуває чинності після стану TRANSIENT_FAILURE при умові вимкненого автоматичного поновлення з’єднання;

- CONNECTING – стан, що сигналізує про налаштовування каналу з’єднання;
- READY – стан, що сигналізує про вдале з’єднання з віддаленим сервером та можливість виконання запитів;
- TRANSIENT_FAILURE – виникнення певної помилки при налаштовуванні з’єднання;
- SHUTDOWN – стан, що показує про занадто велику кількість помилок при з’єднанні; сигналізує про те, що з’єднання з сервером більше не буде налаштовуватися.

Задля реалізації логіки роботи даного контексту додатково розроблено допоміжний хук useThunkReducer. На відміну від присутнього в бібліотеці React хука useReducer, даний хук дозволяє використання таких технік, як відкладання обчислень та вставка логіки [42].

В рамках модуля реалізовані публічні функції, результат виконання яких необхідно перенаправляти як параметр до методу ініціювання змін стану контексту задля маніпулювання зі з’єднанням до сервера. Ними є:

- connect: (url: string) => void – створення нового з’єднання;
- updateAutoconnection: (autoconnection: boolean) => void – зміна налаштувань автоматичного поновлення з’єднання;
- disconnect: () => void – переривання з’єднання.

Також реалізовано низку приватних функцій, подібних до попередніх. Вони слугують для інкапсуляції певних частин логіки та використовують публічними функціями для налаштування послідовності змін, необхідних задля досягання результату. До приватних функцій входять:

- setClient: (client: null | grpc.Client) => void – встановлення нового значення поля client в об’єкті стану контексту;
- setAutoconnection: (autoconnection: boolean) => void – встановлення нового значення поля autoconnection в об’єкті стану контексту;

- setStatus: (status: grpc.connectivityState) => void – встановлення нового значення поля status в об’єкті стану контексту;
- updateStatus: () => void – інкапсулює процеси зчитування поточного стану з’єднання з віддаленим сервером, ініціалізації зміни відповідного поля в об’єкті стану контексту за допомогою функції setClient, при необхідності запускає процес поновлення з’єднання та встановлює обробника події наступного зміну стану з’єднання, що повторно запускає описані процеси.

Реалізація даного модуля у вигляді контексту пов’язана з тим, що доступ та маніпулювання зі з’єднанням необхідно виконувати на різних рівнях дерева компонентів програми.

4. АНАЛІЗ РОЗРОБЛЕНИХ ПРОГРАМНИХ ЗАСОБІВ

4.1. Інтерфейс та логіка графічного середовища користувача

При запуску клієнта користувач потрапляє на сторінку вибору proto-файлу для роботи. Дане вікно складається з компонента вибору proto-файлу, списку компонентів вибору директорій та панелі кнопок керування.

Компонент вибору файлу має вигляд текстового поля з кнопкою «Browse» праворуч. При натисканні на компонент вибору файлу відкривається системне діалогове вікно для вибору основного proto-файлу. Дозволяється вибір лише одного файлу з розширенням «proto». У разі, якщо немає обраного proto-файлу, поле вибору файлу заповнене текстом «Choose proto file...», у протилежному випадку – абсолютним шляхом до обраного файлу.

Список компонентів вибору директорій використовується для встановлення базових директорій пошуку зовнішніх імпортів в основному proto-файлі. Має вигляд вертикального списку. Елементами даного списку є компоненти, візуально подібні до компонента вибору proto-файлу, з додатковою кнопкою, що має вигляд хрестика, праворуч. Додатково, останнім елементом списку є кнопка, що має вигляду знаку «+». При натисканні на один з компонентів списку відкривається системне діалогове вікно для вибору директорії. Дозволяється вибір декількох директорій. У випадку вибору декількох директорій для кожної з них буде створено окремий елемент списку. У випадку відкривання діалогового вікна шляхом вибору елементу, що відображує вже обрану директорію, при виборі іншої початкова директорія видаляється зі списку.

Панель кнопок керування складається з двох кнопок.

Перша кнопка є кнопкою очищення поточного стану вибору. Заповнена текстом червоного кольору «RESET». При натисканні очищує

заповнену інформацію про обраний proto-файл та обрані директорії пошуку зовнішніх імпортів.

Друга кнопка є змінною. Так, при виборі шляхів для аналізу дана кнопка має синій колір та заповнена текстом «LOAD». Слід зауважити, що кнопка є вимкненою, поки не вибрано шлях до основного proto-файлу. При натисканні на дану кнопку запускається процес аналізу proto-файлів. Під час цього кнопка стає вимкненою та має вигляд індикатору завантаження.

У разі виникнення помилки на етапі аналізу показується спливаюче повідомлення у верхній частині вікна з інформацією про помилку. Повідомлення має стандартний для бібліотеки blueprintjs вигляд. Індикатор завантаження набуває вигляду кнопки «LOAD».

У разі успішного кнопка «LOAD» замінюється на кнопку зеленого кольору з текстом «COMPLETE». При натисканні на цю кнопку відбувається перехід на сторінку з основним інтерфейсом програми.

Основний інтерфейс програми складається з таких компонентів: компонент вибору поточного методу сервісу ліворуч, компонент з інформацією про з’єднання зверху праворуч та компоненту проведення запиту праворуч.

Компонент вибору поточного методу сервісу має вигляд дерева списку елементів, кожний з яких відповідає конкретному сервісу, що описаний в proto-файлах. При натисканні на один з елементів він розкривається та показує список методів обраного сервісу. Слід зауважити, що в конкретний момент в часі розкритими може бути необмежена кількість елементів списку сервісів. При натисканні на елемент зі списку методів сервісу компонент для проведення запитів починає працювати з обраним методом.

Компонент з інформацією про з’єднання складається з таких елементів, як індикатор, поле вводу адреси, кнопки та перемикача процесу автоматичного поновлення з’єднання.

Поле вводу слугує для вводу шляху до віддаленого сервісу. При вводі символів постійно валідує введену строку. У разі виявлення невідповідності веденої строки форматам можливих адрес поле підсвічується червоним, а у разі відповідності – зеленим, сигналізуючи про валідність вводу. Поле не активне у разі, якщо в поточний момент часу вже існує з’єднання з сервером незалежно від його статусу.

Індикатор працює у двох режимах. Так, якщо з’єднання с сервером не встановлене (тобто поле вводу є активним), забарвлення індикатору повторює колір підсвітки поля вводу адреси. У разі існуючого з’єднання індикатор має забарвлення, що відповідає певному статусу з’єднання, а саме:

- сірий колір відповідає статусу IDLE;
- синій – CONNECTING;
- зелений – CONNECTED;
- помаранчевий – TRANSIENT_FAILURE;
- червоний – SHUTDOWN.

Опис статусів стану з’єднання наведено у підрозділі 3.7.

Кнопка в компоненті з інформацією про з’єднання може працювати у двох режимах. Так, у разі відсутності з’єднання з сервером ця кнопка має зелений колір та заповнена текстом «CONNECT». При натисканні ініціалізує процес створення з’єднання з сервером. Слід зауважити, що кнопка неактивна у разі виявлення помилок у введеній адресі. У разі присутності з’єднання з сервером дана кнопка має червоний колір з підписом «DISCONNECT», натискання на яку ініціалізує процес роз’єднання з сервером.

Перемикач в компоненті з інформацією про з’єднання слугує для налаштування процесу автоматичного поновлення з’єднання з віддаленим сервісом у разі помилок. Може знаходитись в двох станах: ввімкнений та вимкнений, що свідчить про поточне налаштування даного налаштування.

У вимкненому стані містить в собі текст «no authoconnection», а при ввімкненому – «authoconnection».

Компоненту проведення запиту складається з таких частин: списку поточних запитів та компоненту роботи з певним запитом, який в свою чергу складається з компоненту роботи з потоком запитів, компоненту роботи з потоком відповідей, компоненту роботи з метаданими клієнта, компоненту перегляду метаданих сервера та панелі управління запитом.

Список поточних запитів має вигляд вертикального набору вкладок. При натисканні на певну вкладку інформація про запит відображається у компоненті роботи з певним запитом. Кожна вкладка складається з назви запиту та кнопки зачинення вкладки. Слід зауважити, що при перемиканні між вкладками активні запити не перериваються, а оброблюються незалежно один від одного, що дозволяє проводити декілька запитів одночасно та взаємодіяти з кожним окремим. Також при зачиненні вкладки з активним запитом даний запит відміняється.

Компонент роботи з потоком запитів надає можливість перегляду опису типу даних, що використовується в обраному методі як запити та наповнювати їх даними для подальшого пакування та передачі на сервер.

Компонент роботи з потоком відповідей з сервера надає можливість перегляду опису типу даних, що використовується в обраному методі як відповіді та перегляду даних, отриманих від сервера.

Компонент роботи з метаданими клієнта використовується для налаштування початкових метаданих від клієнта. Має вигляд списку елементів, кожний з яких відповідає за редагування даних за певним ключем та може бути розгорнутий окремо за допомогою відповідної кнопки. Також кожний такий елемент має кнопку для його видалення. Для редагування даних за конкретним ключем необхідно розгорнути відповідний елемент. В розгорнутому стані кожний елемент має поле для редагування самого ключа, а також список полів для зміни даних. Кожне окреме значення може бути видалене за допомогою відповідної кнопки. Є

кнопка для створення нового значення. Також компонент має кнопки для створення нового ключа та очищення даних.

Компонент перегляду метаданих сервера та статусу виклику є надбудовою над попереднім компонентом. Складається з списку вкладок та зони перегляду обраних метаданих. Список вкладок має статичне наповнення, а саме вкладки «Initial», «Trailing» та «Status». Перші дві вкладки відповідають за відображення метаданих сервера. Це пов’язано з тим, що сервер посилає метадані двічі: перед відправленням першої відповіді та після відправлення усіх відповідей в складі об’єкту-статусу виконання методу. Третя вкладка відображає вміст об’єкту статусу.

Панель управління запитом має різний набір кнопок в залежності від типу методу та поточного стану запиту. Так, при роботі з унарним методом дана панель буде мати два можливих варіанти наповнення: або містити одну кнопку, натискання якої призведе до відправки запиту, або містити кнопку, що слугує для відміни поточного запиту.

При роботі з методами, що використовують потік даних зі сторони клієнту, панель може мати лише кнопку для початку роботи з методом, або ряд кнопок для контролю виконання поточного методу, а саме:

- кнопку для надсилання наступного запиту;
- кнопку для закінчення роботи з методом;
- кнопку для відміни роботи з методом.

При роботі з методами, що використовують потік даних зі сторони сервера панель управління має вигляд та функціонал, аналогічні до того, що використовуються при роботі з унарними методами, а при роботі з методами, що використовують потоки даних в обидві сторони – аналогічно до методів, що використовують потік даних зі сторони клієнту.

4.2. Особливості тестування клієнта

Особливість тестування даного клієнту полягає в тому, що є необхідність додатково розробляти програмне забезпечення серверного

типу, що побудоване на базі технології gRPC та тестувати його за допомогою одного з існуючих рішень.

Оскільки при розробленні даного клієнту потрібно було постійно тестувати нові функціональні можливості та виправлення різних помилок, робота над названим програмним забезпеченням не припинялась.

Остаточний вигляд програмного забезпечення для тестування клієнта, що реалізовується в рамках даного проєкту, має наступний вигляд.

Основним файлом є proto-файл test.proto. Даний файл імпортує інший, що називається test-message.proto. Це сприяє тестуванню механізму розв’язання зовнішніх імпортів, а також механізму пошуку файлів відносно списку базових директорій.

Файл test-message.proto містить в собі опис типу TestMessage, що складається з одного поля: google.protobuf.Struct struct = 1. Використання саме типу google.protobuf.Struct дозволяє значно прискорити процес підготовки до тестування, оскільки даний тип містить в собі майже всі ключові особливості технології, такі як списки, словники та взаємовиключні поля.

В основному файлі також визначено ще один тип даних TestMessageWrap з наступним набором полів:

- TestMessage message = 1;
- google.rpc.Status status = 2;
- uint32 n = 3.

Також у даному файлі визначено сервіс TestService з наступними методами:

1. rpc unaryMethod(TestMessage) returns (Message) – служить для тестування роботи з унарними методами. У відповідь повертає зміст поля message запиту, проте якщо визначено поле status, то завершує виклик на стороні сервера з тим статусом, що визначено у запиті.

2. rpc clientStreamMethod(stream TestMessage) returns (Message) – для тестування роботи з методами, що використовують потоки даних зі сторони клієнта. У відповідь через 10 запитів повертає зміст поля message останнього з них; працює з об’єктом статусу аналогічно до методу unaryMethod.
3. rpc serverStreamMethod(TestMessage) returns (stream Message) – для тестування роботи з методами, що використовують потоки даних зі сторони сервера. У відповідь повертає копії змісту поля message запиту у кількості, що дорівнює значенню поля n. Працює з об’єктом статусу аналогічно до методу unaryMethod.
4. rpc bidiStreamMethod(stream TestMessage) returns (stream Messag e) – для тестування роботи з методами, що використовують потоки даних з обох сторін. Працює аналогічно до методу serverStreamMethod. Проте на відміну від останнього приймає та обробляє потік запитів.

4.3. Рекомендації щодо подальшого вдосконалення

Функціональних можливостей, реалізованих в розробленому клієнті, достатньо для проведення процесу відлагодження серверного програмного забезпечення, що побудоване за допомогою технології gRPC. Проте дане програмне забезпечення можна розширити для підтримки роботи з більшим числом серверів.

Так, на даний момент клієнт не підтримує використання методів шифрування при з’єднанні з віддаленим сервером. Одним із можливих вдосконалень даної системи можна вважати реалізацію підтримки протоколу TLS для створення зашифрованого каналу передачі даних між клієнтом та віддаленим сервером.

Ще одним вдосконаленням можна вважати реалізацію історії запитів з можливістю її перегляду.

ВИСНОВКИ

Метою даного дипломного проєкту було розроблення клієнта для відлагодження серверного програмного забезпечення, що побудоване за допомогою технології gRPC.

Проведено докладний аналіз існуючих рішень, за допомогою чого обрано тип програмного забезпечення, а саме прикладне програмне забезпечення, та встановлено список обов’язкових для реалізації функціональних можливостей клієнту:

- аналіз вмісту proto-файлів;
- надання користувачеві детальної інформації про типи даних та сервіси, що визначені у proto-файлах;
- пакування даних у відповідні бінарні формати залежно від їх типу та визначень;
- надання користувачеві можливості інтерактивної роботи з усіма видами запитів, що існують в рамках технології gRPC, а саме унарними, односторонньо-стрімінгові як зі сторони клієнта, так і зі сторони серверу, а також двосторонньо-стрімінговимими;
- підтримка «well-known» типів даних;
- автоматичне поновлення з’єднання з віддаленим сервером.

Також проведено аналіз технологій розроблення прикладного програмного забезпечення. До використання обрано технологію Electron, а для побудови графічного середовища – фреймворк React.

При розробленні особлива увага приділялася підпорядкуванню архітектури системи концепціям, що використовуються з обраними технологіями.

Реалізовані усі висунуті вимоги. Тестування продукту виконано у відповідності до затвердженої програми та методики.

Використання створеного програмного забезпечення дозволяє розробниками та тестувальниками серверного програмного забезпечення,

що побудоване на базі технології gRPC, полегшити процеси аналізу коректності роботи сервісів та їх відлагодження.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Сергей Голубев. Разница между различными типами открытых лицензий [Електронний ресурс] / Сергей Голубев. – 2017. – Режим доступу: https://www.itweek.ru/foss/article/detail.php?ID=194342.
2. Лицензия для вашего open-source проекта / Хабр [Електронний ресурс] – 2014. – Режим доступу: https://habr.com/ru/post/243091/.
3. gRPC – A high-perfomance, open source universal RPC framework [Електронний ресурс] – Режим доступу: https://grpc.io/.
4. Johan Brandhorst. The gRPC Blog – The state of gRPC in the browser [Електронний ресурс] / Johan Brandhorst. – 2019. – Режим доступу: https://grpc.io/blog/state-of-grpc-web/.
5. Windows Forms – Вікіпедія [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Windows_Forms.
6. Windows Presentation Foundation (WPF) – WPF | Microsoft Docs [Електронний ресурс] – 2018. – Режим доступу: https://docs.microsoft.com/ru-ru/dotnet/framework/wpf/.
7. Введение в WPF | Microsoft Docs [Електронний ресурс] – 2016. – Режим доступу: https://docs.microsoft.com/ru-ru/dotnet/framework/wpf/introduction-to-wpf.
8. About Mono | Mono [Електронний ресурс] – Режим доступу: https://www.mono-project.com/docs/about-mono/.
9. WPF | Mono [Електронний ресурс] – Режим доступу: https://www.mono-project.com/docs/gui/wpf/.
10. What’s a Universal Windows Platform (UWP) app? – UWP applications | Microsoft Docs [Електронний ресурс] – 2018. – Режим доступу: https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide.
11. Abstract Window Toolkit (AWT) [Електронний ресурс] – Режим доступу: https://web.mit.edu/java_v1.5.0_22/distrib/share/docs/guide/awt/index.html.

12. Matthew Tyson. What is the JWM? Introducing the Java Virtual Machine | JavaWorld [Електронний ресурс] / Matthew Tyson. – 2020. – Режим доступу: https://www.javaworld.com/article/3272244/what-is-the-jvm-introducing-the-java-virtual-machine.html.
13. John Hunt. Key Java: Advanced Tips and Techniques [Text] / John Hunt, Alexander G. McManus. – Springer-Verlag London Limited, 2000. – c. 226.
14. Swing (Java) – Wikipedia [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Swing_(Java).
15. Библиотека Swing [Електронний ресурс] – Режим доступу: http://java-online.ru/libs-swing.xhtml.
16. Paul Larkin. JavaFX vs Java Swing: The Key Differences | Career Karma [Електронний ресурс] / Paul Larkin. – 2020. – Режим доступу: https://careerkarma.com/blog/javafx-vs-java-swing/.
17. JavaFX Overview (Release 8) [Електронний ресурс] – Режим доступу: https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm.
18. Getting Started with GTK – The GTK Project [Електронний ресурс] – Режим доступу: https://www.gtk.org/docs/getting-started/.
19. Language Bindings – The GTK Project [Електронний ресурс] – Режим доступу: https://www.gtk.org/docs/language-bindings/.
20. Glade – A User Interface Designer [Електронний ресурс] – Режим доступу: https://glade.gnome.org/index.html
21. Uning GTK+ on Windows: GTK+ 3 Reference Manual [Електронний ресурс] – Режим доступу: https://developer.gnome.org/gtk3/stable/gtk-windows.html.
22. Changing GTK themes in OSx [Електронний ресурс] – Режим доступу: http://blog.e-shell.org/60.
23. Qt – Вікіпедія [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Qt.

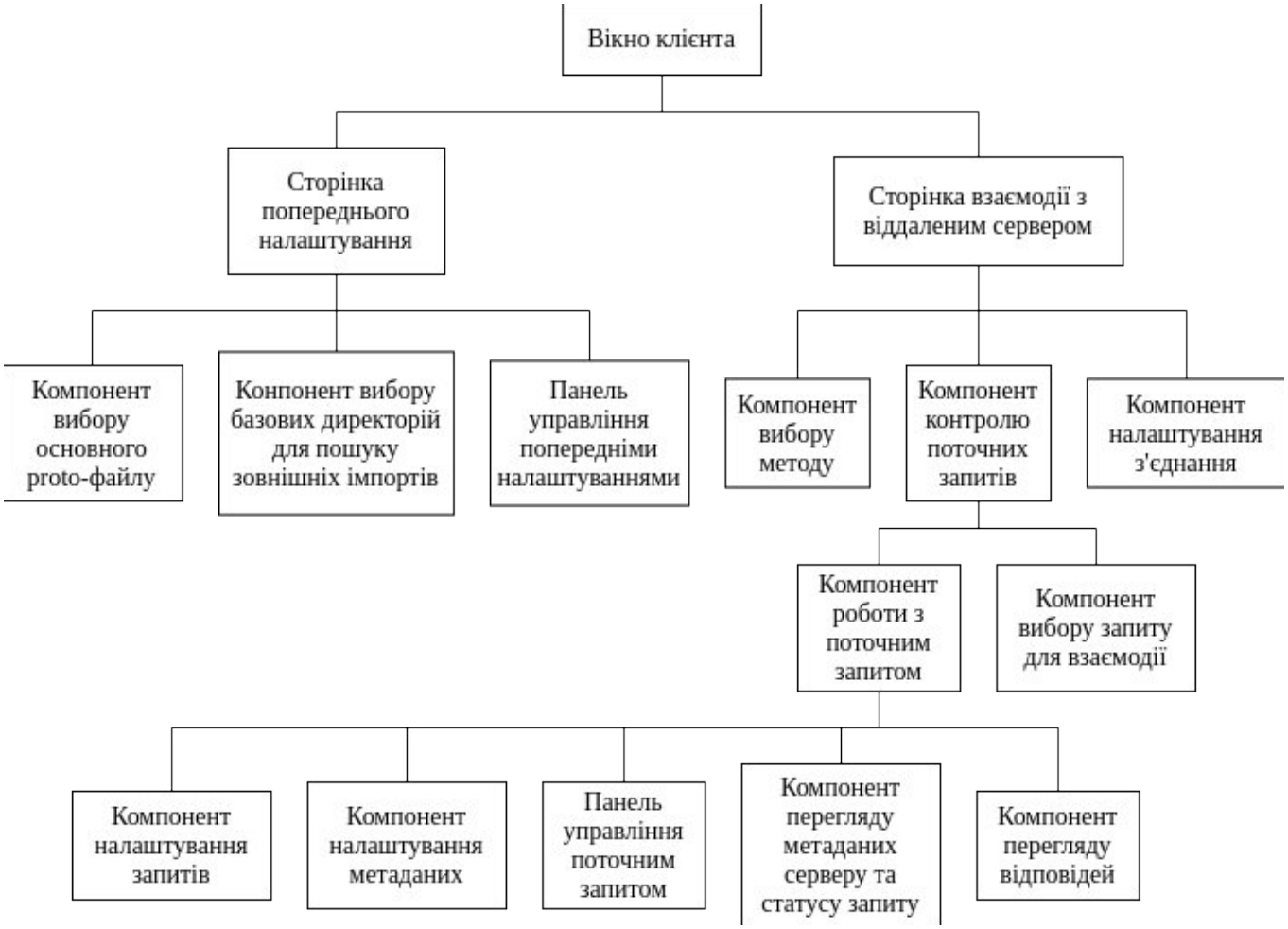
24. Qt Creator – A Cross-platform IDE for Application Development [Електронний ресурс] – Режим доступу: https://www.qt.io/product/development-tools.
25. Open Source Licensing of Qt | Qt 5.15 [Електронний ресурс] – Режим доступу: https://doc.qt.io/qt-5/opensourcelicense.html.
26. Index | Node.js v12.18.0 Documentation [Електронний ресурс] – Режим доступу: https://nodejs.org/dist/latest-v12.x/docs/api/.
27. Modulecounts [Електронний ресурс] – Дата перегляду: 16.04.2020 – Режим доступу: http://www.modulecounts.com/.
28. Technical Differences Between Electron and NM.js | Electron [Електронний ресурс] – Режим доступу: https://www.electronjs.org/docs/development/electron-vs-nwjs.
29. ECMAScript – Вікіпедія [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/ECMAScript.
30. JavaScript | MDN [Електронний ресурс] – Режим доступу: https://developer.mozilla.org/uk/docs/Web/JavaScript.
31. Типи та структури даних у JavaScript – JavaScript | MDN [Електронний ресурс] – Режим доступу: https://developer.mozilla.org/uk/docs/Web/JavaScript/Data_structures.
32. TypeScript – JavaScript that scales. [Електронний ресурс] – Режим доступу: https://www.typescriptlang.org/index.html.
33. Top Front-End Frameworks in 2020 | Existek Blog [Електронний ресурс] – 2020. – Режим доступу: https://existek.com/blog/top-front-end-frameworks-2020/.
34. Angular – Introduction to the Angular Docs [Електронний ресурс] – Режим доступу: https://angular.io/docs.
35. Початок роботи – React [Електронний ресурс] – Режим доступу: https://uk.reactjs.org/docs/getting-started.html.
36. Introduction – Vue.js [Електронний ресурс] – Режим доступу: https://vuejs.org/v2/guide/.

37. Архитектура приложения Electron | Electron [Електронний ресурс] – Режим доступу: https://www.electronjs.org/docs/tutorial/application-architecture.
38. Композиція проти наслідування – React [Електронний ресурс] – Режим доступу: https://uk.reactjs.org/docs/composition-vs-inheritance.html.
39. Ознайомлення з Хуками – React [Електронний ресурс] – Режим доступу: https://uk.reactjs.org/docs/hooks-intro.html.
40. Context – React [Електронний ресурс] – Режим доступу: https://uk.reactjs.org/docs/context.html.
41. gRPC – Quick Start [Електронний ресурс] – Режим доступу: https://www.grpc.io/docs/languages/node/quickstart/.
42. Thunk – Wikipedia [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Thunk.

ДОДАТКИ

Додаток 1

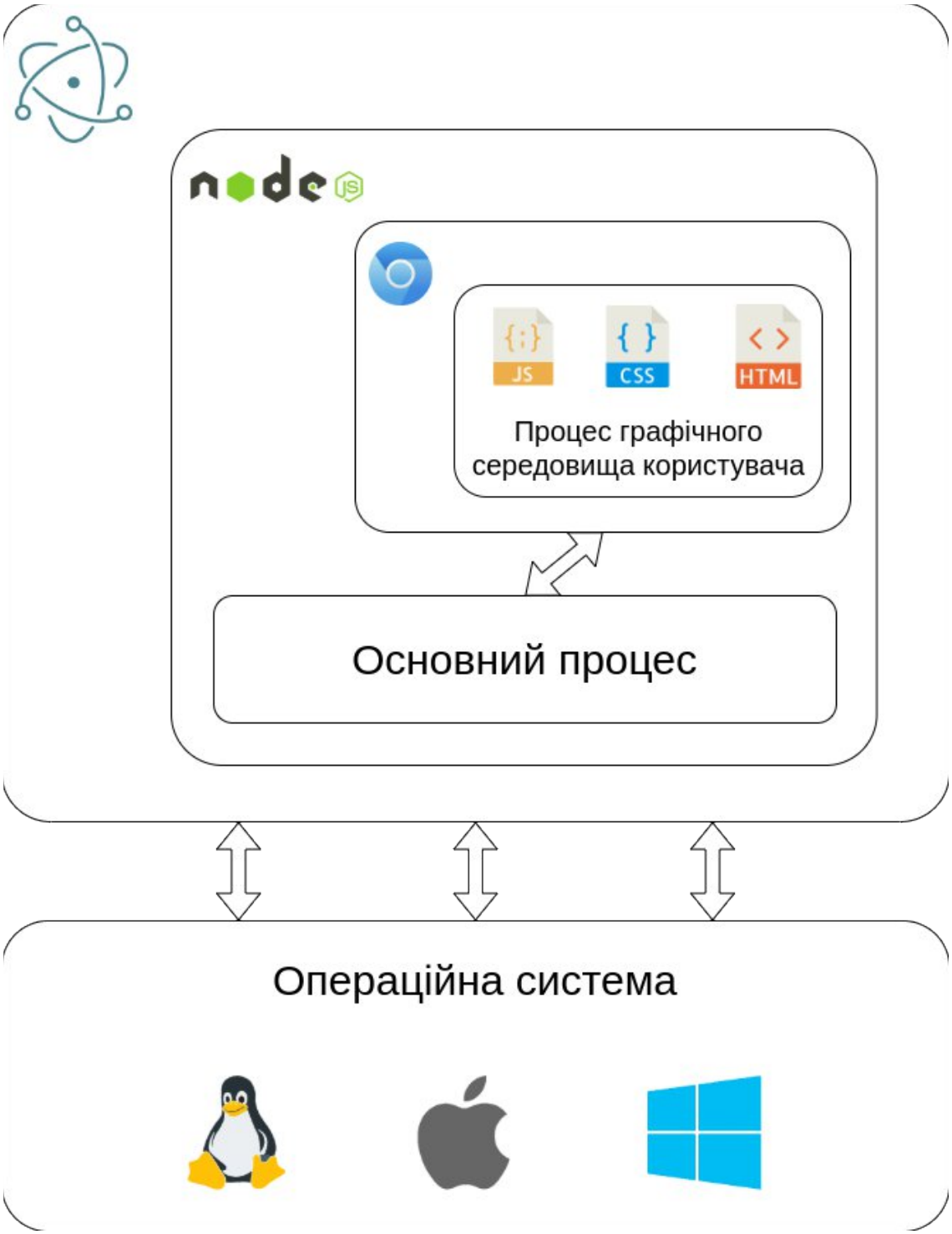
Копії графічних матеріалів



ДП.045440-06-99. gRPC-клієнт для відлагодження серверного програмного забезпечення. Інтерфейс користувача. Схема композиції

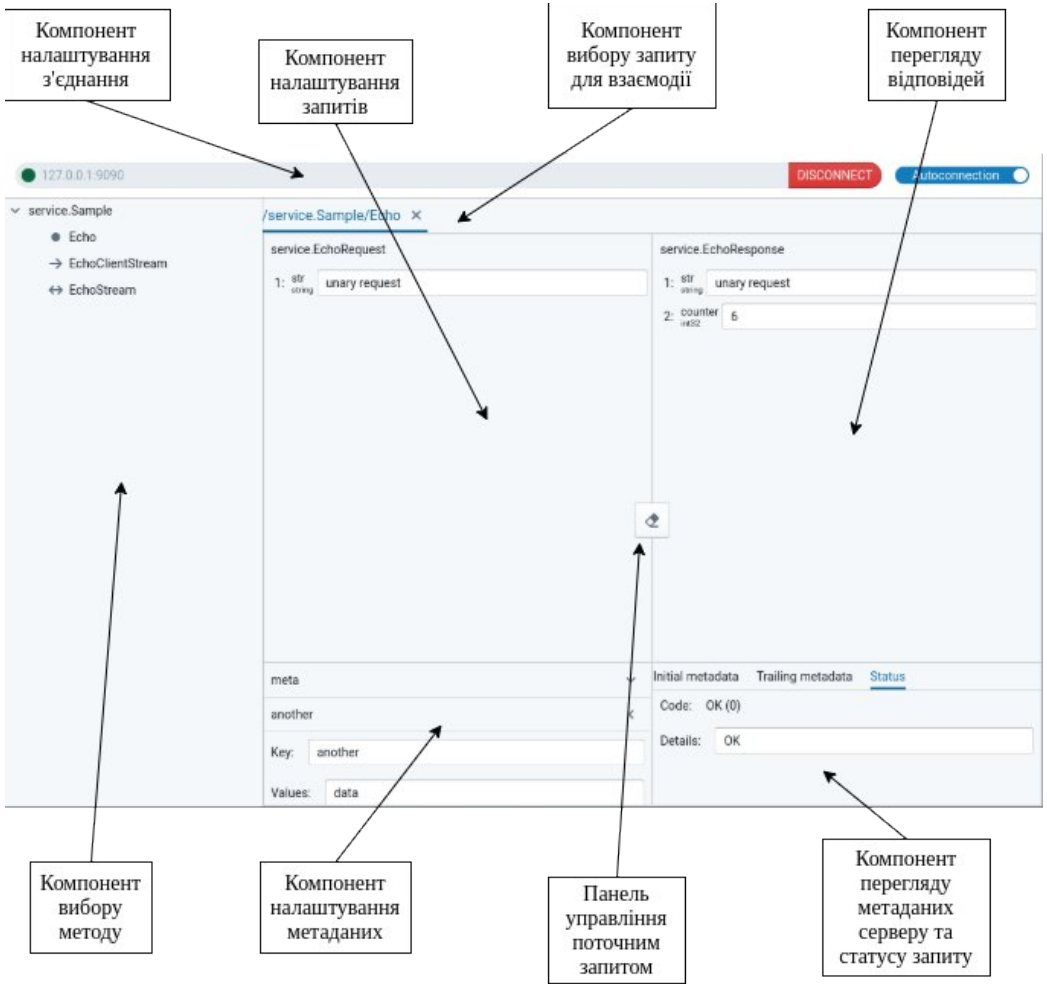
ДП.045440-07-99. gRPC-клієнт для відлагодження серверного програмного забезпечення. Взаємодія користувача з програмою. Діаграма прецедентів

Архітектура прикладного програмного забезпечення на базі платформи Electron



Рухайло П.О., КП-61

Структура вікна проведення запитів



Рухайло П.О., КП-61

Додаток 2

Лістинг програми

```
import { app, BrowserWindow } from 'electron';

function createWindow(): void {
  const window = new BrowserWindow({
    width: 1024,
    height: 768,
    webPreferences: { nodeIntegration: true },
    show: false
  });

  window.on('ready-to-show', () => window.show());

  if (process.env.NODE_ENV === 'development') {
    window.loadURL('http://localhost:8080');

    window.webContents.once('did-frame-finish-load', () =>
window.webContents.openDevTools());
  } else window.loadFile(`index.html`);
}

app.whenReady().then(async () => {
  if (process.env.NODE_ENV === 'development')
    await import('electron-devtools-installer')
      .then(({ default: installExtension, REACT_DEVELOPER_TOOLS }) =>
installExtension(REACT_DEVELOPER_TOOLS))
      .then(name => console.log(`Added React developer tools:  ${name}`))
      .catch(error => console.log('An error occurred while adding React
developer tools: ', error));

  createWindow();
});

app.on('activate', () => {
  if (BrowserWindow.getAllWindows().length === 0) {
    createWindow();
  }
});

app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit();
  }
});
```

```
import { Reducer, useReducer, useCallback, useRef } from 'react';
import { v4 as uuid } from 'uuid';
import isKeyNameValid from '../utils/isKeyNameValid';

export type MetadataDraftValue = {
  id: string;
  value: string;
  valueError: null | string;
};

export type MetadataDraftKey = {
  id: string;
  keyName: string;
  keyNameError: null | string;
  valuesError: null | string;
  values: Array<MetadataDraftValue>;
};

export type MetadataDraft = Array<MetadataDraftKey>;

export type UseMetadataDraftState = {
  errorNumber: number;
  metadataDraft: MetadataDraft;
};

const defaultState: UseMetadataDraftState = { errorNumber: 0,
metadataDraft: [] };

enum ActionTypes {
  ADD_KEY = 'ADD_KEY',
  UPDATE_KEY = 'UPDATE_KEY',
  REMOVE_KEY = 'REMOVE_KEY',
  ADD_VALUE = 'ADD_VALUE',
  UPDATE_VALUE = 'UPDATE_VALUE',
  REMOVE_VALUE = 'REMOVE_VALUE',
  CLEAN = 'CLEAN'
}

type AddKeyAction = {
  type: ActionTypes.ADD_KEY;
};
```

```
type UpdateKeyAction = {
  type: ActionTypes.UPDATE_KEY;
  payload: { id: string; keyName: string };
};

type RemoveKeyAction = {
  type: ActionTypes.REMOVE_KEY;
  payload: { id: string };
};

type AddValueAction = {
  type: ActionTypes.ADD_VALUE;
  payload: { id: string };
};

type UpdateValueAction = {
  type: ActionTypes.UPDATE_VALUE;
  payload: { id: string; valueId: string; value: string };
};

type RemoveValueAction = {
  type: ActionTypes.REMOVE_VALUE;
  payload: { id: string; valueId: string };
};

type CleanAction = { type: ActionTypes.CLEAN };

type UseMetadataDraftAction =
  | AddKeyAction
  | UpdateKeyAction
  | RemoveKeyAction
  | AddValueAction
  | UpdateValueAction
  | RemoveValueAction
  | CleanAction;

const reducer: Reducer<UseMetadataDraftState, UseMetadataDraftAction> =
(prevState, action) => {
  switch (action.type) {
    case ActionTypes.ADD_KEY:
      return {
        errorNumber: prevState.errorNumber + 2,
        metadataDraft: [
```

```
          ...prevState.metadataDraft,
          {
            id: uuid(),
            keyName: '',
            keyNameError: 'Key name in invalid',
            valuesError: 'Need to specify at least one value',
            values: []
          }
        ]
      };
    case ActionTypes.UPDATE_KEY: {
      const i = prevState.metadataDraft.findIndex(({ id }) => id ===
action.payload.id);

      if (i === -1) throw new Error(`Key with id ${action.payload.id} not
found`);

      const isValid = isKeyNameValid(action.payload.keyName);

      return {
        errorNumber: isValid
          ? prevState.metadataDraft[i].keyNameError === null
            ? prevState.errorNumber
            : prevState.errorNumber - 1
          : prevState.metadataDraft[i].keyNameError === null
          ? prevState.errorNumber + 1
          : prevState.errorNumber,
        metadataDraft: [
          ...prevState.metadataDraft.slice(0, i),
          {
            ...prevState.metadataDraft[i],
            keyName: action.payload.keyName,
            keyNameError: isValid ? null : 'Key name in invalid'
          },
          ...prevState.metadataDraft.slice(i + 1)
        ]
      };
    }
    case ActionTypes.REMOVE_KEY: {
      const i = prevState.metadataDraft.findIndex(({ id }) => id ===
action.payload.id);
```

```
      if (i === -1) throw new Error(`Key with id ${action.payload.id} not
found`);

      const errorsCount =
        (prevState.metadataDraft[i].keyNameError === null ? 0 : 1) +
        (prevState.metadataDraft[i].valuesError === null ? 0 : 1) +
        prevState.metadataDraft[i].values.reduce(
          (count, { valueError }) => (valueError === null ? count : count +
1),
          0
        );

      return {
        errorNumber: prevState.errorNumber - errorsCount,
        metadataDraft: [...prevState.metadataDraft.slice(0,
i), ...prevState.metadataDraft.slice(i + 1)]
      };
    }
    case ActionTypes.ADD_VALUE: {
      const i = prevState.metadataDraft.findIndex(({ id }) => id ===
action.payload.id);

      if (i === -1) throw new Error(`Key with id ${action.payload.id} not
found`);

      return {
        errorNumber: prevState.metadataDraft[i].values.length === 0 ?
prevState.errorNumber : prevState.errorNumber + 1,
        metadataDraft: [
          ...prevState.metadataDraft.slice(0, i),
          {
            ...prevState.metadataDraft[i],
            valuesError: null,
            values: [...prevState.metadataDraft[i].values, { id: uuid(),
value: '', valueError: 'Value in invalid' }]
          },
          ...prevState.metadataDraft.slice(i + 1)
        ]
      };
    }
    case ActionTypes.UPDATE_VALUE: {
      const i = prevState.metadataDraft.findIndex(({ id }) => id ===
action.payload.id);
```

```
      if (i === -1) throw new Error(`Key with id ${action.payload.id} not
found`);

      const j = prevState.metadataDraft[i].values.findIndex(({ id }) => id
=== action.payload.valueId);

      if (j === -1) throw new Error(`Value with id
${action.payload.valueId} not found`);

      const isValid = action.payload.value.length > 0;

      return {
        errorNumber: isValid
          ? prevState.metadataDraft[i].values[j].valueError === null
            ? prevState.errorNumber
            : prevState.errorNumber - 1
          : prevState.metadataDraft[i].values[j].valueError === null
          ? prevState.errorNumber + 1
          : prevState.errorNumber,
        metadataDraft: [
          ...prevState.metadataDraft.slice(0, i),
          {
            ...prevState.metadataDraft[i],
            values: [
              ...prevState.metadataDraft[i].values.slice(0, j),
              {
                ...prevState.metadataDraft[i].values[j],
                value: action.payload.value,
                valueError: isValid ? null : 'Value in invalid'
              },
              ...prevState.metadataDraft[i].values.slice(j + 1)
            ]
          },
          ...prevState.metadataDraft.slice(i + 1)
        ]
      };
    }
    case ActionTypes.REMOVE_VALUE: {
      const i = prevState.metadataDraft.findIndex(({ id }) => id ===
action.payload.id);
```

```
      if (i === -1) throw new Error(`Key with id ${action.payload.id} not
found`);

      const j = prevState.metadataDraft[i].values.findIndex(({ id }) => id
=== action.payload.valueId);

      if (j === -1) throw new Error(`Value with id
${action.payload.valueId} not found`);

      const willBeEmptyValues = prevState.metadataDraft[i].values.length
=== 1;

      return {
        errorNumber:
          (prevState.metadataDraft[i].values[j].valueError === null
            ? prevState.errorNumber
            : prevState.errorNumber - 1) + (willBeEmptyValues ? 1 : 0),
        metadataDraft: [
          ...prevState.metadataDraft.slice(0, i),
          {
            ...prevState.metadataDraft[i],
            valuesError: willBeEmptyValues ? 'Need to specify at least one
value' : null,
            values: [
              ...prevState.metadataDraft[i].values.slice(0, j),
              ...prevState.metadataDraft[i].values.slice(j + 1)
            ]
          },
          ...prevState.metadataDraft.slice(i + 1)
        ]
      };
    }
    case ActionTypes.CLEAN:
      return defaultState;
  }
};

export default function useMetadataDraft(): [
  UseMetadataDraftState,
  {
    addKey: () => void;
    updateKey: (id: string, keyName: string) => void;
    removeKey: (id: string) => void;
```

```
    addValue: (id: string) => void;
    updateValue: (id: string, valueId: string, value: string) => void;
    removeValue: (id: string, valueId: string) => void;
    clean: () => void;
  }
] {
  const [state, dispatch] = useReducer(reducer, defaultState);

  const addKey = useCallback<ReturnType<typeof
useMetadataDraft>[1]['addKey']>(
    () => dispatch({ type: ActionTypes.ADD_KEY }),
    []
  );

  const updateKey = useCallback<ReturnType<typeof
useMetadataDraft>[1]['updateKey']>(
    (id, keyName) => dispatch({ type: ActionTypes.UPDATE_KEY, payload:
{ id, keyName } }),
    []
  );

  const removeKey = useCallback<ReturnType<typeof
useMetadataDraft>[1]['removeKey']>(
    id => dispatch({ type: ActionTypes.REMOVE_KEY, payload: { id } }),
    []
  );

  const addValue = useCallback<ReturnType<typeof
useMetadataDraft>[1]['addValue']>(
    id => dispatch({ type: ActionTypes.ADD_VALUE, payload: { id } }),
    []
  );

  const updateValue = useCallback<ReturnType<typeof
useMetadataDraft>[1]['updateValue']>(
    (id, valueId, value) => dispatch({ type: ActionTypes.UPDATE_VALUE,
payload: { id, valueId, value } }),
    []
  );

  const removeValue = useCallback<ReturnType<typeof
useMetadataDraft>[1]['removeValue']>(
```

```
    (id, valueId) => dispatch({ type: ActionTypes.REMOVE_VALUE, payload:
{ id, valueId } }),
    []
  );

  const clean = useCallback<ReturnType<typeof
useMetadataDraft>[1]['clean']>(
    () => dispatch({ type: ActionTypes.CLEAN }),
    []
  );

  const actionsRef = useRef({ addKey, updateKey, removeKey, addValue,
updateValue, removeValue, clean });

  return [state, actionsRef.current];
}

import fs from 'fs';
import path from 'path';
import * as protobuf from 'protobufjs';
import { Reducer, useCallback, useEffect, useReducer, useRef } from
'react';

export type UseProtoDescriptionHookState =
  | ({ isLoading: false } & (
      | ((
          | {
              error: Error;
              root: null;
            }
          | {
              error: null;
              root: protobuf.Root;
            }
        ) & {
          protoPath: string;
        })
      | { error: null; root: null; protoPath: null | string }
    ) & { includeDirs: null | Array<string> })
  | { isLoading: true; error: null; root: null; protoPath: string;
includeDirs: null | Array<string> };

const defaultState: UseProtoDescriptionHookState = {
```

```
  isLoading: false,
  error: null,
  root: null,
  protoPath: null,
  includeDirs: null
};

enum ActionTypes {
  SET_PROTO_PATH = 'SET_PROTO_PATH',
  SET_INCLUDE_DIRS = 'SET_INCLUDE_DIRS',
  START_LOADING = 'START_LOADING',
  LOADED = 'LOADED',
  CLEAN = 'CLEAN'
}

type SetProtoPathAction = {
  type: ActionTypes.SET_PROTO_PATH;
  payload: { protoPath: null | string };
};

type SetIncludeDirsAction = {
  type: ActionTypes.SET_INCLUDE_DIRS;
  payload: { includeDirs: null | Array<string> };
};

type StartLoadingAction = {
  type: ActionTypes.START_LOADING;
};

type LoadedAction = {
  type: ActionTypes.LOADED;
  payload: { error: Error } | { root: protobuf.Root };
};

type CleanAction = {
  type: ActionTypes.CLEAN;
};

type UseProtoDescriptionHookAction =
  | SetProtoPathAction
  | SetIncludeDirsAction
  | StartLoadingAction
  | LoadedAction
```

```
  | CleanAction;

const reducer: Reducer<UseProtoDescriptionHookState,
UseProtoDescriptionHookAction> = (prevState, action) => {
  switch (action.type) {
    case ActionTypes.SET_PROTO_PATH:
    case ActionTypes.SET_INCLUDE_DIRS: {
      if (prevState.isLoading || prevState.root !== null ||
prevState.error !== null)
        throw new Error('Proto definition already loaded');

      return { ...prevState, ...action.payload };
    }
    case ActionTypes.START_LOADING: {
      if (prevState.protoPath === null) throw new Error('Proto path is not
set');

      const { protoPath, includeDirs } = prevState;

      return { ...defaultState, isLoading: true, protoPath, includeDirs };
    }
    case ActionTypes.LOADED: {
      if (!prevState.isLoading) throw new Error('Not in loading state');
      return { ...prevState, ...action.payload, isLoading: false };
    }
    case ActionTypes.CLEAN:
      return { ...defaultState };
  }
};

export default function useProtoDefinition(): [
  UseProtoDescriptionHookState,
  {
    setProtoPath: (protoPath: null | string) => void;
    setIncludeDirs: (includeDirs: null | Array<string>) => void;
    load: () => void;
    clean: () => void;
  }
] {
  const [state, dispatch] = useReducer(reducer, defaultState);

  useEffect(() => {
    if (!state.isLoading) return;
```

```
const root = new protobuf.Root();

const originalResolvePath = root.resolvePath;
root.resolvePath = (origin, target) => {
  if (path.isAbsolute(target)) {
    return target;
  }

  if (state.includeDirs !== null && state.includeDirs.length > 0)
    for (const directory of state.includeDirs) {
      const fullPath: string = path.join(directory, target);

      try {
        fs.accessSync(fullPath, fs.constants.R_OK);
        return fullPath;
      } catch (err) {
        continue;
      }
    }

  return originalResolvePath(origin, target);
};

let ignore = false;

protobuf
  .load(state.protoPath, root)
  .then(root => {
    if (!ignore) {
      root.resolveAll();
      dispatch({
        type: ActionTypes.LOADED,
        payload: { root }
      });
    }
  })
  .catch(error => {
    if (!ignore)
      dispatch({
        type: ActionTypes.LOADED,
        payload: { error }
      });
```

```
      });

    return () => {
      ignore = true;
    };
  }, [state.includeDirs, state.isLoading, state.protoPath]);

  const setProtoPath = useCallback(
    (protoPath: null | string) =>
      dispatch({
        type: ActionTypes.SET_PROTO_PATH,
        payload: { protoPath }
      }),
    []
  );

  const setIncludeDirs = useCallback(
    (includeDirs: null | Array<string>) =>
      dispatch({
        type: ActionTypes.SET_INCLUDE_DIRS,
        payload: { includeDirs }
      }),
    []
  );

  const load = useCallback(
    () =>
      dispatch({
        type: ActionTypes.START_LOADING
      }),
    []
  );

  const clean = useCallback(() => dispatch({ type: ActionTypes.CLEAN }),
[]);

  const actionsRef = useRef({ setProtoPath, setIncludeDirs, load, clean });

  return [state, actionsRef.current];
}

import { Dispatch, Reducer, useState, useRef, useCallback } from 'react';
```

```
export default function useThunkReducer<State, Action extends { type:
string; payload: unknown }>(
  reducer: Reducer<State, Action>,
  initialState: State
): [State, Dispatch<Action | Thunk<State, Action>>] {
  const [state, innerSetState] = useState(initialState);

  const stateRef = useRef(state);

  const getState = useCallback(() => stateRef.current, []),
    setState = useCallback((newState: State) => {
      stateRef.current = newState;
      innerSetState(newState);
    }, []);

  const reduce = useCallback((action: Action) => reducer(getState(),
action), [reducer, getState]);

  const dispatch: Dispatch<Action | Thunk<State, Action>> = useCallback(
    (action: Action | Thunk<State, Action>) =>
      typeof action === 'function' ? action(dispatch, getState) :
setState(reduce(action)),
    [getState, setState, reduce]
  );

  return [state, dispatch];
}

export type Thunk<State, Action extends { type: string; payload?:
unknown }> = (
  dispatch: Dispatch<Action | Thunk<State, Action>>,
  getState: () => State
) => void;
```

Додаток 3

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

GRPC-КЛІЄНТ ДЛЯ ВІДЛАГОДЖЕННЯ СЕРВЕРНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Виконав: студент IV курсу, групи КП-61 Рухайло Павло Олегович

Науковий керівник: доцент кафедри ПЗКС, к.т.н., доцент Онай Микола Володимирович

Київ – 2020

Особливості технології

gRPC – технологія виклику віддалених процедур.

Переваги:

1. HTTP/2
2. Потокові запити
3. Protocol Buffers

```
message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phones = 4;

  google.protobuf.Timestamp last_updated = 5;
}

message AddressBook {
  repeated Person people = 1;
}
```

Актуальність

1. Зростання попиту на технологію.
2. Обмеженість у часі при реалізації функціональних можливостей програмного забезпечення.
3. Необхідність наявності зручного інструменту для тестування та відлагодження програмного забезпечення.

Існуючі рішення

CLI-орієнтовані

- gcall
- grpcurl
- polyglot
- grpcc
- evans

GUI-орієнтовані

- Yodelay
- grpcox
- gRPC UI
- Delivery for gRPC
- BloomRPC

Мета розроблення

Метою розроблення є надання розробникам та тестувальникам серверного програмного забезпечення, що побудоване на базі технології gRPC, зручного інструменту для використання у процесах аналізу коректності роботи сервісів та їх відлагодження.

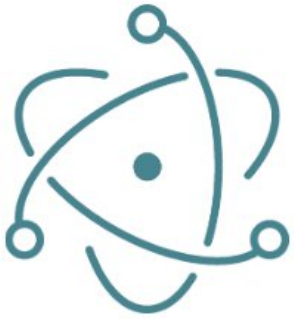
Вимоги

Програмний продукт повинен забезпечувати такі основні функціональні можливості:

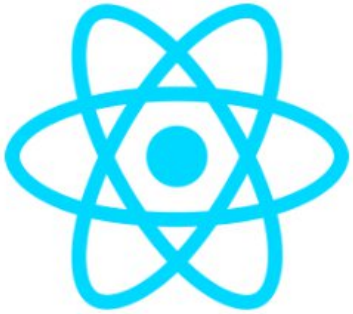
- аналіз вмісту proto-файлів;
- надання користувачеві детальної інформації про типи даних та сервіси, що визначені у proto-файлах;
- пакування даних у відповідні бінарні формати залежно від їх типу та визначень;
- надання користувачеві можливості інтерактивної роботи з усіма видами запитів, що існують в рамках технології gRPC, а саме унарними, односторонньо–стрімінгові як зі сторони клієнта, так і зі сторони серверу, а також двосторонньо–стрімінговимими;
- паралельне виконання запитів;
- підтримка «well-known» типів даних;
- автоматичне поновлення з’єднання з віддаленим сервером.

Засоби реалізації

Тип програмного забезпечення: прикладне програмне забезпечення з графічним інтерфейсом користувача.



Платформа Electron



Клієнтська бібліотека React

Мова програмування Typescript

Мова стилізації Sass

Архітектура програмного забезпечення

Два модуля:

- основний модуль;
- модуль графічного середовища користувача.

Інтерфейс – композиція функціональних компонентів.

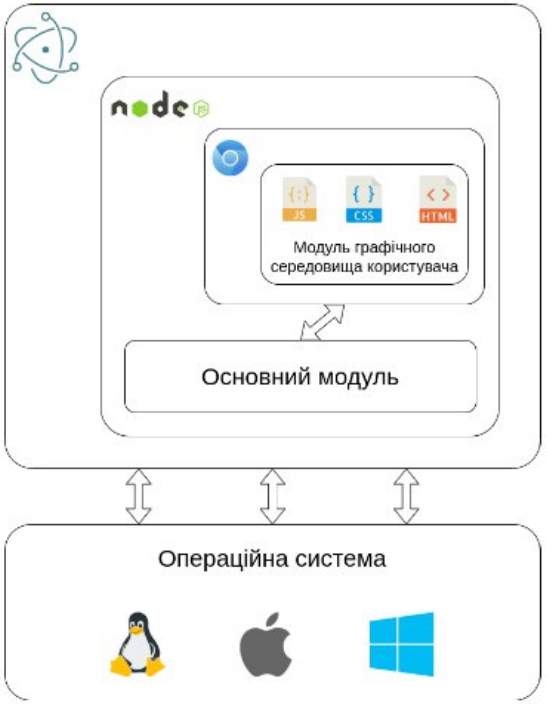
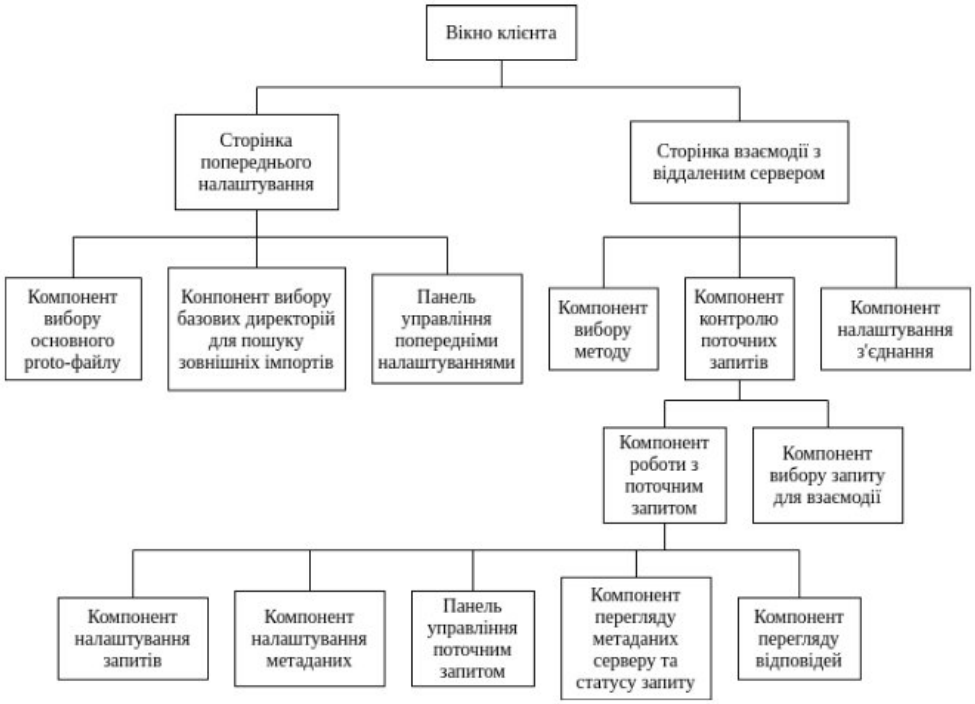
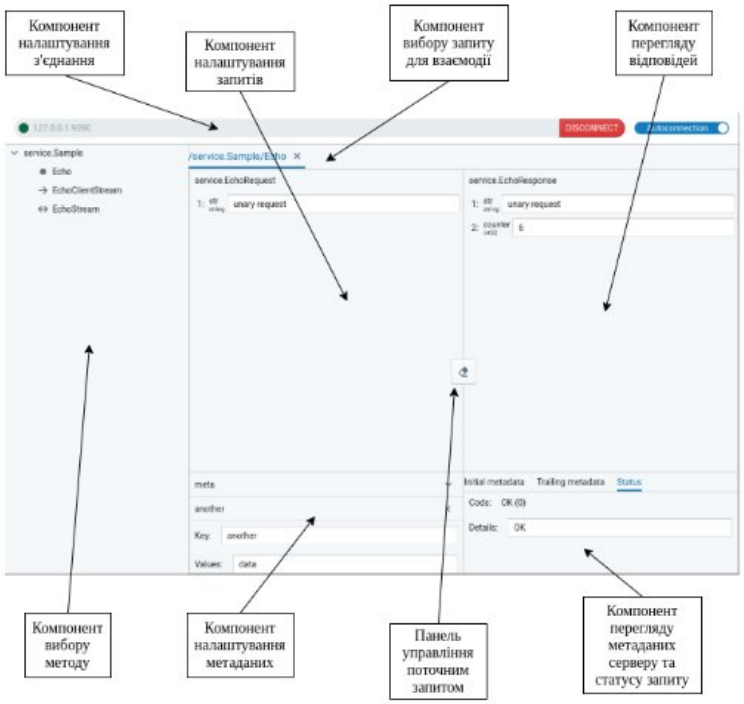


Схема композиції інтерфейсу користувача



Структура вікна проведення запитів



Тестування

White Box Testing.

Використовуються наступні методи:

1. Acceptance testing
2. Compatibility testing
3. Functional testing
4. Interface testing

Додатково реалізовується сервер для тестування.

Рекомендації щодо вдосконалення

- Впровадження підтримки протоколу TLS.
- Реалізація історії запитів.

Висновки

1. Проведено докладний аналіз існуючих рішень.
2. Складено список вимог до gRPC-клієнта.
3. Проведено аналіз технологій розроблення та мов програмування.
4. Розроблено програмний продукт для відлагодження серверного програмного забезпечення на базі технології gRPC.
5. Виконано тестування продукту у відповідності до затвердженої програми та методики.

Результат перевірки на унікальність

4.33% Схожість

Найбільша схожість: 0.56% з джерело бібліотеки. ID файлу: 1000041589

2.17% Схожість з Інтернет джерелами	126	Page 63
4.21% Текстові збіги по Бібліотеці акаунту	404	Page 64

0.31% Цитат

Цитати	2	Page 65

Вилучення переліку посилань вимкнено

0% Вилучень

Вилучений текст відсутній

Дякую за увагу!

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2019 р.

GRPC-КЛІЄНТ ДЛЯ ВІДЛАГОДЖЕННЯ СЕРВЕРНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Програма та методика тестування

ДП.045440-04-51

«ПОГОДЖЕНО»

Керівник проєкту:
_____ Микола ОНАЙ

Нормоконтроль:
_____ Микола ОНАЙ

Виконавець:
_____ Павло РУХАЙЛО

2019

ЗМІСТ

1. ОБ’ЄКТ ВИПРОБУВАНЬ

gRPC-клієнт для відлагодження серверного програмного забезпечення, який являє собою прикладне програмне забезпечення, створене за допомогою технології Electron з використанням фреймворку React.

2. МЕТА ТЕСТУВАННЯ

У процесі тестування має бути перевірено наступне:

1) коректність роботи програми;
2) відповідність вимогам Технічного завдання;
3) зручність інтерфейсу користувача у використанні;
4) робота клієнта у середовищах операційних систем Windows 10, Mac OS та Linux.

3. МЕТОДИ ТЕСТУВАННЯ

Тестування виконується методом White Box Testing. В процесі тестування перевіряється як код, так і безпосередньо програмний продукт на відповідність висунутим функціональним вимогам.

Використовуються наступні методи:

1) Acceptance testing – тестування на відповідність висунутим функціональним вимогам.
2) Compatibility testing – тестування сумісності з операційними системами Windows 10, Mac OS та Linux.
3) Functional testing – тестування коректності виконання всіх необхідних функцій.
4) Interface testing – тестування зручності інтерфейсу.

4. ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Працездатність прикладного програмного забезпечення перевіряється шляхом:

1) динамічного ручного тестування – введенням граничних та недопустимих значень в поля, які можна редагувати;
2) динамічного ручного тестування на відповідність функціональним вимогам;
3) статичного тестування коду;
4) тестування клієнта в середовищах різних операційних систем;
5) тестування стабільності роботи при різних умовах;
6) тестування зручності використання;
7) тестування інтерфейсу.

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

«ЗАТВЕРДЖЕНО»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___» _____ 2020 р.

GRPC-КЛІЄНТ ДЛЯ ВІДЛАГОДЖЕННЯ СЕРВЕРНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Керівництво користувача

ДП.045440-05-34

«ПОГОДЖЕНО»

Керівник проєкту:
_____ Микола ОНАЙ

Нормоконтроль:
_____ Микола ОНАЙ

Виконавець:
_____ Павло РУХАЙЛО

2020

ЗМІСТ

1. Опис структури інтерфейсу програмного забезпечення

Програмне забезпечення складається з 2-х вікон :

- вікно налаштування типів даних та сервісів;
- вікно проведення запитів.

Вікно налаштування типів даних та сервісів призначена для встановлення шляху до основного proto-файлу, а також для встановлення базових директорій для пошуки зовнішніх імпортів. Складається з наступних компонентів:

- компонент вибору основного proto-файлу;
- компонент вибору базових директорій для пошуку зовнішніх імпортів;
- панель управління попередніми налаштуваннями.

Вікно проведення запитів призначена для роботи з віддаленим сервером. Складається з наступних елементів:

- компонент вибору методу;
- компонент контролю поточних запитів;
- компонент налаштування з'єднання.

2. Процедура попереднього налаштування

Вікно попереднього налаштування складається з компонента вибору proto-файлу, списку компонентів вибору директорій та панелі кнопок керування (рис. 1).

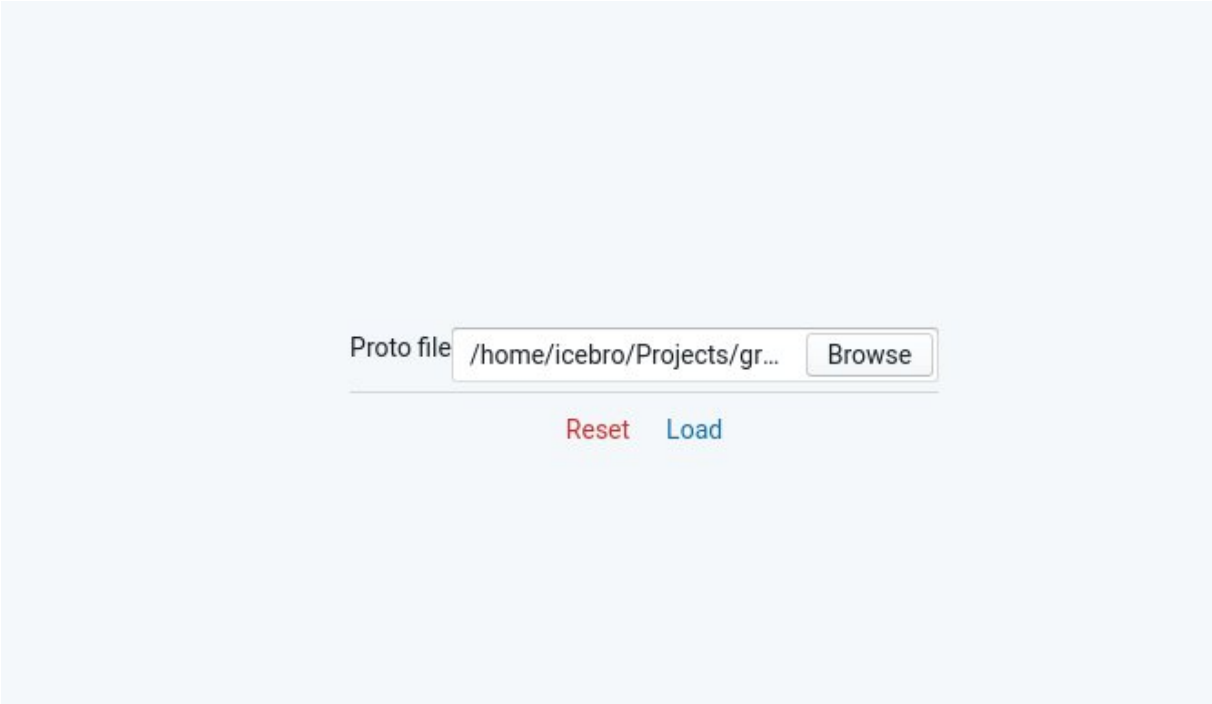


Рис. 1. Вікно попереднього налаштування

Компонент вибору файлу має вигляд текстового поля з кнопкою «Browse» праворуч. При натисканні на компонент вибору файлу відкривається системне діалогове вікно для вибору основного proto-файлу. Дозволяється вибір лише одного файлу з розширенням «proto». У разі, якщо немає обраного proto-файлу, поле вибору файлу заповнене текстом «Choose proto file...», у протилежному випадку – абсолютним шляхом до обраного файлу.

Список компонентів вибору директорій використовується для встановлення базових директорій пошуку зовнішніх імпортів в основному proto-файлі. Має вигляд вертикального списку. Елементами даного списку є компоненти, візуально подібні до компонента вибору proto-файлу, з

додатковою кнопкою, що має вигляд хрестика, праворуч. Додатково, останнім елементом списку є кнопка, що має вигляду знаку «+». При натисканні на один з компонентів списку відкривається системне діалогове вікно для вибору директорії. Дозволяється вибір декількох директорій. У випадку вибору декількох директорій для кожної з них буде створено окремий елемент списку. У випадку відкривання діалогового вікна шляхом вибору елементу, що відображує вже обрану директорію, при виборі іншої початкова директорія видаляється зі списку.

Панель кнопок керування складається з двох кнопок.

Перша кнопка є кнопкою очищення поточного стану вибору. Заповнена текстом червоного кольору «RESET». При натисканні очищує заповнену інформацію про обраний proto-файл та обрані директорії пошуку зовнішніх імпортів.

Друга кнопка є змінною. Так, при виборі шляхів для аналізу дана кнопка має синій колір та заповнена текстом «LOAD». Слід зауважити, що кнопка є вимкненою, поки не вибрано шлях до основного proto-файлу. При натисканні на дану кнопку запускається процес аналізу proto-файлів. Під час цього кнопка стає вимкненою та має вигляд індикатору завантаження.

У разі виникнення помилки на етапі аналізу показується спливаюче повідомлення у верхній частині вікна з інформацією про помилку. Повідомлення має стандартний для бібліотеки blueprintjs вигляд. Індикатор завантаження набуває вигляду кнопки «LOAD».

У разі успішного кнопка «LOAD» замінюється на кнопку зеленого кольору з текстом «COMPLETE». При натисканні на цю кнопку відбувається перехід на сторінку з основним інтерфейсом програми.

3. Процедура налаштування з’єднання з сервером

Компонент налаштування з’єднання складається з таких елементів, як індикатор, поле вводу адреси, кнопки та перемикача процесу автоматичного поновлення з’єднання (рис. 2).

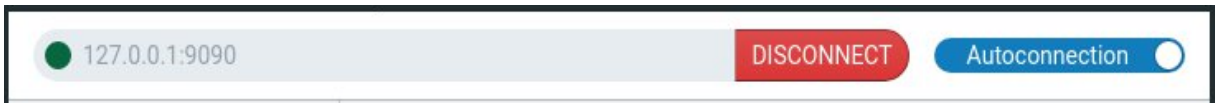


Рис. 2. Компонент налаштування з’єднання

Поле вводу слугує для вводу шляху до віддаленого сервісу. При вводі символів постійно валідує введену строку. У разі виявлення невідповідності веденої строки форматам можливих адрес поле підсвічується червоним, а у разі відповідності – зеленим, сигналізуючи про валідність вводу. Поле не активне у разі, якщо в поточний момент часу вже існує з’єднання з сервером незалежно від його статусу.

Індикатор працює у двох режимах. Так, якщо з’єднання с сервером не встановлене (тобто поле вводу є активним), забарвлення індикатору повторює колір підсвітки поля вводу адреси. У разі існуючого з’єднання індикатор має забарвлення, що відповідає певному статусу з’єднання, а саме:

- сірий колір відповідає статусу IDLE;
- синій – CONNECTING;
- зелений – CONNECTED;
- помаранчевий – TRANSIENT_FAILURE;
- червоний – SHUTDOWN.

Кнопка в компоненті з інформацією про з’єднання може працювати у двох режимах. Так, у разі відсутності з’єднання з сервером ця кнопка має зелений колір та заповнена текстом «CONNECT». При натисканні ініціалізує процес створення з’єднання з сервером. Слід зауважити, що

кнопка неактивна у разі виявлення помилок у введеній адресі. У разі присутності з’єднання з сервером дана кнопка має червоний колір з підписом «DISCONNECT», натискання на яку ініціалізує процес роз’єднання з сервером.

Перемикач в компоненті з інформацією про з’єднання слугує для налаштування процесу автоматичного поновлення з’єднання з віддаленим сервісом у разі помилок. Може знаходитись в двох станах: ввімкнений та вимкнений, що свідчить про поточне налаштування даного налаштування. У вимкненому стані містить в собі текст «no autoconnection», а при ввімкненому – «autoconnection».

4. Процедура проведення запитів

Основний інтерфейс програми складається з таких компонентів: компонент вибору поточного методу сервісу ліворуч, компонент з інформацією про з’єднання зверху праворуч та компоненту проведення запиту праворуч.

Компонент вибору поточного методу сервісу має вигляд дерева списку елементів, кожний з яких відповідає конкретному сервісу, що описаний в proto-файлах (рис. 3). При натисканні на один з елементів він розкривається та показує список методів обраного сервісу. Слід зауважити, що в конкретний момент в часі розкритими може бути необмежена кількість елементів списку сервісів. При натисканні на елемент зі списку методів сервісу компонент для проведення запитів починає працювати з обраним методом.

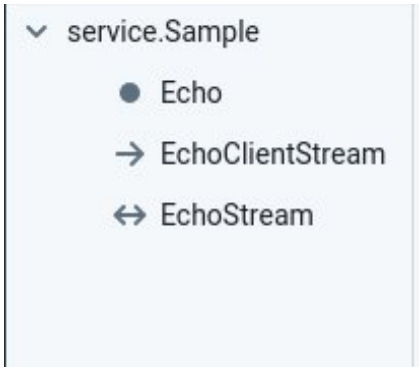


Рис. 3. Компонент вибору поточного методу

Компонент контролю поточних запитів складається з таких частин: списку поточних запитів та компоненту роботи з певним запитом, який в свою чергу складається з компоненту роботи з потоком запитів, компоненту роботи з потоком відповідей, компоненту роботи з метаданими клієнта, компоненту перегляду метаданих сервера та панелі управління запитом (рис. 4).

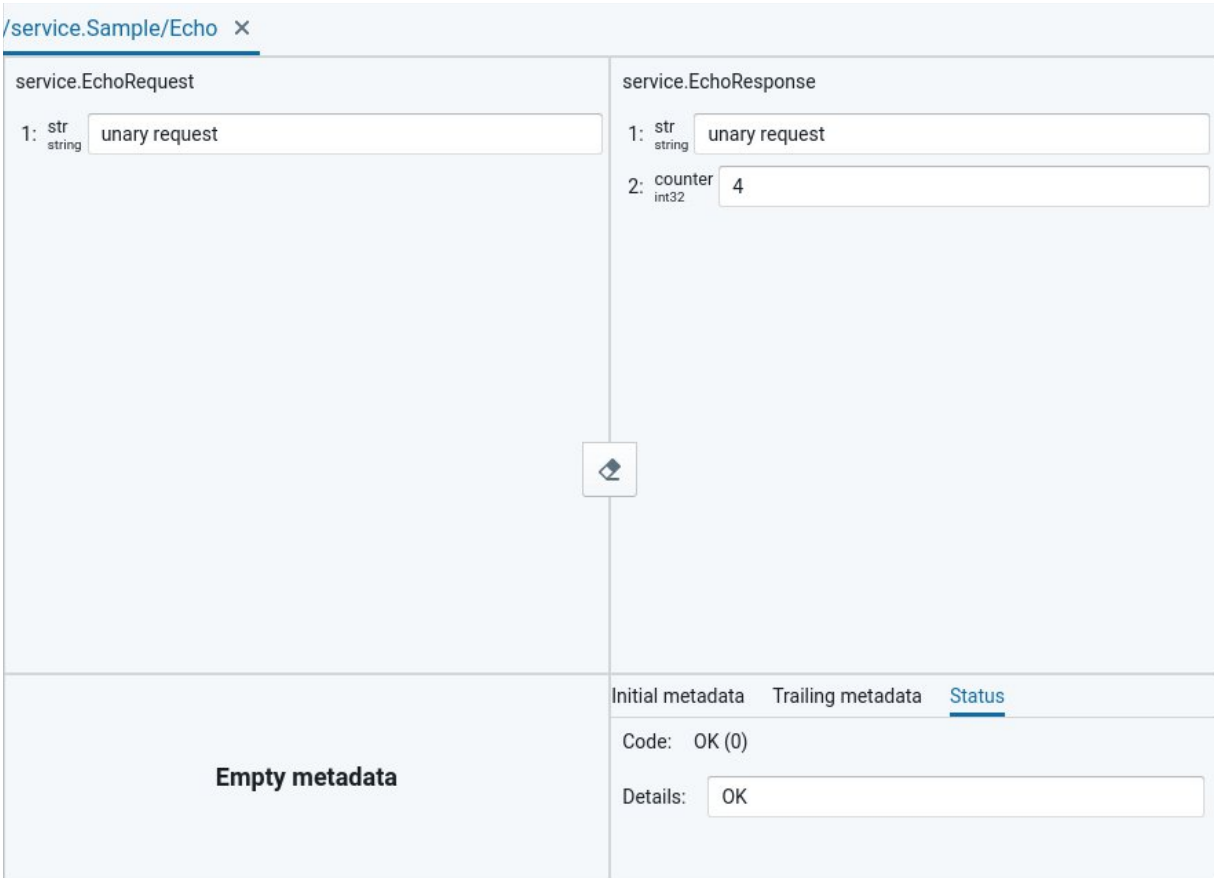


Рис. 4. Компонент контролю поточних запитів

Список поточних запитів має вигляд списку вкладок. При натисканні на кожну окрему вкладку інформація про конкретний запит відображується у компоненті роботи з запитом.Кожна вкладка може бути видалена зі списку натисканням на відповідну кнопку.

Для налаштування початкових метаданих від клієнта використовується компонент роботи з метаданими клієнта (рис. 5). Має вигляд списку елементів, кожний з яких відповідає за редагування даних за певним ключем та може бути розгорнутий окремо за допомогою відповідної кнопки. Також кожний такий елемент має кнопку для його видалення. Для редагування даних за конкретним ключем необхідно розгорнути відповідний елемент. В розгорнутому стані кожний елемент має поле для редагування самого ключа, а також список полів для зміни даних. Кожне окреме значення може бути видалене за допомогою відповідної кнопки. Є

кнопка для створення нового значення. Також компонент має кнопки для створення нового ключа та очищення даних.

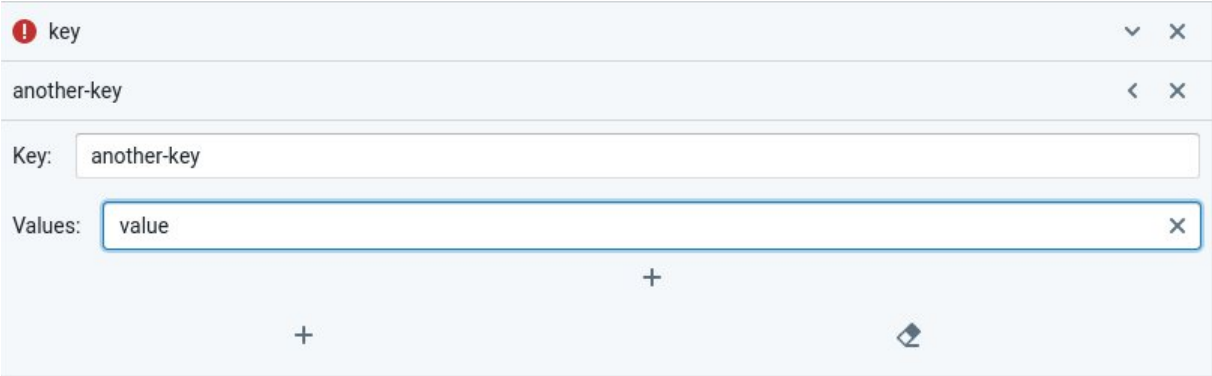


Рис. 5. Компонент роботи з метаданими клієнта

Компонент перегляду метаданих сервера та статусу виклику (рис. 6) є надбудовою над попереднім компонентом. Складається з списку вкладок та зони перегляду обраних метаданих. Список вкладок має статичне наповнення, а саме вкладки «Initial», «Trailing» та «Status». Перші дві вкладки відповідають за відображення метаданих сервера. Це пов’язано з тим, що сервер посилає метадані двічі: перед відправленням першої відповіді та після відправлення усіх відповідей в складі об’єкту-статусу виконання методу. Третя вкладка відображає вміст об’єкту статусу.

Рис. 6. Компонент перегляду метаданих сервера та статусу виклику

Панель управління запитом має різний набір кнопок в залежності від типу методу та поточного стану запиту (рис. 7). Так, при роботі з унарним методом дана панель буде мати два можливих варіанти наповнення: або

містити одну кнопку, натискання якої призведе до відправки запиту, або містити кнопку, що слугує для відміни поточного запиту.

При роботі з методами, що використовують потік даних зі сторони клієнту, панель може мати лише кнопку для початку роботи з методом, або ряд кнопок для контролю виконання поточного методу, а саме:

- кнопку для надсилання наступного запиту;
- кнопку для закінчення роботи з методом;
- кнопку для відміни роботи з методом.

При роботі з методами, що використовують потік даних зі сторони сервера панель управління має вигляд та функціонал, аналогічні до того, що використовуються при роботі з унарними методами, а при роботі з методами, що використовують потоки даних в обидві сторони – аналогічно до методів, що використовують потік даних зі сторони клієнту.

Компонент роботи з потоком запитів надає можливість перегляду опису типу даних, що використовується в обраному методі як запити та наповнювати їх даними для подальшого пакування та передачі на сервер. Компонент роботи з потоком відповідей з сервера надає можливість перегляду опису типу даних, що використовується в обраному методі як відповіді та перегляду даних, отриманих від сервера. Дані компоненти мають дуже схожий вигляд.