КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп’ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота

на здобуття ступеня бакалавра
за спеціальністю 122 Комп’ютерні науки
на тему:

АРХІТЕКТУРА ВЕБ-ЗАСТОСУНКУ НА ОСНОВІ ВІРТУАЛЬНИХ ПРОЦЕСІВ

Виконав студент 4-го курсу
Ставінський Андрій _________
(підпис)

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Ставровський Андрій Борисович _________
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _________
(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри теоретичної кібернетики
«____» _______________ 202_ р.,
протокол № ___
Завідувач кафедри
Ю. В. Крак _________
(підпис)

Київ – 2021

РЕФЕРАТ

Робота складається зі вступу, 5 розділів, висновків, списку використаних джерел (34 найменування). Робота містить 10 рисунків та 3 додатки. Загальний обсяг становить 52 сторінку, основний текст роботи викладено на 33 сторінках.

Ключові слова: ВЕБ-ЗАСТОСУНОК, ВІРТУАЛЬНИЙ ПРОЦЕС, ПЕРЕДАЧА ДАНИХ.

Об’єктом роботи є розробка архітектури передачі даних у веб-застосунках з використанням віртуального процесу.

В якості засобу розроблення прототипу бібліотеки було обрано мову програмування JavaScript та IDE JetBrains.

Метою роботи є створення підходу, який би спрощував написання коду та роботу з переміщення даних при розробці веб-застосунку.

Методи розроблення прототипу: написання коду та прикладів, використовуючи мову JavaScript – головний інструмент веб-розробки.

Результати роботи: проведено огляд стану розробки веб-застосунків, запропоновано перспективну архітектуру взаємодії, реалізовано прототип тестової бібліотеки, порівняно процес написання коду за класичною та за пропонованою архітектурою.

Отримані результати можна використовувати як теоретичну базу та заготовку для розробки бібліотеки для комерційної веб-розробки.

ЗМІСТ

РЕФЕРАТ 2
ЗМІСТ 3
ВСТУП 7
РОЗДІЛ 1 СУЧАСНІ ПІДХОДИ ДО ПОБУДОВИ АРХІТЕКТУРИ ВЕБ-ЗАСТОСУНКІВ 10
1.1 Можливості та задачі веб-платформи 10
1.1.1 Загальний огляд 10
1.1.2 Проблеми та обмеження веб-платформи 12
1.2 Типова методологія розробки веб-застосунків 14
1.3 Архітектури внутрішньої взаємодії веб-застосунків 16
1.3.1 Синхронна 16
1.3.2 AJAX 17
1.3.3 SPA 18
1.3.4 Мікросервісна архітектура веб-застосунку 19
1.4 Типові інструменти розробки веб-застосунків 20
1.5 GraphQL 21
1.6 Декларативне програмування 23
1.7 Змішення ООП та функціонального програмування 24
1.8 Ізоморфізм 24
1.9 Гібридні застосунки 25
1.10 Висновок 26
РОЗДІЛ 2 ПОНЯТТЯ ВІРТУАЛЬНОГО ПРОЦЕСУ 27
2.1 Поняття процесу 27
2.2 Поняття простору імен 27
2.3 Поняття віртуального процесу 27

2.4 Переваги, які може дати віртуальний процес при розробці ВЗ 29

РОЗДІЛ 3 АРХІТЕКТУРА ВЕБ-ЗАСТОСУНКУ З ВИКОРИСТАННЯМ ВІРТУАЛЬНОГО ПРОЦЕСУ ... 31

3.1 Старт віртуального процесу ... 33

3.2 Шлях передачі даних у віртуальному процесі ... 34

3.3 Робота зі змінною у спільному іменному просторі ... 34

3.4 Робота з функцією у спільному іменному просторі ... 35

3.5 Завершення віртуального процессу ... 35

3.6 Проблема гонки даних ... 36

3.7 Проблема неявної зміни даних ... 36

3.8 Проблема надмірної передачі даних ... 36

РОЗДІЛ 4 ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РЕАЛІЗАЦІЇ ... 38

РОЗДІЛ 5 ПОРІВНЯННЯ ПРИКЛАДІВ КОДУ НАПИСАНОГО НА ОСНОВІ РІЗНИХ ПІДХОДІВ ... 41

5.1 Запит на виконання серверної функції ... 41

5.2 Ізоморфна допоміжна функція (хелпер) ... 41

5.3 Ізоморфний об’єкт ... 42

5.4 Обробка запиту сервером ... 43

5.5 Висновок ... 43

ВИСНОВКИ ... 44

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ... 45

ДОДАТОК А ... 48

Код класу VirtualClient ... 48

ДОДАТОК Б ... 50

Код класу RemoteProcess ... 50

ДОДАТОК В 52
Код класів MemoryStoreClient та MemoryStoreServer......................... 52

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

ВЗ – веб-застосунок;

ВП – віртуальний процес;

ОС – операційна система;

СУБД – система управління базою даних;

ПІ – простір імен;

W3C – World Wide Web Consortium;

HTTP – hypertext transfer protocol;

API – application programming interface;

REST – representational state transfer;

DNS – domain name system;

PSN – PlayStation Network;

SPA – single page application;

SSR – server side rendering;

ВСТУП

Оцінка сучасного стану об'єкта розробки.

Веб-застосунки (ВЗ) є сьогодні найбільш затребуваною технологією для розробки застосунків з інтерфейсом користувача.

ВЗ виконується у віртуальній машині браузеру, тобто є ізольованими від операційної системи. Такий підхід забезпечує зразковий рівень безпеки платформи та можливості виконання у системах будь-яких архітектур.

При написанні клієнтської частини веб-застосунку використовуються стандартизовані мови та технології програмування. W3C (Консорціум всесвітньої павутини) відповідає за розробку нових стандартів [1].

Загальноприйнятий набір технологій для написання нових ВЗ складається з мов програмування JavaScript, CSS, HTML, бібліотеки React (або аналогічна) для клієнтської частини, фреймворку Express.js (або аналогічного) для серверної частини застосунку, та REST як архітектури взаємодії клієнта з сервером.

Актуальність роботи та підстави для її виконання.

Веб-платформа є найпопулярнішим інструментом для створення застосунків. Головна її перевага – дешевизна розробки і зручність розповсюдження.

Сьогодні перед веб-розробниками ставлять задачі по створенню застосунків будь-якого рівня складності. В усіх випадках мова йде про клієнт-серверні застосунки, важливу роль у яких відіграє переміщення даних між процесами.

Вартість розробки програмного забезпечення переважно складається з фонду оплати праці, тому підвищення швидкості реалізації функціоналу позитивно впливає не тільки на швидкість виходу на ринок, а ще й на вартість розробки.

Особливо актуальне питання зменшення вартості розробки постало коли IT-компанії були змушені почати самі собі ставити завдання. В такій ситуації

генерується велика кількість непотрібних рішень, а отже фірми несуть зайві витрати на оплату праці розробників.

Комерційними розробниками постійно приймаються спроби спростити та пришвидшити процес розробки, в тому числі написання інструментів та розробка нових парадигм роботи з даними.

Метою кваліфікаційної роботи є створення архітектури веб-застосунку, яка би реалізовувала ідею віртуального процесу та використовувала би її переваги.

Для досягнення цієї мети поставлені наступні **завдання:**

- дослідити та проаналізувати сучасний стан веб-розробки;
- дослідити поняття віртуального процесу;
- запропонувати архітектуру взаємодії клієнта та сервера веб-додатку, використовуючи віртуальний процес;
- розробити прототип бібліотеки;
- порівняти складність написання коду при використанні запропонованої та класичної архітектури взаємодії;

Об'єкт розроблення – архітектура веб-застосунку побудована на ідеї віртуального процесу.

В якості **засобу розроблення** тестової бібліотеки було обрано JetBrains WebStorm – інтегроване середовище розробки з багатьма доступними мовами програмування зокрема JavaScript, яке має пропрієнтарну ліцензію та розповсюджується за підпискою. JavaScript це динамічнотипізована мова програмування високого рівня з підтримкою кількох парадигм програмування. WebSocket – протокол зв'язку, який працює поверх TCP-з'єднання та дозволяє після встановлення підключення надсилати запити від клієнта до сервера і навпаки.

Можливі сфери застосування.

Запропонована архітектура може бути використана при розробці будь-яких ВЗ з помірною кількістю запитів між клієнтом і сервером. Особливий

виграш у спрощенні кодової бази буде отримано при реалізації мікросервісного підходу написання серверної частина застосунку.

Взаємозв'язок з іншими роботами.

Роботу було виконано на основі відомих знань та досвіду у веб-розробці, а також робіт про віртуальні процеси [2, 3].

РОЗДІЛ 1
СУЧАСНІ ПІДХОДИ ДО ПОБУДОВИ АРХІТЕКТУРИ ВЕБ-ЗАСТОСУНКІВ

1.1Можливості та задачі веб-платформи

1.1.1 Загальний огляд

Через свою кросплатформність веб-платформа є дуже економним способом створення інтерфейсу користувача для застосунків з різноманітною користувацькою базою, оскільки за замовчуванням дозволяє пере-використовувати код на усіх популярних платформах.

Велика популярність і гнучкість платформи зумовила стрімкий розвиток екосистеми: готових рішень, інструментів, практик та підходів, що ще сильніше зменшує вартість розробки програмних продуктів.

Всесвітня мережа є вільною площадкою для розміщення інформаційних ресурсів. Немає якогось власника платформи який встановлює правила хто може розміщувати застосунок в інтернеті, або погоджує код, дизайн, оновлення застосунку. На відміну від магазинів застосунків (AppStore, Google Play, Windows Store, PSN), або супераппів (WeChat, QQ) відсутня комісія за використання платформи та модерація.

Через дуже низьку собівартість функціонування ресурсів в інтернеті більшість компаній, які надають послуги розміщення ресурсів в мережі, а саме: хостери, реєстратори імен, DNS-провайдери, надаються можливість запускати додатки повністю безкоштовно, що дозволяє запускати комерційні проекти з будь-якої кількістю стартових ресурсів.

Початковою метою створення всесвітньої мережі було надання можливості доступу до віддалених інформаційних ресурсів. В основі моделі взаємодії лежить запит за універсальним ідентифікатором ресурсу – URI. За замовчуванням кожного разу при запиті завантажується і відображається ресурс. Пізніше ці ресурси стали інтерактивними, а згодом з'явилася можливість створювати повнофункціональні застосунки. Таким чином ВЗ

унаслідували модель розповсюдження з оновленням коду при кожному запуску доданку, що зменшує час розповсюдження оновлення від кількох днів у магазинах застосунків з модерацією оновлень, фактично до нуля.

Веб-платформа стрімко розвивається у напрямку поглиблення функціоналу. Так за останні роки відбулися наступні зміни:

- значно покращилися інструменти для створення інтерфейсів;
- швидкодія графічної підсистеми браузерів зрівнялася з багатьма нативними рішеннями;
- розроблено стандарти та засоби для роботи веб-застосунків без постійного підключення до мережі;
- з’явилися інструменти для просунутої роботи з тривимірною графікою;
- з’явилися інструменти виконання коду на графічному процесорі;
- з’явилися інструменти для створення peer-to-peer зв’язків між браузерами;
- відбувається постійний розвиток синтаксису основних мов веб-програмування з метою покращення виразності та лаконічності;
- відбувся перехід на реактивне програмування інтерфейсів;
- створені інструменти компіляції нативних програм для виконання віртуальних;
- створенні інструменти для розробки у будь-яких стилях та парадигмах програмування;
- суттєво покрашена швидкодія віртуальної машини JavaScript;
- надано доступ до деяких системних API, таких як геолокація, гіроскоп, захват екрану,
- створено та розповсюджено вбудовані в браузер безпечні засоби оплати (Google Pay, Apple Pay).

Загалом можна сказати, що веб-платформа лише збільшує власні можливості та гнучкість, при цьому спрощуючи створення додатків.

На сьогоднішній день не є проблемою створення повнофункціональних веб-застосунків. На сьогоднішній день вже працюють успішні веб-застосунки, які відносяться до наступних категорій:

- повноцінні веб-версії офісних пакетів [7, 8];
- графічні та відео редактори [9, 10];
- картографічні програми [11, 12];
- месенджери та програми для відео-конференцій [13, 14];
- САПР [15, 16];
- онлайн-банкінг [17, 18];
- системи обліку [19, 20];
- бази знань [21];
- розважальні платформи [22, 23];
- навчальні платформи [24];
- інтернет-магазини [25];
- дошки оголошень [26];
- ігри [27];
- тощо.

Отже перед веб-розробниками сьогодні ставлять задачу розробки повнофункціональних, надійних застосунків для будь-яких сфер життя та економіки.

1.1.2 Проблеми та обмеження веб-платформи

Веб-застосунок складається з клієнтської частини, серверної, та засобів взаємодії між ними. Архітектура всесвітньої мережі та стандарти W3C регламентують лише клієнтську частину і її взаємодію, тобто при створенні самого серверного функціоналу немає жодних обмежень ні на мови програмування, ні на апаратні платформи, ні на парадигми та підходи до програмування.

На противагу, задля забезпечення безпеки клієнтська частина має ряд суттєвих функціональних обмежень. Наприклад ВЗ не має доступу до файлової системи, лише до окремих файлів які передасть користувач. Усі дані запроповано зберігати у декількох дуже обмежених за розміром сховищах. Для забезпечення приватності та безпеки даних, доступ до цих сховищ надається лише коду завантаженому з того самого домену, що і створив ці дані.

Отримання коду при кожному зверненні до ресурсу породжує проблему надмірного використання мережі та чутливості до швидкості з’єднання. Для боротьби з негативними ефектами такого підходу застосовують наступні техніки:

- кешування незмінюваних ресурсів;
- завантаження ресурсів, в тому числі коду, за потреби;
- створення SPA, що дозволяє запускати бібліотеки та завантажувати сторонні ресурси лише один раз, та завчасно завантажувати сторінки та необхідних для них дані.

Іншою суттєвою проблемою є фрагментованність як пристроїв, так і інтернет підключення. Хоч і наполеглива праця зі стандартизації браузерів і дозволяє майже ніколи не мати проблем з правильністю виконання коду застосунку, проте потрібно писати код, який би працював на пристроях швидкодія яких відрізняється на порядки. Наприклад, Apple iPhone завжди отримує оцінки синтетичних тестів щонайменше у 2 рази вищі ніж флагманський пристрої на ОС Android того ж року [28]. Розрив з бюджетними пристроями, особливо тими, які на декілька років старші змушує розробників постійно працювати над оптимізацією та надоптимізацією коду, а іноді змушує відмовитись від реалізації складових застосунку з високою обчислювальною складністю. Іншою проблемою є різна швидкість підключення у різних регіонах та при різних сценаріях використання. Так бездротові мережі четвертого покоління забезпечують швидкості більші, ніж стара телефонна мережа у провінційних містах.

1.2 Типова методологія розробки веб-застосунків

Інформаційні технології допомагають людству у розв'язанні різноманітних проблем вже не перше десятиліття, через що очевидні способи застосування програмних продуктів вже закінчилися. В цих умовах бізнес змушений постійно шукати нові потреби для створення прибутку з їх задоволення.

Також особливістю сучасної розробки є принцип неперервності. Програмний продукт не застигає на великий проміжок часу в одній версії, а постійно змінюється. Досить розповсюдженою серед продуктів написаних на веб-платформі є практика впроваджувати мінорні зміни та виправлення щодня. Серйозні зміни впроваджують раз у фіксований проміжок часу, зазвичай це 2 тижні. Програмні продукти постійно еволюціонують протягом багатьох років.

Пошук ще нерозв'язаних прикладних задач та точок покращення існуючого функціоналу є перебором всіх проблем і засобів її розв'язання. Часто проблеми бувають надуманими, а розв'язки непрацюючими, або взагалі шкідливими. Саме тому велика кількість програмістів зайняті у написанні програмного коду, який виявляться нікому непотрібним.

Вартість розробки програмного продукту переважно складається з фонду оплати праці, саме тому бізнес намагається мінімізувати фінансові ризики від розробки непотрібних рішень шляхом зменшення зменшення часу потрібного на створення продукту та активним впровадженням прототипування. Набагато вигідніше зробити неякісну та/або урізану версію продукту/функціоналу аби перевірити гіпотезу про потрібність рішення, ніж витратити час і гроші на розробку повноцінного якісного рішення і з'ясувати що воно незатребуване.

Через свою гнучкість в оновленні та розповсюдженні коду та дешевизну розробки веб-платформа стала основним інструментом пошуку та розв'язання прикладних задач. За останні 15 років утвердилися дві домінуючи методології розробки: гнучка та експериментальна.

Гнучка розробка – методологія за якої кожна окрема частина функціоналу програмного продукту розробляється за своїм окремим циклом. Тобто специфікація, проектування, програмування, тестування, та впровадження виконується незалежно для кожного елементу системи. За цього підходу на початку розробки немає кінцевих вимог до продукту, оскільки в процесі розробки вимоги будуть постійно змінюватись. Гнучка розробка дозволяє рано отримати сиру, але робочу версію продукту і почати її впроваджувати. Підхід є дуже ефективним у випадках, коли програма створюється навмання, а саме коли немає розуміння чи вона взагалі комусь потрібна, і які проблеми має розв’язувати.

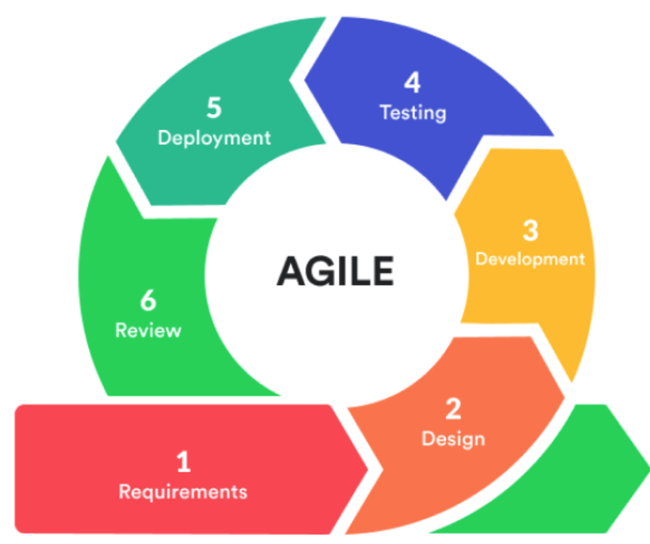

Рис. 1 Діаграма процесу гнучкої розробки [31]

Експериментальна розробка – розширення гнучкою методології, в якій також проводиться систематична робота з експериментального підтвердження ефективності змін у програмному продукті. Для цього встановлюються метрики за змінами яких буде прийматися рішення, наприклад: середньо тижнева виручка для інтернет-магазину, час проведений на сайті для соціальної мережі, тощо. Після визначення метрик проводиться A/B тест – тест в ході якого користувачі діляться на потрібну кількість груп, кожна з яких отримує свою версію програмного продукту. Підчас оцінюваного періоду використання програми збираються нові дані для розрахунку цільових

метрики, після чого проводить статистичний аналіз отриманих результатів. На основі отриманих даних приймається рішення щодо впровадження чи відкату змін. В ході експериментальної розробки доводиться писати багато коду, який буде видалений, тому щоб не марнувати час на якісну розробку непотрібного функціоналу – швидкість розробки важливіша за все інше. В ідеальному варіанті після підтвердження ефективності фунціонал переписується задля підвищення якості, проте типова швидкість розвитку веб-застосунків часто не дозволяє це зробити, тому на ринку дуже цінуються розробники, які вміють балансувати якість кодової бази та швидкість написання. Також за експериментального підходу стає у нагоді модель розповсюдження веб-застосунків:

- постійна актуалізація кодової бази;
- можливість гнучкого контролю яку версію віддавати;
- відсутність модерації.

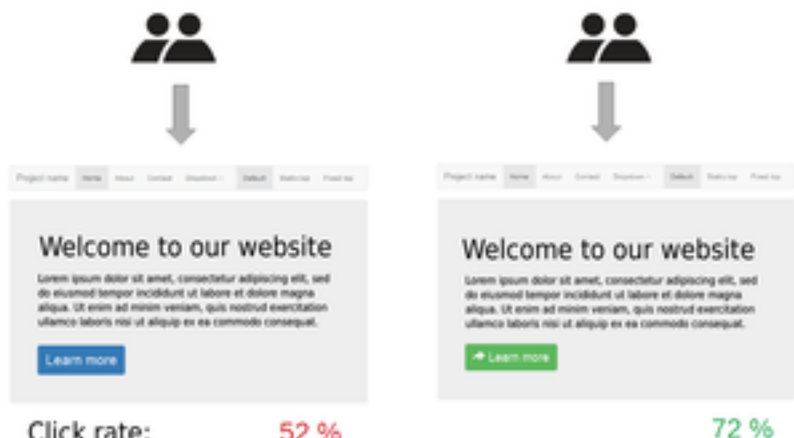

Рис. 2 Приклад A/B тесту змін інтерфейсу користувача [32]

1.3 Архітектури внутрішньої взаємодії веб-застосунків

1.3.1 Синхронна

Синхронна модель взаємодії використовує лише примітивні інструменти всесвітньої мережі, а саме мову розмітки HTML та HTTP запити. Взаємодія відбувається наступним чином: користувач виконує якісь дії на сторінці, у відповідь на що формується посилання на нову версію сторінки.

Параметри посилання та заголовки детермінують результат виконання. Усі запити до серверу (окрім обробки форм) робляться через GET метод, тому що це єдиний метод доступний з адресної строки браузеру.

Сьогодні не використовується, дуже рідко зустрічається на старих сайтах, написаних наприкінці двадцятого та початку двадцять першого сторіччя.

1.3.2 AJAX

AJAX (Asynchronous Javascript and XML) – підхід який використовує фоновий обмін даними. В основі підходу лежить можливість виконувати запити на сервер та маніпулювати деревом елементів без оновлення сторінки.

За цієї архітектури користувач отримує готову версію сторінки з серверу, проте вона може змінюватись – наприклад завантажувати нові пости у стрічку новин при прокрутці.

Для реалізації фонових запитів переважно використовують асинхронні HTTP-запити засобами мови програмування JavaScript. Це дозволяє повністю використовувати семантичні можливості HTTP-методів для покращення виразності серверного коду.

Можливість динамічної зміни сторінки дозволяє економити мережеві ресурси на відсутності завантаження незмінної частини сторінки. Також це покращує досвід користувача

Стандартом архітектури для написання серверної частини є набір правил та практик REST.

Для двонапрвленого зв'язку використовують реалізацію сокетів для веб-платформи – Websockets. Створення підключення за допомогою сокетів дозволяє серверу посилати запити на клієнт. Дуже широко використовується в онлайн чатах, браузерних онлайн іграх, системах моніторингу в реальному часі. На відміну від алгоритму long poling скоети мають мінімальну затримку оновлення даних, проте подієва модель значно ускладнює сприйняття та взаємодію коду.

Підхід досі використовується при створенні простих сайтів, таких як посадочні сторінки для рекламних компаній, сайти для промо акцій, навчальні проекти та інші одноразові речі. Дуже розповсюджений серед проектів, які були написані у 2005-2015 роках, і які ще не були переписані.

1.3.3 SPA

SPA (single page application) – це підхід в якому використовується єдина html сторінка як точка входу. Клієнтська частина розглядається як самостійна програма, яка перевизначає роботу з адресним рядком, історію браузера, а взаємодія з сервером відбувається засобами AJAX або мікросервісної методології.

SPA архітектура змінює значення адресного рядка – рядок у ньому тепер не посилання на ресурс/сторінку, а посилання на застосунок та опис його початкового стану. Так посилання в своїй ієрархічній частині містить домен – ідентифікатор застосунку та шлях до ресурсу – стан додатку.

Розглянемо ідентифікатор вигляду *https://myapp.com/settings/profile*. При класичному підході це означало би або *profile.html* у директорії */settings*, або *index.html* у */settings/profile*. Проте у SPA єдина точка входу, яку завжди віддає сервер, а шлях треба розуміти як опис початкового стану застосунку: розділ *settings*, вкладка *profile*. Або так: *«відкрити застосунок myapp.com в розіділі налаштувань профілю користувача»*.

Використання ідентифікаторів стану застосунку також перейняли мобільні платформи. Вони так само дозволяють за посиланням, наприклад, відкрити застосунок музичного стрімінгового сервісу з активною сторінкою альбому, або додатки соціальних мереж переходять на конкретний пост конкретного користувача.

Єдина точка входу економить час на завантаженні та запуску коду бібліотеки інтерфейсу користувача при переході по різних сторінках застосунку, проте призводить до зростання об’єму коду при початковому

завантаженні. Для боротьби з цим налаштовують code splitting – поділ коду на файли і завантаження частинами за потреби.

При розробці за SPA архітектурою функціонал застосунку ділять на між сервером та клієнтом керуючись наступними міркуваннями:

- безпека даних – усі запити на запис до СУБД мають проходити валідацію у довіреному процесі, а тому виконуються виключно на сервері;
- чуттєві дані, правильність розрахунку яких впливає не тільки на самого користувача, задля запобігання зловживання обробляються на сервері;
- взяття функцій від спільних даних (наприклад розрахунок статистики), які є спільними для багатьох користувачів, але за умови складного обчислення обробляються та кешуються на сервері;
- все інше виконується на клієнті.

1.3.4 Мікросервісна архітектура веб-застосунку

Мікросервісна архітектура веб-застосунку передбачає розділення серверної частини ВЗ на декілька окремих програм, які виконуються незалежно одна від одної.

Незалежність кожного окремого компоненту системи стає в нагоді при розробці великою командою, яка одночасно виконує багато задач. Можливість незалежного вибору технологій, незалежного оновлення та перезапуску компонентів дозволяє підкомандам витрачати менше часу очікуючи одна-одну.

Проте така архітектура ускладнює як серверну інфраструктуру, так і логіку взаємодії додатку. На великих проектах пишуться бібліотеки, які скривають від розробника особливості технічної реалізації.

Також існує практика створення мікрофронтендів – мікросервісів у складі клієнтської частини застосунку. Мікрофрентд – повністю незалежний від сторінки елемент. Збірка та композиція мікрофроентдів у цілісну сторінку

є дуже затратним процесом, тому ця практика використовується лише на дуже надвеликих проектах.

1.4 Типові інструменти розробки веб-застосунків

При розробці клієнтської частини використовують засоби шаблонізації. Вони дозволяють інкапсилювати логіку низького рівня разом з розміткою та стилями в єдиний компонент, що спрощую як підтримку та перевикористання, так і дозволяє проводити тестування окремих компонентів. Усі ці інструменти використовують власне дерево елементів і проводять операції над ним, а об'єктне дерево, яке підтримує браузер є всього лише віддзеркаленням. Найпопулярнішими рішеннями є React.js, Angular.js, Vue.js. Вони мають вбудовані інструменти для виконання на сервері що дозволяє робити рендерінг розмітки на серверній стороні, проте для цього потрібно мати JavaScript оточення на сервері, таке як Node.js.

Необхідність використання Node.js як рендер-серверу, а також можливість перевикористання коду стали причинами виходу Node.js на перше місце за популярністю серед серверних платформ [29]. Навколо неї створено дуже обширну екосистему, в якій є готові рішення на будь-який випадок, наприклад: засоби для автоматичної збірки і запуску коду другими мовами програмування, адаптери до усіх існуючих СУБД, пакетні менеджери, SaaS-хостінг платформи, інструменти для адміністрування та аналітики процесу виконання, тощо.

Node.js позбавлений високорівневого функціоналу з адміністрування веб-серверів, тому для зручного написання серверного API завжди використовують серверні фреймворки, зазвичай, Express.js або Koa.js. Ці фреймворки беруть на себе ідентифікацію підключення, роботу з сесіями, надають зручні інструменти для написання роутінгу та обробки запитів (парсинг запиту, нормалізація даних, валідація).

Необхідність перевикористання компонентів на принципово різних системах, масове використання синтаксичного цукру та розширенн мов веб-

програмування, використання пакетних менеджерів зумовили перехід на компільований принцип веб-розробки. Щонайменше клієнтську частину сучасного веб-застосунка неможливо запустити без попередньої компіляції, а використання серверного рендерінгу розмітки змушує компілювати навіть серверну частину. Для компіляції використовують збірник модулей Webpack, який розширюють компіляторами потрібних розширень мов веб-програмування.

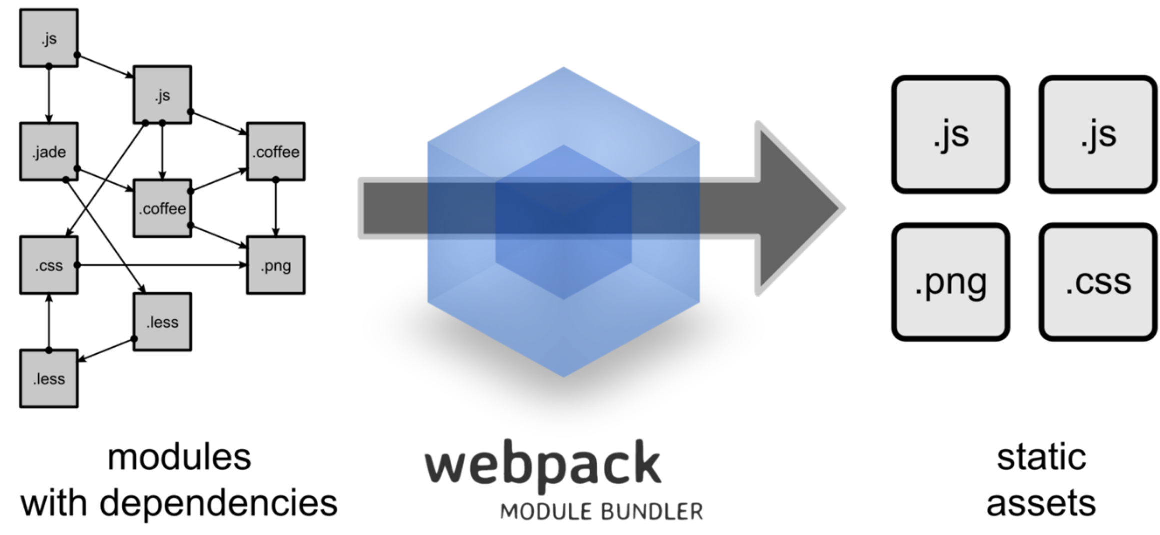

Рис. 3 Схема роботи збірника Webpack [33]

Стрімкий ріст складності інфраструктури веб-розробки призвів до створення фреймворків повного циклу, які беруть на себе збірку проекту, аутентифікацію користувачів, спрощують роутінг. Ці фреймворки можуть надавати готові канали передачі даних з клієнта на сервер та навпаки, давати спрощену обгортку для роботи з СУБД, надавати SaaS-хостінг без потреби в налаштуванні, робити автоматичний менеджмент ресурсів на клієнті. Прикладами таких фреймворків є Meteor.js, Next.js, Nuxt.js.

1.5 GraphQL

Сучасні веб-застосунки працюють над складними і гнучкими структурами даних, що зумовило широке запровадження

документоорієнтованих БД. Їх використання спрощує роботу з структурами з багатьма рівнями вкладення, проте створює додаткові складнощі з контролем доступу при обробці запитів. На великих та довготривалих проектах також є розповсюдженим сумісне використання принципово різних СУБД: ключ-значення, табличних, документоорієнтованих, спеціалізованих.

Складнощі з наданням доступу до таких розрізнених ресурсів спонукали до створення уніфікованої системи доступу, яка би спрощувала написання клієнтського коду. Таким рішенням є мова запитів GraphQL та екосистема клієнтів, які її використовують. GraphQL – альтернатива REST API, вона надає уніфікований спосіб запитів до даних для системи на будь-яких платформах. Замість створення специфічних точок отримання даних створюється опис колекцій даних та систем контролю доступу до них, що дозволяє декларативно описати агрегацію даних з різних джерел на специфічний запит.

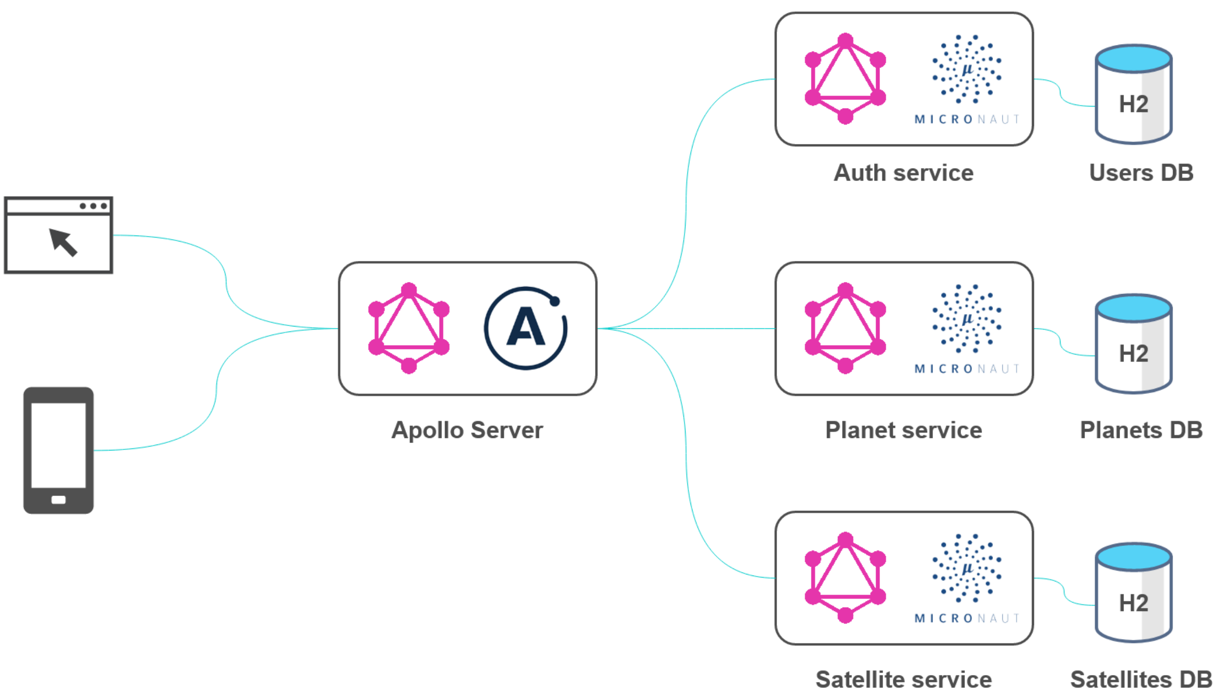

Рис. 4 GraphQL для запиту змішаних даних [34]

Такий підхід стає у нагоді при запиті частини великої структури даних, як от профіль користувача: для мініатюри над постом потрібно витягнути лише аватар, ім’я та посилання на профіль, а на сторінці повний документ. При

цьому дані останнього відвідування зберігаються у SQL-подібній СУБД, дані профілю у документоорієнтованій, а кількість лайків під постами у БД ключ-значення. GraphQL дозволяє створити лише одну точку видачі, спростивши написання серверної частини.

Негативною стороною використання GraphQL є те, що точки видачі даних можуть ставати дуже складними. Мова запитів маловиразна та є ускладненням порівняно з класичними REST запитами. Використання GraphQL недоцільно на проектах з однорідною структурою даних.

1.6 Декларативне програмування

Трендом сучасної веб-розробки є перехід на декларативне програмування. Декларативне програмування – парадигма за якої програміст описує не спосіб отримання рішення, а те який вигляд воно повинно мати.

Раніше головним прикладом була верстка за допомогою мов HTML та CSS – вони описували вигляд сторінки, а браузер сам розбирався з тим як розкласти елементи та зафарбувати пікселі. Сьогодні на зміну HTML та CSS прийшли шаблонізатори і препроцесори, які збагачують функціонал мов розмітки, а потім компілюються в HTML+CSS+JavaScript.

Використання компонентного підходу і інкапсуляції стану і логіки дозволяє поглибити використання декларативного програмування у створенні інтерфейсів. Наприклад тепер, коли потрібно відкрити модальне вікно немає потреби писати скрипт, який би програв анімацію. Більш того відкрити вікно чи ні вирішується не на основі команди, а на основі стану компоненту, наприклад, результату обчислення логічної функції від стану. При використанні готових бібліотек компонентів імперативне програмування зведено до мінімуму і полягає переважно у створенні запитів на сервер та обробці їх результатів.

Так само декларативне програмування впроваджується і у серверній розробці. Першим етапом було створення зручних інструментів роутінгу, коли

розробник не торкався технічної частини розбору запиту, а описував шляхи та функції які надавали ресурс за заданим шляхом.

Наступним кроком була розробка того самого стандарту запитів GrapgQL. При обробці запитів на результат описується як композиція результатів функцій отримання полів. В більшості випадків це органічно спрощує збірку складної структури даних до композицію декількох простих запитів ресурсів.

1.7 Змішення ООП та функціонального програмування

ООП (об’єктно орієнтоване програмування) – парадигма програмування у якій усі сутності подаються як екземпляри деякого класу. Програміст описує клас об’єктів та операцій над даними, а потім працює з ними. Екземпляр класу інкапсулює дані і методи для їх обробки.

Функціональне програмування – парадигма, за якої програма описується як композиція функцій в математичному сенсі.

У веб-розробці завжди було стандартною практикою писати гібридний код, який оперує як екземплярами класів, так і функціями. Це пов’язано в тому числі з системою роботи з об’єктами у мові програмування JavaScript. До 2015 року в мові не було звичного синтаксису для створення класів, проте був дуже зручний спосіб створення об’єктів без класів – літералів. Хоча до них можна додавати методи і за допомогою фабричного методу створювати екземпляри класу, але звичною практикою стало тримати обробку даних окремо від них.

1.8 Ізоморфізм

Використання однієї мови програмування як на клієнті так і на сервері породило нове поняття – ізоморфний код. Ізоморфний – це такий код, який без змін може виконуватися як на клієнті, так і на сервері. Першими прикладами стали шаблони, які ренедерилися на сервері. Ізоморфізм – дуже зручний і прогресивний спосіб перевикористання коду.

1.9 Гібридні застосунки

На ринку існують так звані гібридні застосунки, які в повному обсязі використовують технології веб-програмування поряд з технологіями нативної розробки.

Перша категорія – застосунки, в яких веб-застосунок обгорнуто в нативну оболонку, надаючи повний доступ до системних API. Прикладом є фреймворк Electron.js, на якому працюють такі відомі проекти як Slack, VisualStudio Code, WhatsApp [30]. Фактично це сайти, запущені в браузерах без панелі керування. Також через модель розповсюдження є доцільним надати доступ до всіх системних бібліотек, що реалізовано у вбудованих бібліотеках Electron. Такі застосунки за своїми можливостями рівні нативним, проте поєднують дешевизну розробки з порівняно низькою швидкодією.

Друга категорія – нативні застосунки, в яких лише частина функціонала реалізована з використанням веб-стеку. Прикладами таких застосунків є Apple Music для MacOS та CleanMyMac від MacPaw, Spotify для iOS. В усіх наведених застосунках більша частина інтерфейсу використовує нативний UIKit, проте окремі елементи зроблені як веб-сторінки і безшовно вбудовано в інтерфейс. Так у Apple Music каталог музики є веб-сторінкою, яку можна гнучко змінювати під нові формати представлення матеріалу, промо акції, спеціальні події, тощо. У CleanMyMax за допомогою веб-стеку реалізовано процес навчання і допомогу до програми, що дозволяє гнучко оновлювати ці матеріали без модерації з боку магазину додатків. Spotify у мобільному додатку використовує веб-інтерфейс для налаштування підписок і акаунту, оскільки такий функціонал використовується надзвичайно рідко – тому було просто вставлено вже існуючу веб-сторінку.

Єдина причина робити гібридні застосунки – так швидше і дешевше. Вони змушують розробляти та підтримувати шар взаємодії двох частин додатку, сильно погіршують користувацький досвід.

1.10 Висновок

Сучасні веб-застосунки мають дуже багату функціональність, через що росте розмір кодової бази та навантаження на мережі. Для зменшення негативних ефектів на користувача використовують SPA архітектуру веб-застосунку і технології SSR та code splitting.

Використання SPA архітектури на клієнті призвело до використання клієнтських JavaScript фреймворків, задля ізоморфного виконання яких стався перехід на сервер Node.js.

Використання SPA архітектури разом з ростом функціональності призвело до стрімкого ускладнення кодової бази веб-застосунків.

Постійно ведуться роботи щодо спрощення процесу розробки. Заради зменшення навантаження на розробників компанії готові жертвувати швидкодією, якістю коду, якістю фінального продукту.

Отже доцільним є розробка нової спрощеної у роботі архітектури взаємодії клієнтської та серверної частини, навіть попри можливі проблеми зі швидкодією.

РОЗДІЛ 2
ПОНЯТТЯ ВІРТУАЛЬНОГО ПРОЦЕСУ

2.1 Поняття процесу

Процесом у програмуванні називають процес виконання програми. Він розміщується в оперативній пам’яті. До нього належить код та оброблювані дані [4]. Оскільки пам’ять комп’ютера має адресацію, процес розміщується в адресному просторі. Команди та значення мають адреси, за якими їх можна використовувати. Команди та значення розрізняються процесором тільки під час виконання коду. Після завершення процесу пам’ять вивільняється й адресний простір зникає. Проте в інтерпретованих програмах наявність адресного простору зазвичай є схованою від розробника. В таких процесах ідентифікація змінних відбувається за іменами й, відповідно, змінні розміщуються не в адресному просторі, а абстрагуються в простір імен.

2.2 Поняття простору імен

Простором імен (ПІ) називають сукупність посилань та їх ідентифікаторів. У контексті програмування також потрібно включити у визначення синтаксис обробки посилань, оскільки може використовуватися як вбудований в мову програмування синтаксис роботи зі змінними, так і будь-яка абстракція яка реалізує наступні операції:

- створення;
- видалення;
- встановлення значення;
- взяття значення змінною.

Посилання може бути спрямовано як на змінну, так і на функцію.

2.3 Поняття віртуального процесу

Раніше було запропоновано наступні визначення віртуального процесу:

- абстрактна інтерпретація процесу є поняття віртуального процесу, який базується на однопотоковій моделі, та має можливість паралельного виконання. Планування виконання компонентів системи виконує модуль управління та передає управління. Планування може здійснюватися періодично або на подієвій моделі. Компоненти, виконання яких заплановано, відповідають за виконання запланованих дій і повертають управління в компонент управління. Ця модель заснована на логічному розкладанні дій на прості кроки, для виконання яких потрібні тільки короткі проміжки часу. Відповідальність за розбиття дій компонентів відповідно до вимог додатку покладається на розробника [2, 5, 6].
- Віртуальний процес – це сукупність кодів та значень, що можуть виконуватися та оброблятися на різних, навіть архітектурно, обчислювальних системах з метою розв'язання певної задачі за певним алгоритмом [2]. Також було наведено наступні властивості віртуального процесу:
  - спрямованість на алгоритмізовану обробку значень;
  - процес є атомарним віртуальним процесом;
  - два віртуальні процеси, які обробляють спільні значення утворюють віртуальний процес;
  - програмний код віртуального процесу може генеруватися та змінюватися всередині самого віртуального процесу;
  - знищення простору імен призводить до знищення віртуального процесу.

Як можна побачити, сучасні підходи до взаємодії клієнта та сервера, такі як REST та GraphQL, у ВЗ є прикладом ВП, але тільки за першим визначенням, оскільки атомарні процеси (клієнтська та серверна частини, СУБД) не мають спільного простору імен. Взаємодія за такою архітектурою відбувається через інтерфейси.

Для того, щоб ВЗ став ВП другого типу, потрібно щоб браузерний та серверний процес мали спільний простір імен. Першим комерційно-успішним

проектом, який частково реалізовував спільний простір імен був Meteor.js, а саме розробники реалізували доступ до СУБД з клієнту та можливість напряму викликати функції абстрагувавшись від створення HTTP запитів. Проте запис напряму з клієнта є небезпечним, тому цей функціонал відключають і залишається лише можливість підтягувати на клієнт Mongo-колекції без запитів.

Очевидно, що для повноцінності ВП і сервер і клієнт повинні мати можливість викликати методи і записувати змінні один одного.

Для забезпечення безпеки даних на сервері спільний іменний простір має бути відокремленим від СУБД та бути сховищем ключ-значення для значень примітивних типів, або структур з примітивних типів. Усі виклики від імені серверу є за визначенням безпечними, а тому не потребують валідацію вхідних даних.

2.4 Переваги, які може дати віртуальний процес при розробці ВЗ

Поняття віртуального процесу дозволяє абстрагуватися від фактичної реалізації атомарних процесів та реалізації зв'язків між ними, а отже спростити видиму логіку додатку. В такому випадку для досягнення результату розробник пише менше коду, що, очевидно, підвищує його ефективність та зменшує кількість помилок.

Використання віртуального процесу абстрагує програміста від написання мережевих запитів та робить код лаконічнішим і виразнішим.

Дуже суттєве спрощення логіки і процесу написання виходить у разі використання мікросервісної архітектури, тобто підходу, коли клієнт під’єднується до декількох серверів (процесів), кожен з яких виконує власну ізольовану функцію. Спільний простір імен для змінних та функцій дозволить не думати розробнику над технічними деталями запиту, а саме на який з серверів відправити запит, яким методом.

Окремо слід зазначити, що факт суттєвого спрощення логіки несе у собі неоціненні продуктові переваги:

- зменшення когнітивного навантаження на розробника, що полегшує підтримку, відладку, введення нових розробників, та зменшує ризики професійного вигоряння;
- суттєве зменшення кількості унікальних тестів, які потребує проект, що значно зменшує час на розробку та зменшує витрати;
- відсутність необхідності реалізувати складну міжсистемну взаємодію дозволяє писати якісний, підтримуємий та надійний код розробникам з меншою кваліфікацією, що також суттєво знижує витрати особливо у довгостроковій перспективі;
- зменшення часу на доставку нового функціоналу, запобігання збільшенню строків розробки з часом.

РОЗДІЛ 3
АРХІТЕКТУРА ВЕБ-ЗАСТОСУНКУ З ВИКОРИСТАННЯМ ВІРТУАЛЬНОГО ПРОЦЕСУ

Архітектура веб-застосунку з використанням віртуального процесу ґрунтується на ідеї створення спільного для клієнта та сервера/серверів іменного простору зі змінними та функціями.

Архітектура віртуального процесу регламентує лише спосіб взаємодії між клієнтським процесом і серверними, але ніяк не регламентує архітектуру будь-якого окремого процесу, або структуру даних, тощо.

Змінні в іменному просторі – будь-які примітивні типи та структури даних, які можуть бути серіалізовані в JSON. Для підвищення швидкодії доступу змінні копіюються у кожен програмний процес у складі віртуального процесу. При передачі між складовими процесами віртуального процесу усі дані трансформуються у JSON. Серіалізація потрібна хоча б тому, що не можна передавати змінні мови програмування між процесами в чистому вигляді – потрібна текстова інтерпретація. Також серіалізація у JSON дозволяє створити універсальну структуру даних, яка потім може бути десеріалізована у потрібний формат, що дозволяє комбінувати сервери написанні різними мовами програмування, наприклад, JavaScript та Python.

Функції доступні в спільному іменному просторі виконуються в тому програмному процесі в якому вони додані до простору, вони мають доступ до лексичного оточення, в якому створені. Доступ до оточення має на меті створити контрольований доступ до ресурсів іншого процесу, тобто клієнт таким чином може звертатися до СУБД-адаптера на сервері, проходячи валідацію вхідних параметрів. Функції є аналогом REST запити, але абстрагуються від того, де вони зберігаються, що дозволяє використовувати спільний код у різних програмних процесах.

Дані між клієнтом і серверами передаються за допомогою сокетів. Їх використання дає можливість серверу посилати запити на клієнт і є оптимальнішим з точки зору затримки, ніж алгоритм long polling.

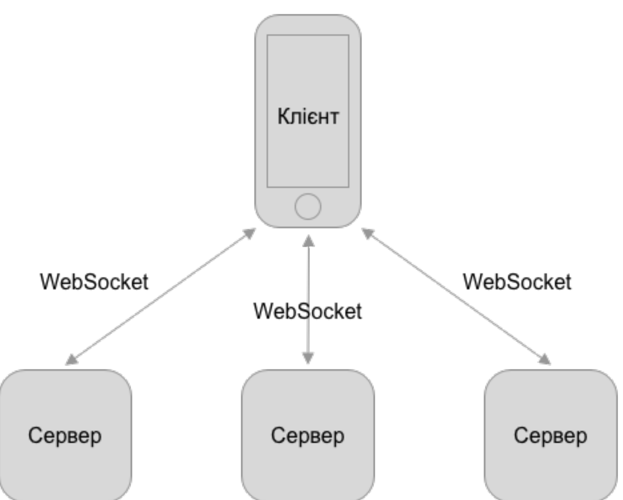

Рис. 5 Схема з'єднань між процесами.

Для забезпечення безпеки даних та підвищення швидкодії у спільний іменний простір виноситься лише частина даних. За замовчування – нічого. Розробник повинен сам явно створити змінну у іменному просторі, або надати доступ до функції. Проблема безпеки стосується в першу чергу серверних процесів, оскільки у них в лексичному оточенні доступні системні дані та дані інших користувачів. Явне створення змінних та функцій у іменному просторі не дасть випадково передати зайві дані у спільний простір.

Спільні змінні не повинні створюватися у лексичному просторі програми, а повинні надаватися через функцію, яка поверне значення за ідентифікатором типу рядок. Це потрібно для того, щоб віддалені процеси не могли впливати на захищене лексичне середовище.

Потрібно зазначити, що при реалізації інструментарію роботи зі спільним іменним простором для систем написаних виключно мовою програмування JavaScript потрібно зробити ізоморфний інтерфейс роботи з простором імен як для клієнта так і для сервера, що дозволить пере використовувати частини коду, які виконують дії з іменним простором.

Архітектура взаємодії в поєднані з ізоморфним інструментарієм роботи з даними дозволяє створювати ізоморфні обгортки над цими даними та операціями, які фактично є об'єктами (сутність що енкапсулює дані і операції над ними) у спільному іменному просторі. Використання спільного іменного простору дозволяє легко рознести методи об'єкту у різні програмні процеси, зберігши одну спільну ізоморфну обгортку для доступу до них.

Програмний процес на сервері є учасником одразу багатьох віртуальних процесів. Для кожної вкладки браузеру створюється свій віртуальний процес.

Архітектура інструментарію доступу до спільного іменного простору повинна мати можливість використання різних локальних сховищ даних. Наприклад оперативна пам'ять та СУБД на сервері, оперативна пам'ять, локальне сховище (localStorage) на клієнті.

Використання персистентних сховищ дозволить з коробки отримати можливість поновлення роботи застосунку зі збереженням даних попередньої сесії.

3.1 Старт віртуального процесу

Віртуальний процес починається після того, як клієнт завантажить та запустить скрипт і створить за допомогою сокетів підключиться до серверів. Клієнту надається унікальний ідентифікатор віртуального процесу, який зберігається у локальному сховищі або кукіс на клієнті, та є складовою ідентифікатора даних на сервері. Після підключення може бути виконано підготовчий скрипт у кожному складовому процесі віртуального процесу для початкового завантаження даних у спільний іменний простір.

Рис. 6 Початок віртуального процесу

3.2 Шлях передачі даних у віртуальному процесі

Ініціатором і головним у віртуальному процесі є процес запущений на клієнті (браузері). Цей процес відповідає за оновлення даних на всіх дочірніх процесах. Тобто коли один з серверів надсилає значення змінної у спільному іменному просторі, клієнтський процес поширює значення цієї змінної серед усіх інших серверних процесів. При спробі доступу до значення спільної змінної виконується читання з локального сховища.

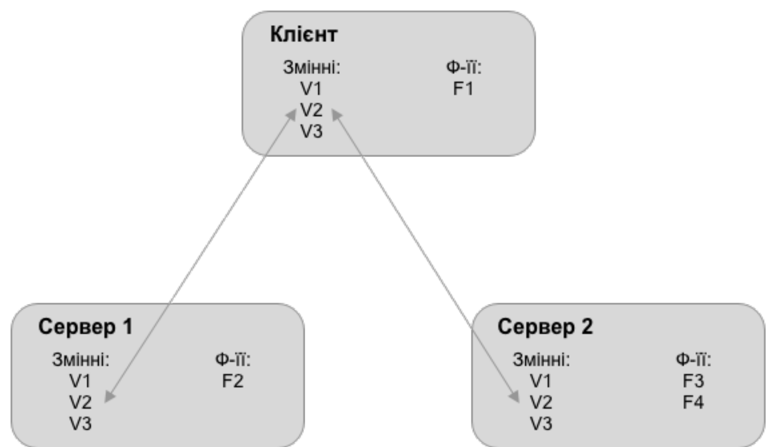

Рис. 7 Схема даних у локальних сховищах

3.3 Робота зі змінною у спільному іменному просторі

Змінна у спільному іменному просторі може бути створена будь-яким програмним процесом. Ім’я змінної – рядок будь-якої розумної довжини. Ім’я

змінною може бути будь-яким, проте рекомендується притримуватись якоїсь логіки іменування подібної до БЕМ або REST. При записі змінної з якогось програмного процесу нове значення змінної транслюється у всі інші складові віртуального процесу.

На зміну значення змінної можна підписатись. Значення змінних транслюються через клієнтський процес, що автоматично вирішує проблему прив'язки значення до конкретного віртуального процесу.

3.4 Робота з функцією у спільному іменному просторі

Функція у спільному іменному просторі – це запит на виконання функції у лексичному оточенні, в якому вона створена. Функція в іменному просторі реєструється при старті віртуального процесу.

При спробі виконання функції, спочатку перевіряється чи не зареєстрована функція локально. У випадку коли функція знаходиться у віддаленому процесі – робиться запит на її виконання з переданими параметрами. На клієнті проводить пошук серверу, який би мав віддалену функцію з вказаним ім'ям.

Потрібно пам'ятати, що аргументи будуть серіалізовані у JSON при передачі, а тому методи об'єктів будуть втрачені. Також аргументи-екземпляри деяких класів мають бути завчасно серіалізовані вручну.

На результат виконання функції можна підписатися. Тобто коли якийсь програмний процес запросить виконання функції, в усі підписані програмні процеси будуть передані результат та аргументи.

3.5 Завершення віртуального процессу

Віртуальний процес рахується завершеним, коли завершує роботу клієнтський програмний процес.

Архітектура надає гнучкість, щодо даних які залишились після завершення віртуального процесу. Вони можуть бути примусово видалені з серверних процесів та автоматично з оперативної пам'яті браузеру.

За умови використання персистентного сховища можливо реалізувати продовження роботи. Оскільки значення спільних змінних є однаковими для всіх програмних процесів збереження можливо реалізувати на будь-якому боці. Завжди потрібно зберігати ідентифікатор незавершеної сесії у браузері, якщо якась частина даних зберігається на сервері.

3.6 Проблема гонки даних

Через асинхронну природу віртуального процесу існує проблема гонки даних. Гонка даних – ситуація, за якої декілька процесів намагаються змінити значення однієї спільної змінної. Потрібно запобігати ситуацій, коли спільні змінні можуть бути записані з декількох серверних процесів. Для захисту від випадкових перезаписів слід давати змінним чіткі імена.

3.7 Проблема неявної зміни даних

Оскільки дані змінюється неявно, тобто спільної змінної немає в лексичному оточенні, інтегроване середовище розробки не може відслідкувати всі місця використання змінної. Також зміни однієї спільної змінної можуть (і мають) відбуватися у різних місцях програмного коду. Отже може виникнути ситуація, коли розробник не знає, що значення змінної ще в якомусь місці змінюється. Це може статися випадково, коли відбувається конфлікт імен, а отже потрібно використовувати систему позначення змінних на кшталт БЕМ, або систему позначення шляху за принципом REST архітектури. Також використання складних семантично-змістовних ідентифікаторів дозволить розробнику швидко виконувати пошук операцій запису змінної засобами інтегрованого середовища розробки.

3.8 Проблема надмірної передачі даних

Передача даних від одного серверу до іншого може створити проблему надмірного навантаження на мережу. Дублювання невеликих обсягів інформації, які описують стан додатку не є якоюсь проблемою на фоні обсягів

передачі даних сучасних веб-застосунків, але можливі випадки коли розробники будуть передавати об'ємні структури даних. Цього слід уникати і використовувати функції для передачі великих обсягів інформації.

РОЗДІЛ 4
ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Прототип бібліотеки було реалізовано мовою програмування JavaScript в інтегрованому середовищі програмування WebStorm.

Як ізоморфну обгортку для сокетів було використано бібліотеку WS.

```
VirtualClient
    constructor(props)
    _validateProps(props)
    _createRemote(route): void
    hasCallback(key)
    getCallback(key)
    setCallback(key, value)
    callCallback()
    _propagate(key, value)
    _propagateSubscription(key)
    get(key, targetRemote)
    set(key, value)
    call()
    subscribeVariable(key, callback)
    type
    server
    isServer
    methods
    remotes
    subscriptions
    store
```

Рис. 8 структура класу VirtualClient

Клас VirtualClient реалізовує логіку верхнього рівня, а саме:

- поля масиву зареєстрованих методів, масиву віддалених процесів, сховища даних, об'єкту WebSocket підключення;
- метод для встановлення значення спільної змінної;

- метод для взяття значення спільної змінної;
- метод для пошуку та визову спільної функції;
- сервісні методи.

Сховище даних є об’єктом з інтерфейсом, який реалізує клас DataStore. Об’єкти сховища даних пишуться окремо під кожну платформу і тип сховища і надають абстрактний засіб роботи з даними. Спільна змінна вважається видаленою або невизначеною, якщо її значення дорівнює null або undefined. Операція видалення – операція присвоювання значення null або undefined.

Клас DataStore окрім конструктора містить наступні методи:

- метод взяття даних;
- метод запису даних;
- метод перевірки наявності змінної.

DataStore
constructor(startData)
getData(key)
setData(key, data)
has(key)
data

Рис. 9 структура класу DataStore

Підключення до віддаленого процесу є об’єктом класу RemoteProcess. Клас DataStore окрім конструктора містить наступні методи:

- метод який інкапсулює відправку запита та отримання побудованих на подієвій моделі у звичайну асинхронну функцію;
- метод для обробки вхідних повідомлень;
- метод прямого читання/запису змінної у віддаленому сховищі;

- метод для перевірки наявності та виклику функції зареєстрованої на віддаленому процесі;

```
RemoteProcess
    constructor(uri, callbacks, connectionSettings)
    request(action, data)
    listen(msg): void
    has(key)
    get(key)
    set(key, value)
    hasMethod(name)
    call()
    uri
    callbacks
    connect
    promises
```

Рис. 10 структура класу RemoteProcess

РОЗДІЛ 5 ПОРІВНЯННЯ ПРИКЛАДІВ КОДУ НАПИСАНОГО НА ОСНОВІ РІЗНИХ ПІДХОДІВ

5.1 Запит на виконання серверної функції

При використанні запропонованої архітектури:

```
const response = await VirtualClient.call('module.getSomeData', data);
```

При використанні REST:

```
async function makeRequest(url = '', data = {}) {
  const response = await fetch(url, {
    method: 'GET',
    mode: 'cors',
    cache: 'no-cache',
    credentials: 'same-origin',
    headers: {
      'Content-Type': 'application/json'
    },
    redirect: 'follow',
    referrerPolicy: 'no-referrer',
    body: JSON.stringify(data)
  });
  return await response.json();
}
```

Після чого можна робити запит:

```
makeRequest('/myapi/module/somedata', data);
```

Проте запит у варіанті REST неможливо перевикористати на самому сервері, або на іншому сервері/мікросервісі.

5.2 Ізоморфна допоміжна функція (хелпер)

При використанні пропонованої архітектури:

```
const myAwesomeHelper = async (params) => {
  const data = await VirtualClient.call('module.getSomeData', params.args);
  const someMath = data.reduce((acc, el) => acc + el.days, 0);

  return someMath;
}
```

При використанні REST:

```
const getDataFromLocal = (args) => {
  return MyDataCollection.find(args).fetch();
}

const getDataFromRemote = (args) => {
  return makeRequest('/remote/module/somedata', args);
}
```

```
const myAwesomeHelper = async (params) => {
  const data = typeof window === "undefined" ?
    await getDataFromLocal(params.args) :
    await getDataFromRemote(params.args);

  const someMath = data.reduce((acc, el) => acc + el.days, 0);

  return someMath;
}
```

5.3 Ізоморфний об’єкт

При використанні REST:

```
const getDataFromLocal = (id) => Posts.findOne({ id });
const getDataFromRemote = (id) => makeRequest('/remote/module/postData', id)
const myAwesomeHelper = id =>
  typeof window === "undefined" ?
    getDataFromLocal(params.args) :
    getDataFromRemote(params.args);

class UserProfile {
  constructor(data) {
    this.data = data;
  }
  doSomeEasyStuff() {
    return this.data.age;
  }
  doMoreComplicatedThings () {
    const res = [];
    this.data.posts.forEach(
      id => res.push(
        myAwesomeHelper( id)
      ));

    return Promise.all(res);
  }
}
```

При використанні пропонованої архітектури:

```
class UserProfile {
  constructor(data) {
    this.data = data;
  }
  doSomeEasyStuff() {
    return this.data.age;
  }
  doMoreComplicatedThings () {
    const res = [];
    this.data.posts.forEach(
      id => res.push(
        VirtualClient.call('module.getPostData', id)
      ));

    return Promise.all(res);
  }
}
```

5.4 Обробка запиту сервером

При використанні пропонованої архітектури:

```
const methods = {
  ...
  'module.action': async (params, session) => {
    if (session.isAuthenticated) {
      return await SecretData.findOne(params);
    }
  }
  ...
}
```

При використанні REST:

```
router.use('/module/', {method: 'POST'}, (res, req) => {
  if (req.isAuthenticated) {
    return await SecretData.findOne(res.body.params);
  }
})
```

5.5 Висновок

Можна легко побачити як швидко збільшується складність коду при ускладненні логіки модуля та при спробі писати модулі, які можна перевикористовувати як на сервері так і на клієнті. Код з використанням запропонованою архітектури є більш лаконічним і виразним ніж варіанти написані під REST API.

Більш того, при спробі звернутися від одного серверу до іншого, (наприклад запросити дані про останню сесію від сервісу авторизації) сервер-джерело має авторизувати свій запит, що непотрібно робити, коли запит передається через вже авторизований клієнт.

При істотному спрощенні створення перевикористовуємих запитів запропонована архітектура жодним чином не ускладнює їх обробку приймаючою стороною.

ВИСНОВКИ

Метою роботи була розробка нової архітектури взаємодії клієнта і сервера/серверів у веб-застосунку.

У процесі виконання були виконані наступні завдання:

- досліджено та проаналізовано сучасний стан веб-розробки;
- досліджено поняття віртуального процесу;
- запропоновано архітектуру взаємодії клієнта та сервера веб-додатку, використовуючи віртуальний процес;
- розроблено прототип бібліотеки (код ключових класів наведено у додатках А, Б, В);
- порівняно складність написання коду при використанні запропонованої та класичної архітектури взаємодії;

Запропонована архітектура дозволяє писати функціонал, який можна без змін використовувати у будь-якій частині веб-застосунку. Вона спрощує сприйняття коду та підвищує швидкість реалізації функціоналу веб-застосунку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. W3C // MDN Web Docs [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en/docs/Glossary/W3C.
2. Крак Ю. В. Віртуальний процес: означення та застосування в створенні системи жестового інтерфейсу / Ю. В. Крак, Ю. В. Коваль, А. Б. Ставровський. // Вісник Київського національного університету імені Тараса Шевченка Серія фізико-математичні науки. – 2015. – №1. – С. 141–144.
3. Реалізація віртуальних змінних / Ю. В. Крак / Портал науково-практичних заходів [Електронний ресурс] – Режим доступу до ресурсу: https://iconfs.net/infocom2017/realizatsiya-virtualnykh-zminnykh.
4. von Neumann J. First Draft of a Report on the EDVAC/ J. von Neumann// Moore School of Electrical Engineering, University of Pennsylvania. – 1945.
5. *Brugali D.* Software Development: Case Studies in Java /D. Brugali, M. Torchiano // Addisson-Wesley. – 2005.
6. Garlan, D. Using Style to Give Meaning to Software Architecture / Abowd G., Allen R., Garlan D.// SIGSOFT '93: Foundations of Software Engineering, Software Engineering Notes, ACM Press. – Los Angeles, USA. – 1993. – Proceedings.– 1993. – Vol. 118. – No. 3. – pp. 9-20.
7. Microsoft Office Online // Microsoft Office Online [Електронний ресурс] – Режим доступу до ресурсу: https://www.microsoft.com/uk-ua/microsoft-365/free-office-online-for-the-web
8. Google Docs // Google Docs [Електронний ресурс] – Режим доступу до ресурсу: https://www.google.com/intl/ru_uA/docs/about/
9. Figma: the collaborative design tool // Figma [Електронний ресурс] – Режим доступу до ресурсу: https://www.figma.com
10. Video editor // Kapwing [Електронний ресурс] – Режим доступу до ресурсу: https://www.kapwing.com

11. OpenStreetMap // OpenStreetMap [Електронний ресурс] – Режим доступу до ресурсу: https://www.openstreetmap.org
12. Maps // Google [Електронний ресурс] – Режим доступу до ресурсу: https://www.google.com.ua/maps/
13. Telegram Web // Telegram [Електронний ресурс] – Режим доступу до ресурсу: https://web.telegram.org/
14. Video meetings // Google Meet [Електронний ресурс] – Режим доступу до ресурсу: https://meet.google.com
15. AutoCad Web App // Autodesk Autocad [Електронний ресурс] – Режим доступу до ресурсу: https://web.autocad.com
16. Sketchup – software for 3D modeling // Sketchup [Електронний ресурс] – Режим доступу до ресурсу: https://www.sketchup.com/en
17. Приват24. Ваш живий інтернет банк // ПриватБанк [Електронний ресурс] – Режим доступу до ресурсу: https://www.privat24.ua/
18. Monobank WEB-кабінет // Monobank [Електронний ресурс] – Режим доступу до ресурсу: https://fop.monobank.ua
19. 1С веб клиент // 1С:Предприятие [Електронний ресурс] – Режим доступу до ресурсу: https://v8.1c.ru/platforma/web-klient/
20. Creatio // Terrasoft [Електронний ресурс] – Режим доступу до ресурсу: https://www.terrasoft.ua/page/ru/creatiocrm
21. Wikipedia – вільна енциклопедія // Wikipedia [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Головна_сторінка
22. YouTube // YouTube [Електронний ресурс] – Режим доступу до ресурсу: https://www.youtube.com
23. Unlimited movies, TV shows, and more. // Netflix [Електронний ресурс] – Режим доступу до ресурсу: https://www.netflix.com/ua-ru/
24. Coursera // Coursera [Електронний ресурс] – Режим доступу до ресурсу: https://www.coursera.org
25. Amazon:Online shopping // Amazon [Електронний ресурс] – Режим доступу до ресурсу: https://www.amazon.com

26. Новобудови мрії // ЛУН [Електронний ресурс] – Режим доступу до ресурсу: https://lun.ua
27. Kongregate: Play free games // Kongregate [Електронний ресурс] – Режим доступу до ресурсу: https://www.kongregate.com
28. The iPhone 12 & 12 Pro Review: New Design and Diminishing Returns // AnandTech [Електронний ресурс] – Режим доступу до ресурсу: https://www.anandtech.com/show/16192/the-iphone-12-review/3
29. 2020 Developer Survey // StackOverflow [Електронний ресурс] – Режим доступу до ресурсу: https://insights.stackoverflow.com/survey/2020
30. Electron Apps // Electron Електронний ресурс] – Режим доступу до ресурсу: https://www.electronjs.org/apps
31. Proof Of Concept In SoftwareDevelopment: 5 Critical Factors To Success / Thanh (Bruce) Pham / Saigon Technology [Електронний ресурс] – Режим доступу до ресурсу: https://saigontechnology.com/blog/proof-of-concept-in-software-development-5-critical-factors-to-success
32. Deception: Degenerate A/B Testing / Doug Arcuri / Hackernoon [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/hackernoon/deception-degenerate-a-b-testing-ecce6635000e
33. Материалы к вебинару "Разработка SPA на React, NodeJS, Express и MongoDB" / krambertech / [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/krambertech/spa-webinar/blob/master/README.md
34. How to GraphQL in Kotlin and Micronaut and create a single endpoint for access to microservices' APIs / Roman Kudtyashov / [Електронний ресурс] – Режим доступу до ресурсу: https://romankudryashov.com/blog/2020/02/how-to-graphql/

ДОДАТОК А

Код класу VirtualClient

```
export default class VirtualClient {
   constructor(props) {
      let { storage, startData, type, methods, remotes, connectionId, server
} = this._validateProps(props);

      this.type = type;
      this.server = server;
      this.isServer = this.type === 'server';
      this.methods = methods;
      this.remotes = [];
      this.subscriptions = {};

      if (stores[type] && stores[type][storage]) {
         this.store = new stores[type][storage](startData);
         this.store.reloadData();
      } else {
         throw new Error(`Wrong storage "${storage}" at process type
"${type}"`);
      }

      remotes.forEach(uri => this._createRemote(uri));
   }
   _validateProps(props) {
      const defProps = {
         storage: 'memory',
         startData: {},
         type: 'client',
         methods: {},
         remotes: [],
         connectionId: undefined
      }

      return { ...defProps, ...props };
   }
   _createRemote(route) {
      this.remotes.push(
         new Remote(
            route,
            {
               has: this.hasCallback,
               get: this.getCallback,
               set: this.setCallback,
               call: this.callCallback
            },
            {
               reconnects: 25,
               timeout: 20
            }
      ));
   }
   hasCallback (key) {
      return this.store.has(key);
   }
   getCallback (key) {
      return this.store.get(key);
   }
   setCallback (key, value) {
      return this.store.set(key, value);
   }
   callCallback () {
```

```
        return this.call(...arguments);
    }
    async _propagate (key, value) {
        this.remotes.forEach(remote => remote.set(key, value));
    }
    async _propagateSubscription (key) {
        this.remotes.forEach(remote => remote.su)
    }
    async get(key, targetRemote) {
        if (await this.store.has(key)) {
            return await this.store.get(key);
        } else {
            let res = undefined;
            for(let i = 0; i < this.remotes.length; i++) {
                const remote = this.remotes[i];
                if (await remote.has(key)) {
                    res = await remote.get(key);
                    break;
                }
            }
            return res;
        }
    }
    async set(key, value) {
        if (this.isServer) {
            this.store.set(this.connection, key, value);
        } else {
            this.store.set(key, value);
        }
        this._propagate(key, value);
    }
    async call() {
        const name = arguments[0];
        if (this.methods[name]) {
            return this.methods[name](...[ ...arguments ].slice(1));
        } else {
            let remote;
            for(let i = 0; i < this.remotes.length; i++) {
                if (await  this.remotes[i].hasMethod(name)) {
                    remote =  this.remotes[i];
                    break;
                }
            }
            if (remote) {
                return await remote.call(...arguments);
            } else {
                throw new Error(`No method ${name} avaliable`);
            }
        }
    }
    async subscribeVariable (key, callback) {
        if (!this.subscriptions[key]) this.subscriptions[key] = [];
        this.subscriptions[key].push(callback);
    }
}
```

ДОДАТОК Б

Код класу RemoteProcess

```
class RemoteProcess {
   constructor(uri, callbacks, connectionSettings) {
      this.uri = uri;
      this.callbacks = callbacks;
      this.connect = new WebSocket(uri);
      this.promises = [];

      this.connect.on('open', () => {

      });
      this.connect.on('message', this.listen);
   }
   async request (action, data) {
      this.connect.send(JSON.stringify({
         actionId: `${action}-req`,
         data
      }));
      return new Promise((resolve) => {
         this.promises.push({
            actionId: `${action}-res`,
            action: resolve
         });
      });
   }
   listen(msg) {
      try {
         const obj = JSON.parse(msg);
         const { actionId, data } = obj;
         const [ action, actionDirection ] = actionId.split('-');
         if (actionDirection === 'res') {
            let newPromises = [];
            let found = false;

            for(let i = 0; i < this.promises.length; i++) {
               if(!found && this.promises[i].actionId === actionId) {
                  this.promises[i].action(data);
               } else {
                  newPromises.push(this.promises[i]);
               }
            }

            this.promises = newPromises;
         } else {
            this.callbacks[action](...data.args);
         }
      } catch (e) {
         console.log(e);
      }
   }

   async has(key) {
      return await this.request('has', { key });
   }
   async get(key) {
      return await this.request('get', { key });
   }
   async set(key, value) {
      return await this.request('set', { key, value });
   }
   async hasMethod(name) {
```

```
      return await this.request('hasMethod', { name });
   }
   async call() {
      return await this.request('call', { name: arguments[0], args: [
...arguments].slice(1) });
   }
}
```

ДОДАТОК В

Код класів MemoryStoreClient та MemoryStoreServer

```
class MemoryStoreClient {
   constructor(startData = {}) {
      this.data = startData;
   }
   async reloadData () {
      return true;
   }
   async getData(key) {
      return this.data[key]
   }
   async setData(key, data) {
      this.data[key] = data;
      return true;
   }
   async has(key) {
      return this.data.hasOwnProperty(key);
   }
}


class MemoryStoreServer {
   constructor(startData = {}) {
      this.data = startData;
   }
   async reloadData () {
      return true;
   }
   async get(connection, key) {
      if(this.data[connection]){
         return this.data[connection][key];
      }
      throw new Error('Client is not connected');
   }
   async set(connection, key, data) {
      if(!this.data[connection]) this.data[connection] = {};
      this.data[connection][key] = data;
      return true;
   }
   async has(connection, key) {
      if (this.data[connection]) {
         return this.data[connection].hasOwnProperty(key);
      }
      return false;
   }
}
```