Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)
(рівень вищої освіти)

Система верифікації контрольних робіт при дистанційному навчанні
(тема)

Виконав: студент 2 курсу, групи СКСм-21-1

Блудов Д.О.
(прізвище, ініціали)

Спеціальність *123 – Комп'ютерна інженерія*
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма *Спеціалізовані комп'ютерні системи*
(повна назва освітньої програми)

Керівник проф., д.т.н., Чумаченко С.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Чумаченко С.В.
(підпис) (прізвище,ініціали)

2022 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління

Кафедра Автоматизації проектування обчислювальної техніки

Рівень вищої освіти другий (магістерський)

Спеціальність 123_ Комп'ютерна інженерія
(шифр і назва)

Тип програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Спеціалізовані комп'ютерні системи
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«_19_» грудня 20 22 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Блудову Дмитру Олеговичу
(прізвище, ім'я, по батькові)

1. Тема роботи Система верифікації контрольних робіт при дистанційному навчанні

затверджена наказом по університету від 14 листопада 2022 р. № 1478

2. Термін подання студентом роботи до екзаменаційної комісії 12 грудня 2022 р.

3. Вихідні дані до роботи
Мікросервис
NodeJs
WebSocket
Клієнт-серверна взаємодія
Віддалений сервер

4. Перелік питань, що потрібно опрацювати в роботі
Аналіз предметної області та постановка задачі
Аналіз літератури
Розробка алгоритмів
Розробка веб-додатку
Тестування розробленої системи

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) 20 слайдів

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

7. Дата видачі завдання 01.09.2022

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Видача теми роботи, узгодження і затвердження	01.09.2022 – 04.09.2022	
2	Аналіз літератури	10.09.2022 – 15.10.2022	
3	Розробка алгоритмів	16.10.2022 – 27.10.2022	
4	Програмування апаратної частини	03.10.2022 – 05.10.2022	
5	Розробка веб-додатку	15.11.2022 – 01.12.2022	
6	Проведення тестування отриманої моделі	02.12.2022 – 05.12.2022	
7	Оформлення пояснювальної записки	06.12.2022 - 11.12.2022	
8	Перевірка виконаного проекту керівником	19.12.2022	
9	Захист проекту	22.12.2022	

Студент _____
(підпис)

Керівник роботи _____ проф., д.т.н., Чумаченко С.В.
(підпис) (посада, прізвищ

РЕФЕРАТ

Пояснювальна записка містить 62 сторінок, 11 рисунків, 16 джерел за переліком посилань.

NODEJs, TYPESCRIPT, TYPEORM, DATABASE, СЕРВЕР, AUTHENTICATION, MICRO-SERVICE ARCHITECTURE.

Метою роботи є підвищення якості збирання, збереження й обробки інформації для обліку успішності студентів шляхом розробки системи верифікації контрольних робіт на основі мікросервісної архітектури та програмного застосунку в рамках реалізації хмарних сервісів кіберуніверситету.

Було проведено аналіз та розробка системи верифікації контрольних робіт. Розглянуто сучасні технології розробки та архітектури проекту.

ABSTRACT

The explanatory note contains 62 pages, 11 figures, 16 sources according to the list of references.

NODEJS, TYPESCRIPT, TYPEORM, DATABASE, SERVER, AUTHENTICATION, MICRO-SERVICE ARCHITECTURE.

The purpose of the work is to improve the quality of information collection, storage and processing for students success accounting by developing a system for verification of control works based on microservice architecture and a software application within the framework of the implementation of cloud services of the cyber university.

An analysis of data transmission models for the smart home system was conducted. Their advantages and disadvantages. The concept of a smart house and its main uses are considered.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних;

Нода – NodeJs framework;

IoT – Інтернет речей (англ. Internet of Things);

ООП – одна з парадигм програмування, яка розглядає програму як множину «об'єктів», що взаємодіють між собою;

СКБД – система керування базами даних;

API (англ. Application Programming Interface) – це набір чітко визначених методів для взаємодії різних компонентів;

Backend – стадія веб-розробки, яка відповідальна за роботу з боку сервера (взаємодія з даними сервера, базою даних);

Frontend – стадія веб-розробки, яка відповідальна за роботу з боку користувача (стиль, інтерфейс, функціонал тощо);

ORM – технологія, яка зв'язує бази даних з концепціями ООП мов програмування, створюючи «віртуальну об'єктну базу даних».

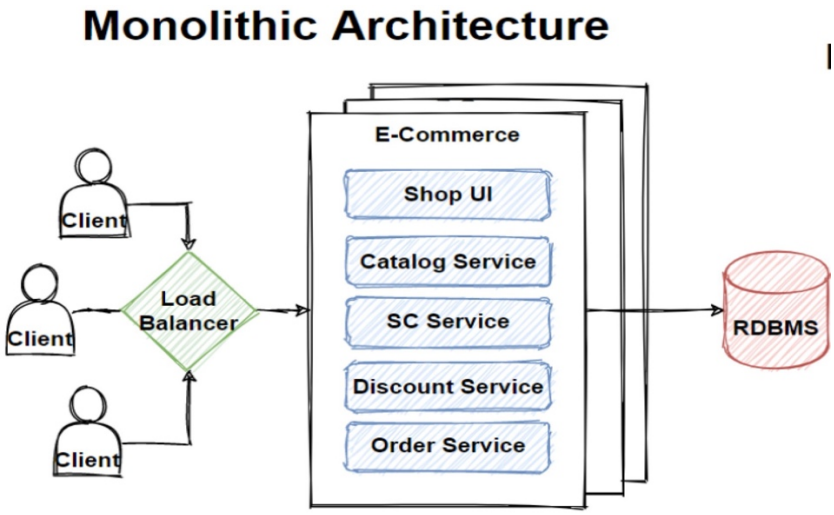
ВСТУП

У теперішній час дистанційне навчання є досить актуальним, бо людина може навчатися у будь-якому місці вдома, в кафе, або навіть на вулиці у парку. Дистанційна освіта аткивно поширюється впроваджується у світі з 2010, коли провідні університети розпочали надавати онлайн доступ до своїх курсів. Технології дистанційного навчання особливо стали у нагоді підчас пандемії, коли ніхто не може бути у безпеці.

Дистанційне навчання повинне супроводжуватись контролем знань студентів, тому розробка дистанційної верифікації робіт студента є затребуваною.

У сучасному світі існує багато різноманітних технологій в розробці додатків, різні серверні мови програмування: Java, C#, C, Ruby, JS. У фронтенді існує багато фреймворків для JS: Angular, Vue, React. Хоч вони і відрізняються між собой швидкістю, рівнем складності але кожен з них має свої переваги над іншими як і не достатки.

Важливою є також архітектура розробки, найпопулярніша це – моноліт. Моноліт розглядається як архітектурне рішення, де усі компоненти та модулі тісно пов’язані між собою, та залежать один від одного, що докладніше проілюстровано на рисунку.



Архітектура моноліту

Переваги: простота розгортання. Моноліти дуже швидко і відносно просто розгортати, завдяки тому що у моноліта, зазвичай, єдина точка входу.

Розробка. Розробка моноліту зазвичай дуже швидка, бо усі компоненти та модулі знаходяться в одній кодовій базі та завжди під рукою.

Відладка. Відладка моноліту дуже спрощена за рахунок того, що усе поряд, і є можливість відстежити усю ланку.

Недоліки: маштабування. Моноліти маштабуються лише повністю, тобто якщо навантаження зростає лише на один модуль, неможна маштабувати лише цей модуль, слід маштабувати весь моноліт.

Надійність: недостатня гнучкість. Моноліти негнучкі, тобто зміна одного модуля у моноліті майже завжди впливатиме на інший модуль

Але в даному проекті пропонується використовувати мікросервісну архітектуру.

Метою роботи є підвищення якості збирання, збереження й обробки інформації для обліку успішності шляхом розробки системи верифікації контрольних робіт на основі мікросервісної архітектури та програмного застосунку в рамках реалізації хмарних сервісів кіберуніверситету.

Для досягнення мети виконується огляд інноваційних сервісів розумного кіберуніверситету, аналіз сучасних технологій, розробляється архітектура проекту, програмний додаток для системи верифікації контрольних робіт, виконується мануальне тестування.

1 ІННОВАЦІЙНІ СЕРВІСИ РОЗУМНОГО КІБЕР-УНІВЕРСИТЕТУ

Інноваційні сервіси, що формують розумний кібер-університет як структурний прототип глобального науково-освітнього віртуального кібер-простору Global Smart Cyber University, на рис. 1.1.

1. Хмарний кібер-сервіс захищеного електронного документообігу для цифрового моніторингу та інтелектуального кіберуправління науково-освітніми процесами (створення, реалізація та утилізація документа), у форматі замкнутого циклу: «факт – вимір – оцінка – дія», що повністю виключає папір носії шляхом використання Cloud-Mobile Service Computing.

Рисунок 1.1 – Інноваційні сервіси розумного кібер-університету [1]

Хмарний кібер-сервіс мобільного голосування e-voting для моніторингу громадської думки; реалізації студентських опитувань; ухвалення рішень на оперативних нарадах, засіданнях вченої ради, конференціях трудового колективу; проведення виборів експертів, студентського сенату, керівника та навчально-педагогічного складу при заміщенні вакантних посад.

Хмарний кібер-сервіс управління персоналом на основі online моніторингу, вимірювання, рейтингування та накопичення цифрових метрик компетенцій для оцінювання діяльності: студентів та всіх категорій співробітників з метою вироблення прозорих регуляторних моральних та матеріальних стимулів.

Хмарний кібер-сервіс управління структурним підрозділом на основі online моніторингу, вимірювання та накопичення цифрових метрик компетенцій кафедри, пов'язаних з науково-освітнім процесом, для вироблення регуляторних керуючих впливів та генерування пакету документів, необхідних для життєдіяльності.

Хмарний кібер-сервіс оцінки якості освітніх процесів та компонентів, online тестування знань та умінь, що виключає нелегітимні стосунки між викладачем та студентом під час складання іспитів та заліків.

Хмарний кібер-сервіс управління науковими процесами на основі цифрового оцінювання діяльності вчених, підрозділів, наукових результатів, проектів та пропозицій за метриками, розробленими експертами, з метою прозорого та легітимного розподілу фінансових, кадрових та тимчасових ресурсів між підрозділами та співробітниками.

Хмарний кібер-сервіс надання освітніх послуг у вигляді MOOC online и onsite курсів, а також управління освітнім процесом на основі прозорого розподілу фінансових та часових ресурсів між підрозділами та співробітниками у суворій відповідності до метричного оцінювання вкладу кожного суб'єкта в актив та імідж університету.

Хмарний кібер-сервіс моніторингу та управління науково-освітнього процесу студента в реальному масштабі часу, генерування та зберігання електронних документів для його супроводу в часі та просторі через створення персонального віртуального кабінету, пов'язаного з мобільним пристроєм та e-mail .

Хмарний кібер-сервіс вимірювання та супроводу бакалаврських, магістерських та дисертаційних робіт, а також конкурсних проектів на основі

інтеграції міжнародних метрик оцінювання наукової та практичної значущості результатів проведених досліджень із внутрішніми критеріями якості, розробленими експертами.

Хмарний кібер-сервіс ліцензування та акредитації спеціальностей на основі виміру науково-освітньої діяльності кафедр та наступного генерування пакета документів, необхідного для зовнішнього оцінювання якості навчальних процесів.

Хмарний кібер-сервіс електронного 24/7 доступу та моніторингу присутності співробітників та студентів в інфраструктурних аудиторіях університету на основі використання мобільних пристроїв, а також електронний банкінг для оплати освітніх послуг та використання корпоративних кафедральних карток для придбання товарів та послуг у межах зароблених кафедрою засобів.

Хмарний кібер-сервіс захисту інформаційно-фізичного простору університету та санкціонування електронного доступу до всіх кіберфізичних компонентів та процесів, пов'язаних з життєдіяльністю вишу.

2 АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ РОЗРОБКИ

2.1 NodeJs

NodeJs – це JavaScript–оточення побудоване на JavaScript–рушієві Chrome-V8.

Як асинхронне подієве JavaScript–оточення, Node.js спроектований для побудови масштабованих мережевих додатків. У нижче наведений приклад "hello world", який може одночасно обробляти багато з’єднань. Для кожного з’єднання викликається функція зворотнього виклику, проте коли з’єднань немає Node.js засинає.

Node.js створений під впливом таких систем, як Event Machine в Ruby або Twisted в Python. Node.js використовує подієву модель значно ширше, він приймає за основу оточення, замість того, щоб використовувати його як бібліотеку. В інших системах завжди стається блокування виклику, щоб запустити цикл подій (event loop).

Зазвичай поведінка визначається через функції зворотнього виклику на початку скрипта і в кінці запускає сервер через блокуючий виклик, як от EventMachine::run(). В Node.js немає нічого подібного на виклик початку циклу подій. Node.js просто входить в подієвий цикл після запуску скрипта на виконання. Node.js виходить з подієвого циклу тоді, коли не залишається зареєстрованих функцій зворотнього виклику. Така поведінка схожа на поведінку браузерного JavaScript: подієвий цикл прихований від користувача.

HTTP є об'єктом першого роду в Node.js, розробленим з потоковістю та малою затримкою. Це робить Node.js хорошою основою для веб–бібліотеки або фреймворку.

Те, що Node.js спроектований без багатопоточності, не означає, що він не надає можливості використовувати кілька ядер у середовищі. Можливо

також створювати дочірні процеси, якими легко керувати з допомогою API child.process.fork(). Модуль cluster побудований на цьому інтерфейсі і дозволяє ділитись сокетами між процесами та розподіляти навантаження між ядрами. Node.js був створений Райаном Далом (Ryan Dahl) в 2009 році.

Сам node.js включає в себе кілька основних складових:

рушій V8;

бібліотеки libuv, яка відповідає за центральну частину node – цикл подій (event loop), який здійснює взаємодію з ОС, а так само за асинхронне введення/виведення (I/O);

з набору різних JS бібліотек і безпосередньо самої мови JS.

Перейдемо до його переваг та недоліків.

Переваги:

легкість і швидкість написання;

легковаговик;

відносна простота (у порівнянні з java);

npm (node package manager (величезна кількість бібліотек які можуть бути встановлені в один рядок);

кожна бібліотека легко потрапляє до дерева залежностей;

постійний розвиток (зараз активно розвивається TypeScript, який привносить в JS типізацію, декоратори і використовується, наприклад, для Angular).

Недоліки:

гнучкість і швидкий розвиток породжує також і недоліки, оскільки слід постійно стежити за оновленнями, деякі речі виходять недостатньо протестованими;

був випадок, коли розробник видалив свою бібліотеку з NPM і сукупність застосунків, що використовують її, перестали працювати.

На рис. 2.1 наведено приклад управління потоками в nodeJs.

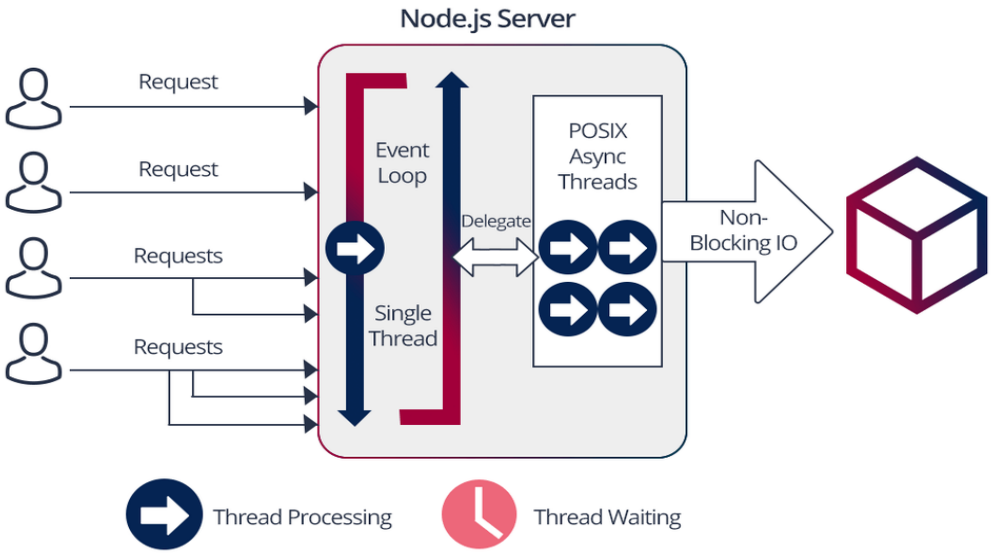


Рисунок 2.1 – Управління потоками в node.js[4]

2.2 Typescript

TypeScript — мова програмування, представлена Microsoft восени 2012 року, що позиціонується як засіб розробки вебзастосунків, яка розширює можливості JavaScript.

Розробником мови TypeScript є Андрес Гейлсберг, який створив раніше C#, Turbo Pascal і Delphi.

Код експериментального компілятора, котрий транслює код TypeScript в представлення JavaScript, поширюється під ліцензією Apache, розробка ведеться в публічному репозиторії через сервіс CodePlex. Специфікації мови відкриті і опубліковані в рамках угоди Open Web Specification Agreement.

TypeScript є зворотньо сумісним з JavaScript. Фактично, після компіляції програму на TypeScript можна виконувати в будь-якому сучасному браузері або використовувати спільно з серверною платформою NodeJs.

Переваги над JavaScript:

1) можливість явного визначення типів (статична типізація);

2) підтримка використання повноцінних класів (як в традиційних об'єктно-орієнтованих мовах);

3) підтримка підключення модулів.

За задумом ці нововведення мають підвищити швидкість розробки, прочитність, рефакторинг і повторне використання коду, здійснювати пошук помилок на етапі розробки та компіляції, а також швидкодію програм.

Планується, що в силу повної зворотної сумісності адаптація наявних застосунків на нову мову програмування може відбуватися поетапно, шляхом поступового визначення типів. Підтримка динамічної типізаці зберігається – компілятор TypeScript успішно обробить і не модифікований код на JavaScript.

Основний принцип мови – будь-який код на JavaScript сумісний з TypeScript, тобто в програмах на TypeScript можна використовувати стандартні JavaScript-бібліотеки і раніше створені напрацювання. Більш того, можна залишити наявні JavaScript-проекти в незмінному вигляді, а дані про типізації розмістити у вигляді анотацій, які можна помістити в окремі файли, які не заважатимуть розробці і прямому використанню проекту (наприклад, подібний підхід зручний при розробці JavaScript-бібліотек).

2.3 Бази даних (Postgresql)

База даних – це засіб збирання та впорядкування інформації. Вони можуть зберігати відомості про людей, продукти, замовлення або будь-що інше.

Багато баз даних розпочинаються зі списку в текстовому редакторі або електронній таблиці. Оскільки список збільшується, то в даних з'являються неузгоджені та невідповідні значення.

У формі списку дані буде важко розібратися, а також є обмежені способи пошуку або витягування підмножини даних для перевірки. Після того, як ці проблеми почнуть з'являтися, рекомендується перенести дані до бази даних, створеної системою керування базами даних (СКБД), наприклад Access.

Комп’ютеризована база даних – це контейнер об’єктів. Одна база даних може містити кілька таблиць.

Таблиця бази даних схожа на електронну – в обох дані зберігаються в рядках і стовпцях. Тому зазвичай досить легко імпортувати електронну таблицю в таблицю бази даних. Головна відмінність між тим, як дані зберігаються в електронній таблиці та базі даних, – це спосіб, яким упорядковуються дані.

Щоб база даних була максимально гнучка, дані має бути впорядковано в таблиці, щоб позбутися зайвих елементів. Наприклад, зберігаючи відомості про працівників, слід настроїти відповідну таблицю, у яку дані кожного працівника потрібно ввести лише один раз. Дані про продукти зберігатимуться у власній таблиці, а дані про філіали – в іншій. Ця процедура називається нормалізацією.

PostgreSQL – об’єктно-реляційна система керування базами даних (СКБД). Є альтернативою як комерційним СКБД, так і СКБД з відкритим кодом (MySQL, Firebird, SQLite).

Порівняно з іншими проєктами з відкритим кодом, такими як Apache, FreeBSD або MySQL, PostgreSQL не контролюється якоюсь однією компанією, її розробка можлива завдяки співпраці багатьох людей та компаній, які хочуть використовувати цю СКБД та впроваджувати у неї найновіші досягнення.

Сервер PostgreSQL написаний на мові C. Зазвичай розповсюджується у вигляді набору текстових файлів із сирцевим кодом. Для інсталяції необхідно відкомпілювати файли на комп’ютері і скопіювати в деякий каталог. Весь процес детально описаний в документації.

PostgreSQL підтримує великий набір вбудованих типів даних:

1. Числові типи.
2. Цілі.
3. З фіксованою крапкою.
4. З нефіксованою крапкою.

5. Грошовий тип.
6. Символьні типи довільної довжини.
7. Двійкові типи (включаючи BLOB).
8. Типи «дата/час».
9. Булевий тип.
10. Перерахування.
11. Геометричні примітиви.
12. Мережеві типи.
13. IP і IPv6-адреси.
14. CIDR-формат.
15. MAC-адреса.
16. UUID-ідентификатор.
17. XML-дані.
18. JSON-дані.
19. Масиви.
20. OID-типи.
21. Псевдотипи.

2.4 TypeORM

TypeORM – це ORMFramework, він може працювати на платформах NodeJS, browser, Cordova, PhoneGap, Ionic, React Native, Expo та Electron, а також може використовуватися з TypeScript та JavaScript (ES5, ES6, ES7).

Мета – підтримувати новітні функції JavaScript і надавати додаткові функції, які допоможуть розробити будь-яку програму, яка використовує базу даних, чи то невелику програму з кількома таблицями або кількома базами даних.

На відміну від інших існуючих платформ ORM JavaScript, TypeORM підтримує режими Active Record і Data Mapper, що означає, що ви пишете високоякісні, слабо пов'язані, масштабовані і легко обслуговуються програми

найбільш ефективним способом.

TypeORM відноситься до реалізації багатьох інших чудових ORM, таких як Hibernate, Doctrine за участю Entity Framework.

Приклад вигляду моделі у TypeORM

```
import {Entity, PrimaryGeneratedColumn, Column} from "typeorm";
@Entity()
export class User {

    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    firstName: string;

    @Column()
    lastName: string;

    @Column()
    age: number;
}
```

2.5 NPM/Yarn

NPM (Node Package Manager) – це менеджер пакетів для NodeJs. NPM дозволяє завантажувати і встановлювати пакети, модулі для NodeJs.

Розроблений з метою пришвидшити і автоматизувати завантаення і встановлення пакетів, модулів.

NPM написаний на мові програмування JavaScript. **Npm** – найбільший у

світі реєстр програмного забезпечення. Реєстр містить понад 800 000 пакетів кодів.

Розробники з відкритим кодом використовують **npm** для обміну програмним забезпеченням. Багато організацій також використовують **npm** для управління приватним девелопментом.

Назва npm (Node Package Manager) походить від того, що npm вперше було створено як менеджер програмних пакетів для Node.js. Усі пакети npm записуються у файлах, що називаються package.json. Вміст package.json повинен бути записаний у форматі JSON. У файлі визначення повинні бути принаймні два поля: ім’я та версія програмного забезпечення. Приклад файлу:

```
{
        "name": "gateway",
        "version": "0.1.0",
        "description": "API Gateway",
        "main": "service.js",
        "scripts": {
          "start:ts": "tsc -w --preserveWatchOutput",
          "start": "tsc --preserveWatchOutput && npm-run-all --parallel start:ts
      start:nodemon",
          "start:nodemon": "nodemon -x \"node --es-module-specifier-resolution=node
      ./dist/index.js\"",
          "build": "tsc"
        },
        "repository": {
          "type": "git",
          "url": "git+ssh://git@github.com/folder/repo.git"
        },
        "keywords": [
          "rewards"
        ],
        "license": "ISC",
        "bugs": {
          "url": "https://github.com/folder/repo/issues"
        },
        "homepage": "https://github.com/folder/repo#readme",
        "dependencies": {

          "@realisnetwork/agents": "^0.2.1365",
          "@realisnetwork/auth-wrapper": "^0.2.1",
          "@realisnetwork/configurator": "^0.1.0",
          "@realisnetwork/errors-registry": "^0.2.65",
          "@realisnetwork/form-url-encoded": "^0.0.2",
```

```
  "@realisnetwork/logger": "^0.1.0",
  "@realisnetwork/nats-streaming": "^0.3.0",
  "@realisnetwork/nats-streaming-message-reply": "^0.3.0",
  "@realisnetwork/router": "^0.3.4",
  "@realisnetwork/topics": "0.0.161",
  "@realisnetwork/types": "^0.1.406",
  "@sinclair/typebox": "0.11.0",
  "@sweet-monads/either": "^2.1.4",
  "@types/ajv-merge-patch": "^4.1.1",
  "@types/bluebird": "^3.5.33",
  "@types/redis": "^2.8.32",
  "@types/ws": "^7.4.4",
  "ajv": "^8.11.0",
  "ajv-errors": "^3.0.0",
  "ajv-formats": "^2.1.1",
  "ajv-keywords": "^5.1.0",
  "ajv-merge-patch": "^4.1.0",
  "axios": "^0.21.1",
  "bignumber.js": "^9.0.1",
  "bluebird": "^3.7.2",
  "dotenv": "^8.2.0",
  "express-session": "^1.17.2",
  "http-status-codes": "^2.1.4",
  "nodemon": "^2.0.6",
  "redis": "^3.1.2",
  "reflect-metadata": "^0.1.13",
  "serialize-error": "^8.0.1",
  "trace-unhandled": "^2.0.1",
  "tsyringe": "^4.4.0",
  "uWebSockets.js": "uNetworking/uWebSockets.js#v20.3.0",
  "uuid": "^8.3.2",
  "uuidv4": "^6.2.11",
  "uws": "^200.0.0"
},
"devDependencies": {
  "@types/dotenv": "^8.2.0",
  "@types/express-session": "^1.17.4",
  "@types/node": "^16.11.4",
  "@types/uuid": "^8.3.0",
  "@typescript-eslint/eslint-plugin": "^4.17.0",
  "@typescript-eslint/parser": "^4.17.0",
  "eslint": "^7.15.0",
  "eslint-config-airbnb-typescript": "^12.3.1",
  "eslint-config-prettier": "^6.10.0",
  "eslint-plugin-import": "^2.20.2",
  "nodemon": "^2.0.6",
  "npm-run-all": "^4.1.5",
  "ts-node": "^8.10.1",
```

```
    "typescript": "^4.1.3"
  }
}
```

Yarn – це клієнт npm (* менеджер пакунків для Node. Тут і надалі примітка перекладача) з відкритим первинним кодом, який було розроблено командою Facebook та у багатьох аспектах переважає стандартний клієнт npm. У даному посібнику я зосереджуся на розгляданні шести ключових характеристик, які дозволяють Yarn бути чудовим інструментом:

1. Швидкість.
2. Надійність установок.
3. Можливість перевірки ліцензій.
4. Сумісність із npm та Bower.
5. Наявність декількох реєстрів.
6. Можливість використання емодзі.

Одним із досягнень Yarn є його швидкість роботи в порівнянні зі стандартним npm. Але наскільки він швидкий? Недавно проведене еталонне тестування показало, що Yarn був у два-три рази швидше npm. У цьому тесті замірявся час установлення React, Angular 2 та Ember. Це вельми надійна перевірка менеджера пакетів, оскільки кожний із цих фреймворків потребує багато залежностей та містить велику долю залежностей реальних веб-застосунків.

Додамо ще одне значення та самостійно протестуємо швидкість шляхом установлення модуля create-react-app за допомогою yarn та npm. Нижче наведено результат установки за допомогою yarn:

```
$ yarn global add create-react-app --prefix /usr/local
yarn global v0.27.5
warning package.json: No license field
warning No license field
[1/4] Resolving packages...
[2/4] Fetching packages...
```

```
[3/4] Linking dependencies...
[4/4] Building fresh packages...
success Installed "create-react-app@1.4.0" with binaries:
      - create-react-app
warning No license field
Done in 2.59s.
```

Результат установлення за допомогою npm:

```
$ npm install -g create-react-app
/usr/local/bin/create-react-app ->
/usr/local/lib/node_modules/create-react-app/index.js
+ create-react-app@1.4.0
added 80 packages in 9.422s
```

Цей результат безумовно узгоджується з іншими повідомленнями про значну перевагу yarn у швидкості. Yarn установив модулі за .59 секунд, тоді як npm знадобилося 9.422 секунд. Yarn був швидше в 3.63 рази.

Yarn також може похвалитися більш надійними установками, ніж npm. Коли встановлення вважається надійним? Якщо при наступних встановленнях відбувається збій чи виходить інший результат, то установка є ненадійною. Це може відбуватися по двом причинам:

1. Часові проблеми комп'ютерної мережі можуть послужити причиною збою при отриманні пакетів.
2. Результатом виходу нових версій пакетів можуть бути зміни, через які пакети стають несумісними з іншими пакетами та порушується робота застосунку.

Yarn упорується з обома проблемами. Yarn використовує глобальний офлайн кеш для зберігання пакетів, котрі ви одного разу встановили, тому при установках нових версій використовується гешована версія та досягається усталеність до періодичних збоїв комп'ютерної мережі. Для деяких проектів необхідне дотримання певних ліцензійних вимог чи просто створення звіту для власних чи зовнішніх цілей. Це легко здійснюється за допомогою команди Yarn yarn licenses ls. У результаті створюється компактний звіт, що містить офіційне ім'я пакета, його URL-адресу та ліцензію.

2.6 Docker

Кількість сервісів у програмній системі може бути великим, та кожен сервіс може мати відмінну технологію від інших. З таким різноманіттям технологій існує проблема управління середовищами. Щоб спростити процес створення різних середовищ використовують технологію контейнерів. Одним з найвідоміших інструментів з підтримкою контейнеризації є Docker.

Докер – популярний загальнодоступний проект на основі контейнерів Linux. Докер написаний на мові GO і розвинений з Dotcloud (компанія PaaS). Докер – контейнерний двигун, який використовує функції ядра Linux, такі як простір імен та контрольні групи, щоб створити контейнери зверх операційної системи і автоматизувати розгортання на контейнері.

Найкраща функція Докера – можливість співпраці. Образи докера можуть бути вигружені в сховище і можуть бути загружені на будь-який інший хост, щоб управляти контейнерами образу. Крім того, сховище Докера має тисячі образів створених іншими користувачами, і ви можете загрузити ті образи на свій хост, залежно від вимог до конкретної програмної системи (рис. 2.2).

Контейнеризацію завжди порівнювали як підхід до розгортання за віртуалізацією. Віртуальні машині використовувались як спосіб оптимізації, бо на одному комп’ютері можливо запустити багато віртуальних машин. Віртуальні машини створюють середовище для кожної програми. Але віртуальні машини складно масштабувати, оскільки віртуалізація потребує багато процесорного часу та пам’яті.

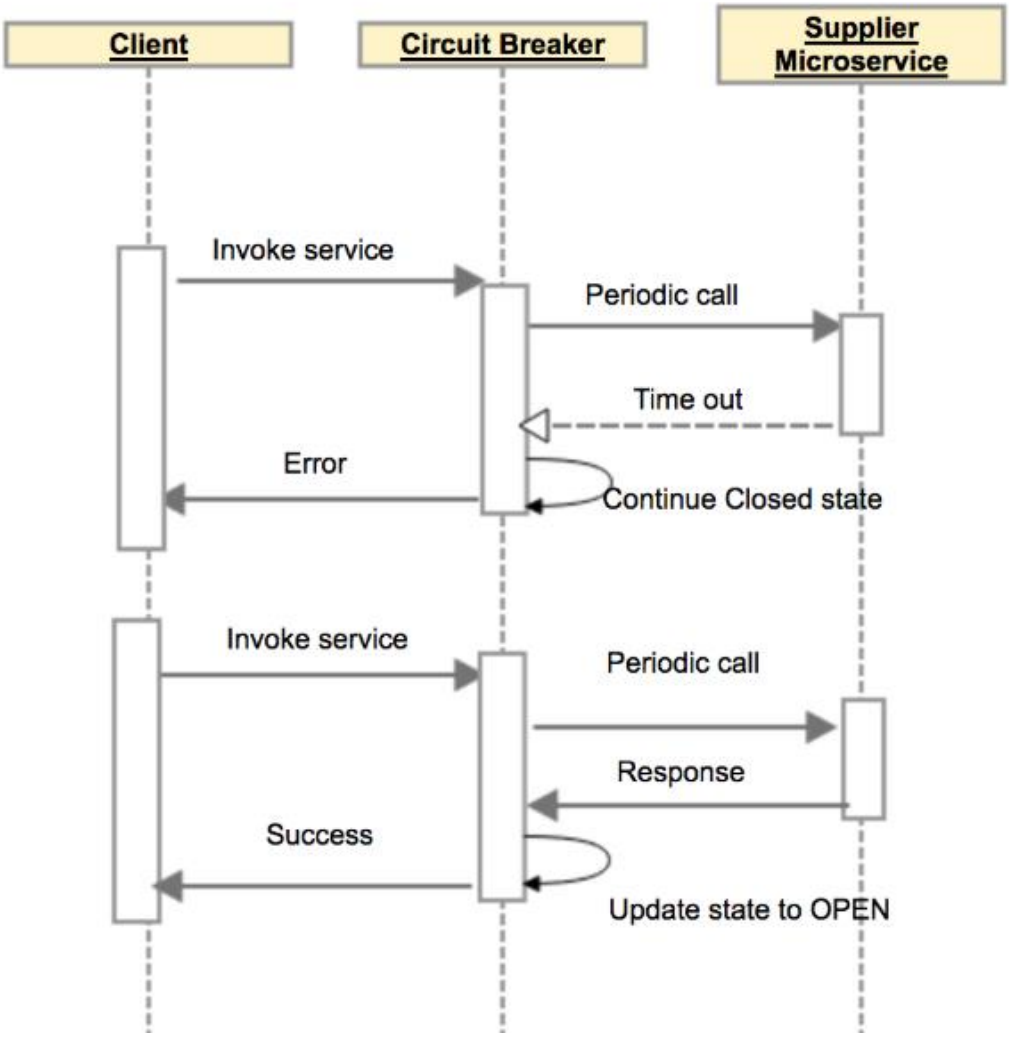


Рисунок 2.2 – Приклад роботи автоматичного вимикача [11]

Мікросервіси не є великими та громіздкими, зазвичай це компактні програмні рішення, яке потребує своє окреме середовище. Тому створення віртуальної машини для такої задачі не є оптимальним рішенням. Docker вимагає набагато менше системних ресурсів. За допомогою Docker можна розгорнути багато сервісів на одному сервері.

Основні переваги використання Docker:

час запуску;

час розгортання;

легке масштабування;

оптимізація;

підтримка великої кількості технологій.

3 МІКРОСЕРВІСНА АРХІТЕКТУРА

3.1 Порівняння мікросервісу та моноліту

MSA – принципова організація розподіленої системи на основі мікросервісів та їх взаємодії один з одним і з середовищем по мережі, а також принципів, що спрямовують проектування архітектури, її створення та еволюцію.

Зрозуміти суть мікросервісу найпростіше в порівнянні, або навіть протиставленні його великому додатку – моноліту (рис. 3.1).

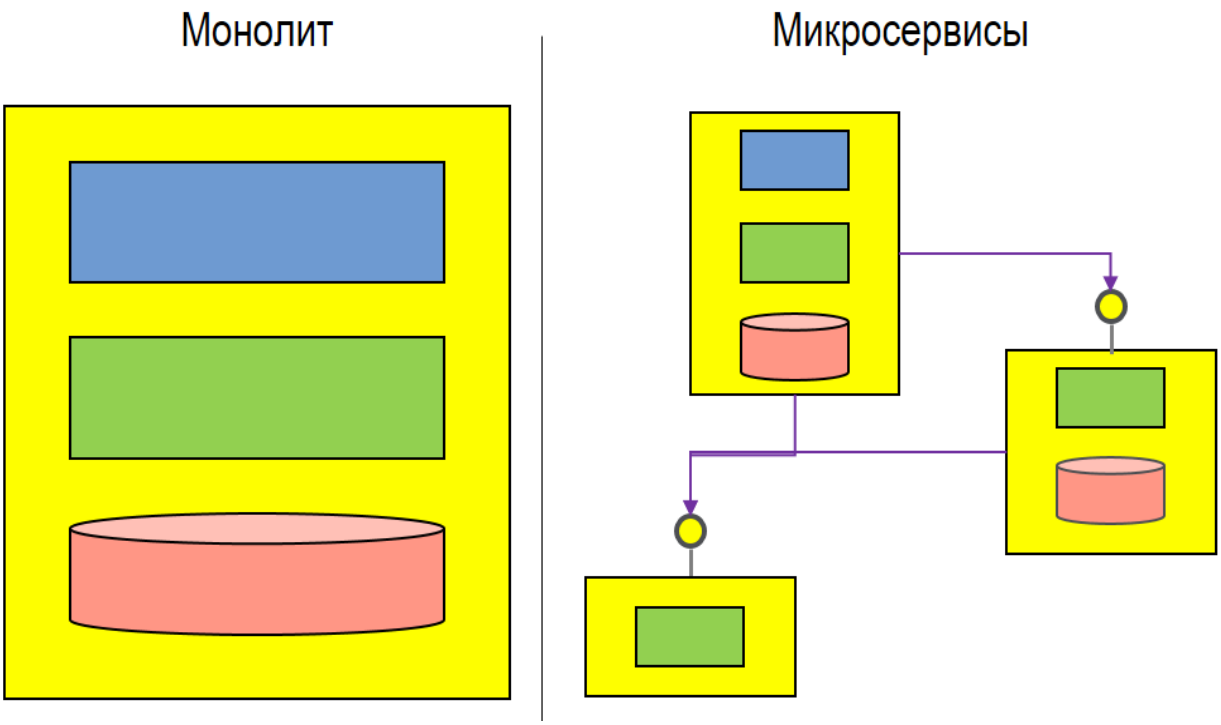


Рисунок 3.1 – Порівняння моноліту та мікросервісу[13]

Можна виділити вісім властивостей мікросервісу:

1) невеликий;
2) незалежний;

3) будується навколо бізнес-потреби та використовує обмежений контекст (Bounded Context);

4) взаємодіє з іншими мікросервісами через мережу на основі патерну Smart endpoints and dumb pipes;

5) його розподілена суть зобов'язує використати підхід Design for failure;

6) централізація обмежена зверху на мінімумі;

7) процеси його розробки та підтримки вимагають автоматизації;

8) його розвиток ітераційний.

У той же час методологія розбиття на окремі мікросервіси змушує дотримуватися жорсткого їхнього поділу, адже вони повинні відповідати більш жорстким критеріям незалежності (рис. 3.2).

Так, кожен мікросервіс працює у своєму процесі і тому має явно позначити свій API. Враховуючи, що інші компоненти можуть використовувати лише цей API, і до того ж він віддалений, мінімізація зв'язків стає життєво важливою.

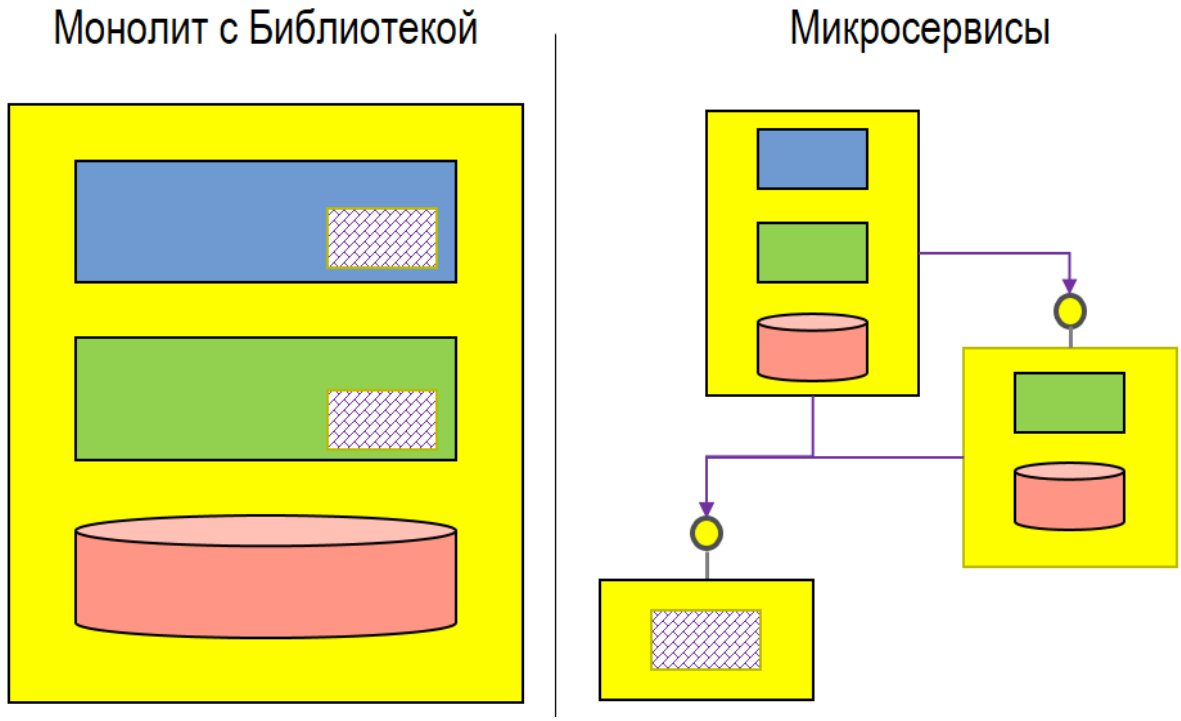


Рисунок 3.2 – Порівняння моноліту с бібліотекою та мікросервісу[13]

Такий розподіл дає явний виграш із погляду незалежного розвитку різних компонентів. І з урахуванням цього різні мови вводять конструкції, що дозволяють явне створення незалежних компонентів (наприклад, модулі Java 9), і це перестає бути прерогативою мікросервісного підходу.

Незалежність мікросервісів дозволяє організувати незалежний життєвий цикл розробки, створювати окремі збирання, тестувати та розгортати.

Оскільки розмір мікросервісів невеликий, очевидно, що у великих системах їх буде чимало. Керувати ними вручну буде складно. Тому команда повинна мати прийнятний рівень автоматизації відповідно до Continuous integration і Continuous Delivery.

Мікросервіс – це найпростіша одиниця, сервіс, який приймає вхідні запити для здійснення дії. Це може бути бекенд сервіс, який доступний цілодобово та без вихідних, або функція, яка викликається, коли відбувається подія. Простими словами, функція або набір функцій, доступних через певний API через мережу. Отже, це бекенд служба, розгорнута на сервері. У якомусь сенсі це монолітний додаток. Однак він не несе в собі всю функціональність системи, а лише меншу частинку логіки. На відміну від моноліту, отриманий додаток побудований як набір відносно невеликих незалежних служб, що називаються «мікросервісами», які комунікують через комп’ютерну мережу. Можна сказати, мікросервіси – це ті самі логічні модулі монолітного додатка, які розподілені через комп’ютерну мережу, замість того, щоб працювати в рамках одного процесу (пристрою).

У неструктурованому розподіленому середовищі дуже легко втратити контроль над множиною мікросервісів. З цієї причини рекомендується контролювати мікросервіси, структуруючи їх за ярусами. Існують різні архітектурні шаблони, які можуть допомогти. Загальним для цих шаблонів є те, що всі вони зосереджені на experience.

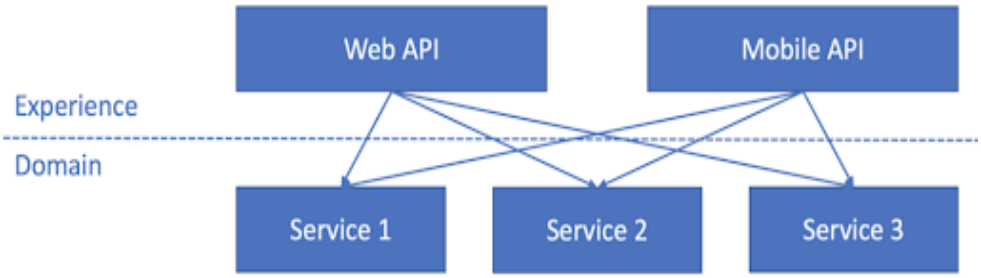


Рисунок 3.3 – Структура Experience сервіса[16]

3.2 WebSockets

Веб-сокети це просунута технологія, що дозволяє відкрити постійне двонаправлене мережне з'єднання між браузером користувача та сервером. За допомогою його API ви можете відправити повідомлення на сервер і отримати відповідь без виконання http-запиту, причому цей процес буде подієво-керованим.

Для встановлення з'єднання WebSocket клієнт та сервер використовують протокол, схожий на HTTP. Клієнт формує спеціальний HTTP-запит, який сервер відповідає певним чином (рис. 3.4).

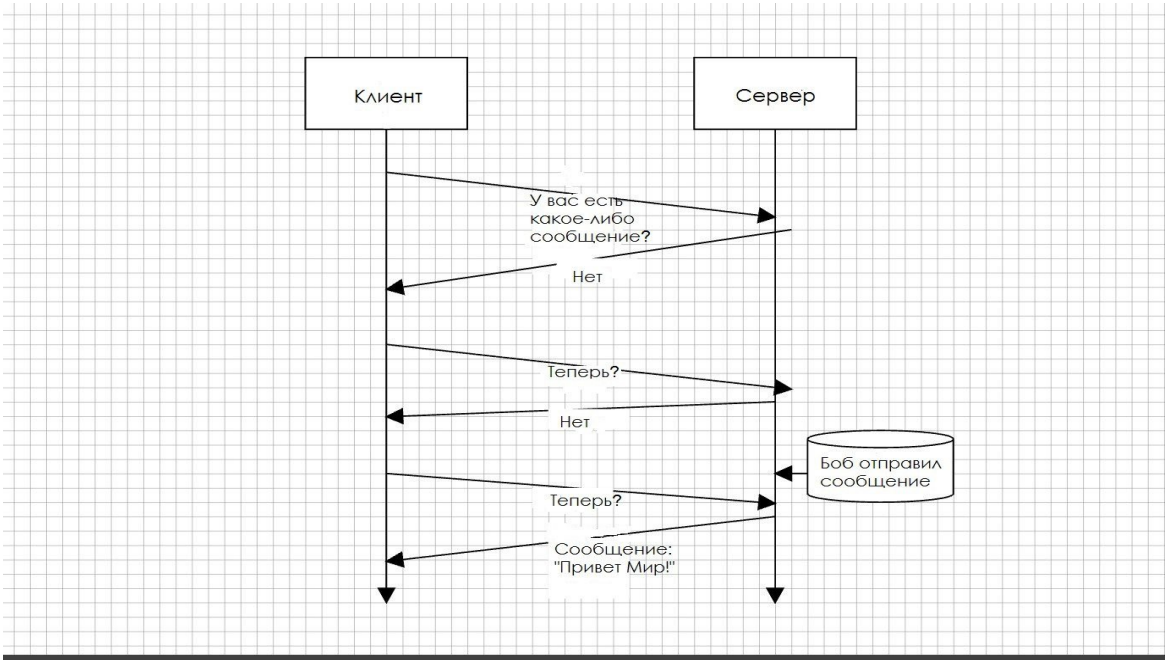


Рисунок 3.4 – Приклад роботи HTTP[2]

Веб-сокетам для відповіді не потрібні повторювані запити. Достатньо виконати один запит і чекати на відгук. Можна просто слухати сервер, який надсилатиме повідомлення в міру готовності (рис. 3.5).

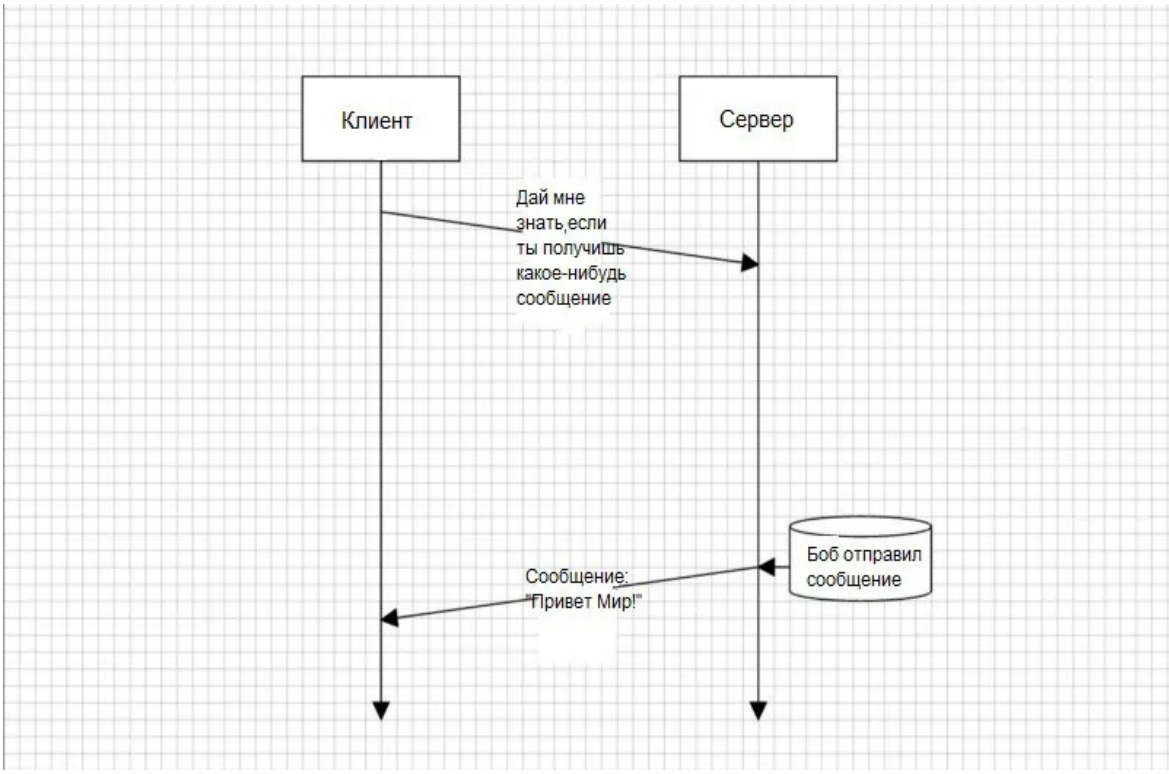


Рисунок 3.5 – Приклад роботи websockets[2]

Веб-сокети є однією з найперспективніших веб-технологій, яку вже зараз використовують багато розробників. Вона відмінно підходить для взаємодії в режимі реального часу, в тому числі, онлайн-ігор (рис. 3.6).

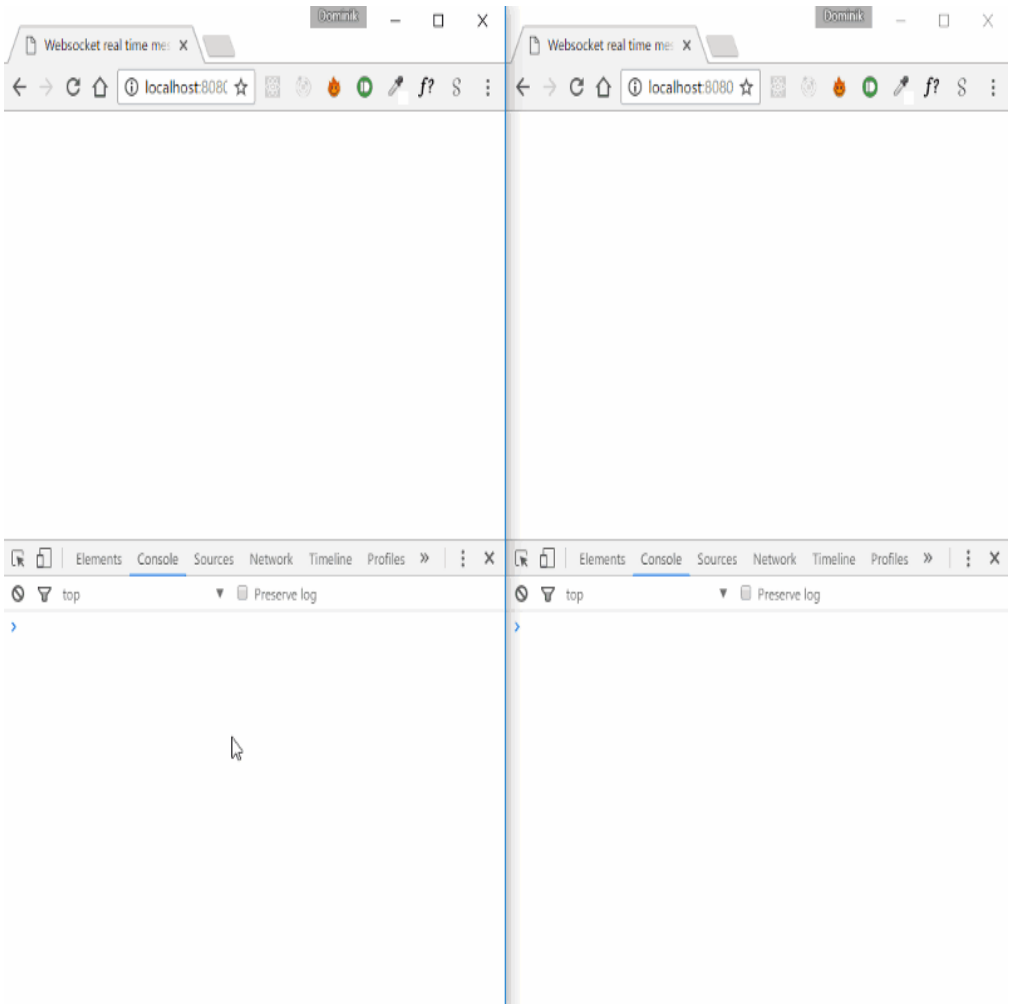


Рисунок 3.6 – Приклад роботи вебсокетів[6]

3.3 План розробки додатку

Отже, для переходу до розробки додатку спочатку треба розписати план дій та намалювати схему. Пропонується наступна блок-схема для додатку, на основі якої і буде розроблена мікросервісна архітектура (рис. 3.7).

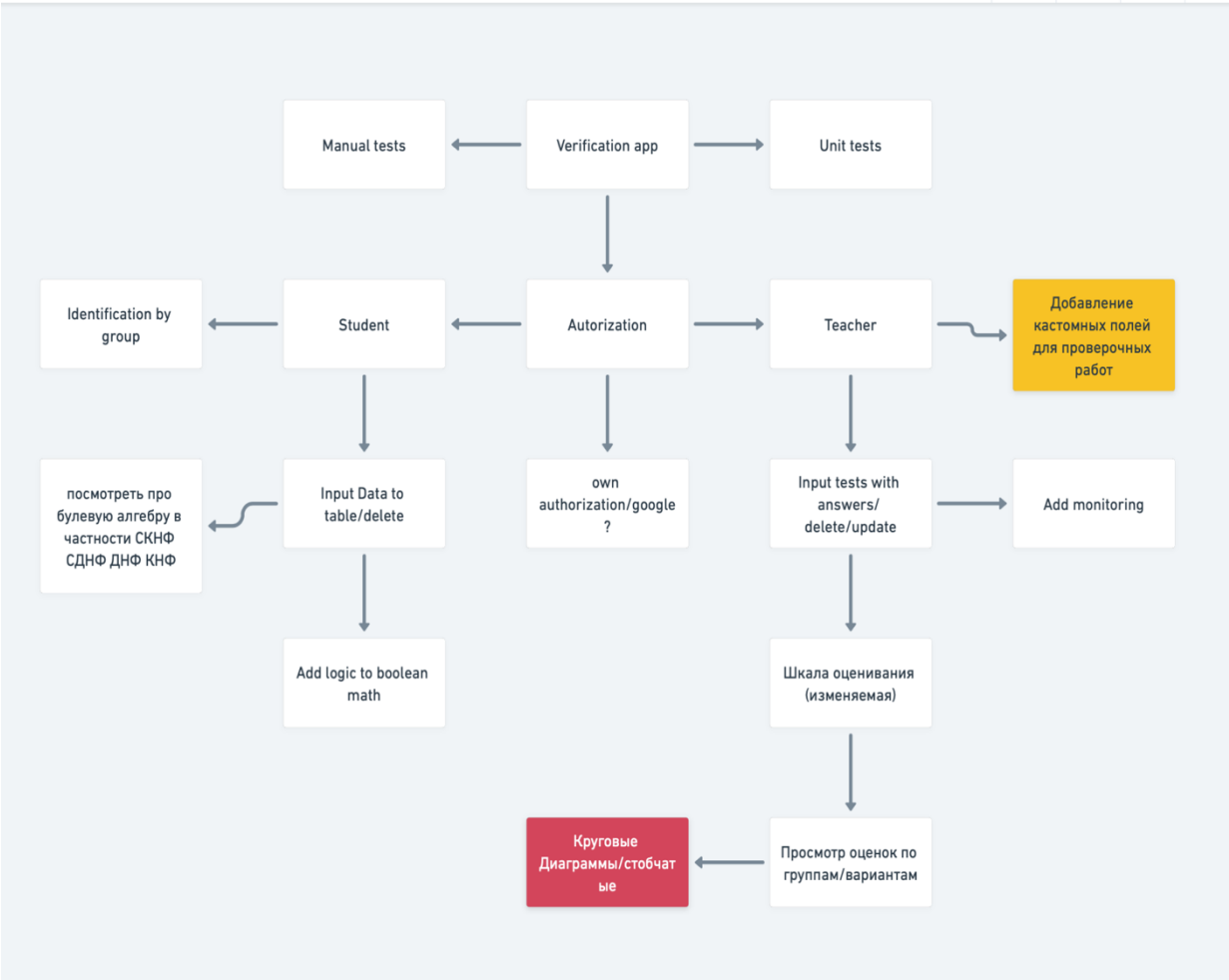


Рисунок 3.7 – Блок-схема додатку

Отже, з блок-схеми видно, що першим кроком являється розробка сервісу авторизації та система ролей у додатку, тому і слід виконати опис саме цих сервісів.

3.4 Сервіс авторизації

Цей сервіс відповідатиме за реєстрацію як студентів, так і викладачів.

```
import
{
        Column, CreateDateColumn, Entity, Index, PrimaryGeneratedColumn,
    UpdateDateColumn,
    } from 'typeorm';

    @Entity('users')
```

```
export class UserEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({
    unique: true,
    nullable: true,
  })
  email: string;

  @Column({
    unique: true,
    nullable: true, // TODO: DELETE THIS LINE
  })
  userId: string;

  @Index()
  @Column({
    nullable: true,
  })
  passwordHash: string;

  @Column({
    type: 'json',
    default: [],
  })
  roles: string[] = [] as string[];

  @Index()
  @Column({
    default: false,
    type: 'boolean',
  })
  isBanned = false;

  @Column({
    nullable: true,
  })
  googleId: string;

  @Column({
```

```
  nullable: true,
 })
 facebookId: string;

 @Column({
  type: 'jsonb',
  default: [],
 })
 appIds: number[];

 @CreateDateColumn()
 createdAt: Date;

 @UpdateDateColumn()
 updatedAt: Date;

}
```

Приклад вигляду зареєстрованих людей у додатку, де будуть зберігатися всі основні креденшіали юзерів.

Наступний фрагмент коду продемонструє роботу з базою данних постгрес

```
 import {
 DeepPartial,
 EntityRepository,
 Repository }
 from 'typeorm';
                        import {
                          AsyncEitherWithErr, errorsRegistry, left, right,
                        } from '@realisnetwork/errors-registry';
                        import { UserEntity } from
                        '../../entities/main/UserEntity';
                        import { SocialMediaProvider } from
                        '../../common/types/auth';

                        @EntityRepository(UserEntity)
                        export class UserRepository extends
                        Repository<UserEntity> {
                          async add(param: DeepPartial<UserEntity>):
                        AsyncEitherWithErr<UserEntity> {
                            try {
                              const user = this.create(param);

                              await this.save(user);
                              return right(user);
                            } catch (e) {
                              return left({
                                type: errorsRegistry.db.insert,
```

```
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async getByUserId(userId: string):
AsyncEitherWithErr<UserEntity> {
    try {
      const user = await this.createQueryBuilder('u')
        .andWhere('u.userId = :userId', { userId })
        .getOne();
      return right(user);
    } catch (e) {
      return left({
        type: errorsRegistry.db.select,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async getByEmailOrProviderId(
    { email, providerId, provider }: { email: string,
providerId: string, provider: string },
  ): AsyncEitherWithErr<UserEntity> {
    try {
      const providerColumn = provider ===
SocialMediaProvider.Google ? 'googleId' :
'facebookId';

      return right(await this.createQueryBuilder('u')
        .where(`u.email = '${email}'`)
        .orWhere(`u.${providerColumn} =
'${providerId}'`)
        .getOne());
    } catch (e) {
      return left({
        type: errorsRegistry.db.select,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async updateById(id: number, props:
DeepPartial<UserEntity>):
AsyncEitherWithErr<true> {
    try {
      const { tableName } = this.metadata;
```

```
      await this.createQueryBuilder(tableName)
        .update()
        .set({ ...props })
        .where(`${tableName}.id = :id`, { id })
        .execute();

      return right(true);
    } catch (e) {
      return left({
        type: errorsRegistry.db.select,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async updateAppListByUserId({ userId, appIds }:
{ userId: string, appIds: number[] }):
AsyncEitherWithErr<true> {
    try {
      const { tableName } = this.metadata;

      await this.createQueryBuilder(tableName)
        .update()
        .set({ appIds })
        .where(`${tableName}.user_id = '${userId}'`)
        .execute();

      return right(true);
    } catch (e) {
      return left({
        type: errorsRegistry.db.update,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async findByIdStrict(id: number):
AsyncEitherWithErr<UserEntity> {
    try {
      const user = await this.createQueryBuilder('u')
        .andWhere('u.id = :id', { id })
        .getOne();

      if (!user) {
        return left({
          type: errorsRegistry.db.notFound,
          msg: `User not found by id: ${id}`,
        });
```

```
      }

      return right(user);
    } catch (e) {
      return left({
        type: errorsRegistry.db.select,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async banUserByUserId(userId: string):
AsyncEitherWithErr<true> {
    try {
      const { tableName } = this.metadata;

      await this.createQueryBuilder(tableName)
        .andWhere(`${tableName}.user_id =
:userId`, { userId })
        .update()
        .set({ isBanned: true })
        .execute();

      return right(true);
    } catch (e) {
      return left({
        type: errorsRegistry.db.update,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async unbanUserByUserId(userId: string):
AsyncEitherWithErr<true> {
    try {
      const { tableName } = this.metadata;

      await this.createQueryBuilder(tableName)
        .andWhere(`${tableName}.user_id =
:userId`, { userId })
        .update()
        .set({ isBanned: false })
        .execute();

      return right(true);
    } catch (e) {
      return left({
        type: errorsRegistry.db.update,
```

```
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async findByEmailStrict(email: string):
AsyncEitherWithErr<UserEntity> {
    try {
      const user = await this.createQueryBuilder('u')
        .andWhere('u.email = :email', { email })
        .getOne();

      if (!user) {
        return left({
          type: errorsRegistry.db.notFound,
          msg: 'User not found',
        });
      }

      return right(user);
    } catch (e) {
      return left({
        type: errorsRegistry.db.select,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async findByEmail(email: string):
AsyncEitherWithErr<UserEntity> {
    try {
      const user = await this.createQueryBuilder('u')
        .andWhere('u.email = :email', { email })
        .getOne();
      return right(user);
    } catch (e) {
      return left({
        type: errorsRegistry.db.select,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async updatePasswordById(id: number,
passwordHash: string): AsyncEitherWithErr<true>
{
    try {
```

```
      await this.createQueryBuilder()
        .andWhere('id = :id', { id })
        .update()
        .set({ passwordHash })
        .execute();

      return right(true);
    } catch (e) {
      return left({
        type: errorsRegistry.db.update,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async updateEmailById(id: number, email:
string): AsyncEitherWithErr<true> {
    try {
      await this.createQueryBuilder('u')
        .andWhere('u.id = :id', { id })
        .update()
        .set({ email })
        .execute();
    } catch (e) {
      return left({
        type: errorsRegistry.db.update,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }

  async deleteById(id: number):
AsyncEitherWithErr<true> {
    try {
      await this.createQueryBuilder('u')
        .andWhere('u.id = :id', { id })
        .delete()
        .execute();
    } catch (e) {
      return left({
        type: errorsRegistry.db.update,
        msg: (e as Error).message,
        trace: (e as Error).stack,
      });
    }
  }
}
```

І найважливіше – це код самого сервісу та його ендпоінти:

```
import {
singleton
} from
'tsyringe';
                import { Logger, LoggerCategorized } from '@realisnetwork/logger';
                import { DeepPartial, EntityManager } from 'typeorm';
                import {
                  AsyncEitherWithErr,
                } from '@realisnetwork/errors-registry';
                import { UserRepository } from '../repositories/main/UserRepository';
                import { UserEntity } from '../entities/main/UserEntity';
                import { SocialMediaProvider } from '../common/types/auth';
```

```
@singleton()
export class UserService {
  private readonly logger: LoggerCategorized;
  static publicMethods = (): string[] => [];

  constructor(
    private readonly em: EntityManager,
    loggerFactory: Logger,
  ) {
    this.logger = loggerFactory.getLogger(`(${this.constructor.name})`);
  }

  async getByEmailStrict(email: string):
AsyncEitherWithErr<UserEntity> {
    return
this.em.getCustomRepository(UserRepository).findByEmailStrict(email
);
  }

  async getByEmail(email: string): AsyncEitherWithErr<UserEntity> {
    return
this.em.getCustomRepository(UserRepository).findByEmail(email);
  }

  async add(params: DeepPartial<UserEntity>):
AsyncEitherWithErr<UserEntity> {
    return this.em.getCustomRepository(UserRepository).add(params);
  }

  async getByUserId(userId: string): AsyncEitherWithErr<UserEntity> {
    return
this.em.getCustomRepository(UserRepository).getByUserId(userId);
  }

  async updatePassword(
    { id, passwordHash, transactionEM }: { id: number, passwordHash:
string, transactionEM?: EntityManager },
  ): AsyncEitherWithErr<true> {
    const tem = transactionEM ?? this.em;
    return
tem.getCustomRepository(UserRepository).updatePasswordById(id,
passwordHash);
```

```
  }

  async updateAppList(
    { userId, appIds, transactionEM }: { userId: string, appIds: number[],
transactionEM?: EntityManager },
  ): AsyncEitherWithErr<true> {
    const tem = transactionEM ?? this.em;
    return
tem.getCustomRepository(UserRepository).updateAppListByUserId({
userId, appIds });
  }

  async getByEmailOrProviderId(
    { email, providerId, provider }: { email: string, providerId: string,
provider: SocialMediaProvider },
  ): AsyncEitherWithErr<UserEntity> {
    return
this.em.getCustomRepository(UserRepository).getByEmailOrProviderI
d({ email, providerId, provider });
  }

  async updateGoogleIdById({ id, googleId }: { id: number, googleId:
string }): AsyncEitherWithErr<true> {
    return
this.em.getCustomRepository(UserRepository).updateById(id, {
googleId });
  }

  async updateFacebookIdById({ id, facebookId }: { id: number,
facebookId: string }): AsyncEitherWithErr<true> {
    return
this.em.getCustomRepository(UserRepository).updateById(id, {
facebookId });
  }
}
```

Отже, як можна побачити за допомогою Typeorm та typescript, код виглядає охайніше та більш читабельним, завдяки мікросервісній архітектурі.

3.4 Сервіс ролей

Сервіс Ролей буде відповідати ролі студента та викладача у кожного з яких будуть свої рівні доступу та повноваги. Для цього розробляються 3 таблиці: у першій будуть зберігатися ролі, у другій юзери з ролями які будуть

між собою пов’язані та третя таблиця буде відповідати за повноваги ролі.

```
import
'reflect-
metadata';
                import {
                  Column, Entity, PrimaryGeneratedColumn, OneToMany,
                ManyToMany, JoinTable, Unique,
                } from 'typeorm';
                import { UserEntity } from './UserEntity';
                import { PermissionEntity } from './PermissionEntity';

                @Unique(['name'])
                @Entity({
                  name: 'roles',
                  synchronize: true,
                })
                export class RoleEntity {
                  @PrimaryGeneratedColumn()
                  id: number;

                  @Column({
                    type: 'varchar',
                  })
                  name: string;

                  @OneToMany(() => PermissionEntity, (permision) =>
                permision.roleId)
                  methods: PermissionEntity[];

                  @OneToMany(() => UserEntity, (userEntity) => userEntity.role)
                  users: UserEntity[];
                }

    import
'reflect-
metadata';
                import {
                  Column, Entity, PrimaryGeneratedColumn, Unique, ManyToOne,
                } from 'typeorm';
```

```
// import { RoleSubGroupEntity } from './RoleSubGroupEntity';
import { RoleEntity } from './RoleEntity';

@Unique(['userId', 'role'])
@Entity('users')
export class UserEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  userId: string;

  @Column({
    default: true,
  })
  isActive:boolean;

  @ManyToOne(() => RoleEntity, (roleEntity) => roleEntity.users, {
nullable: true, onDelete: 'CASCADE' })
  role: RoleEntity;
}

  import
  {
                Column, Entity, ManyToOne,
              PrimaryGeneratedColumn,
              } from 'typeorm';
              import { RoleEntity } from './RoleEntity';

              @Entity('permissions')
              export class PermissionEntity {
                @PrimaryGeneratedColumn()
                id: number;

                @Column({
                  nullable: true,
                  default: 'somePermission',
                })
                permissionName: string;

                @ManyToOne(() => RoleEntity, (role) => role.methods,
              { onDelete: 'CASCADE' })
                roleId: number;
              }
```

3.6 Api-Gateway

API Gateway – це центральні двері для всіх публічних API. Це єдина

точка входу для Experience APIs, доступних для клієнтських програм, і є важливою складовою найкращих практик управління API (API Management). По суті, він передає запити реальним бекенд сервісам та здатен приймати рішення. Частіше за все, API Gateway відповідає за такий функціонал, як:

- Request handling;
- Upstream (або downstream) data transfer (або transformation);
- Access control;
- Security policies;
- Throttling;
- Rate limiting;
- Analytics;
- Caching.

У розробленому додатку знадобиться апігейтвей, щоб усі запроси шли тільки через цей мікросервіс, так буде легче викроистовувати ендпоінти на сервері:

```
import { agents } from
'@realisnetwork/agents';
                                import {
                                  AgentTopicProcessed, SocketRequest,
                                } from '@realisnetwork/types';
                                import {
                                  left, right, AsyncEitherWithErr, EitherWithErr,
                                errorsRegistry,
                                } from '@realisnetwork/errors-registry';
                                import { delay, inject, singleton } from 'tsyringe';
                                import { ConfigService } from
                                '@realisnetwork/configurator';
```

```
import { Logger, LoggerCategorized } from
'@realisnetwork/logger';
import { AjvWrapper } from '../../lib/ajv/AjvWrapper';
import { SocketProcessedRequest } from
'../types/transport';
import { WebSocketClient } from
'../../lib/ws/WebSocketClient';
import { ConnectUserCallback } from
'../types/usersOnline';
import { Auth } from '../auth/Auth';
import { baseRequest } from
'../../helpers/definitions/baseDefinitions';
import { UserRedisRepository } from
'../redis/repository/UserRedisRepository';

export const processResponseTimeLog = (id: string)
=> `PROCESS RESPONSE[${id}]`;

@singleton()
export class Router {
  private highestRequestProcessTime = 0;
  private highestRequest: string;

  private readonly logger: LoggerCategorized;

  private readonly agentNames: Set<string>;
  private readonly keycloakNamespace: string;
  private readonly topics: Map<string,
AgentTopicProcessed>;
  private readonly baseRequestSchema =
JSON.stringify(baseRequest);
  private readonly onConnectUserCallbacks = [] as
ConnectUserCallback[];

  constructor(
    private readonly auth: Auth,
    private readonly configService: ConfigService,
    private readonly ajvWrapper: AjvWrapper,
    @inject(delay(() => UserRedisRepository)) private
readonly userRedisRepository: UserRedisRepository,
    loggerFactory: Logger,
```

```
  ) {
    this.keycloakNamespace =
this.configService.get('keycloak.namespace');

    this.logger =
loggerFactory.getLogger(`${this.constructor.name}`);

    const agentNames = new Set<string>();

    // @ts-ignore
    this.ajvWrapper.addSchema({ rawSchema:
baseRequest, method: 'base' });

    this.topics = new Map(
      agents.reduce((acc, { agent, service, topics }) => {
        agentNames.add(agent);

        const topicsEntries = Object.entries(topics);
        if (topicsEntries.length) {
          topicsEntries.forEach(([method, agentTopic]) =>
{
            const topic = agentTopic.topic ||
`${agent}_${service}_${method}`;
            const agentName = agentTopic.agentName ||
agent;

            const schema = {
              type: 'object',
              properties: {},
              required: [] as string[],
              additionalProperties: false,
              ...agentTopic.schema,
            };

            acc.push([topic, {
              ...agentTopic,
              topic,
              agentName,
              service,
```

```
            schema: JSON.stringify(schema, null, 2),
          }]);

          // @ts-ignore
          this.ajvWrapper.addSchema({ rawSchema:
JSON.stringify(schema), method: topic });
        });
      }

      return acc;
    }, [] as [string, AgentTopicProcessed][]),
  );

  this.agentNames = agentNames;

  setTimeout(() => this.logger.debug(
    'HIGHEST REQ PROCESS TIME: ',
    this.highestRequestProcessTime,
    this.highestRequest,
  ), 2000);
 }

 resolveAgent(agent: string, method: string):
EitherWithErr<AgentTopicProcessed> {
  if (!this.agentNames.has(agent)) {
    return left({
      type: errorsRegistry.bff.invalidAgent,
      msg: `Agent '${agent}' not found`,
    });
  }

  const agentTopic =
this.topics.get(`${agent}_${method}`);

  if (!agentTopic) {
    return left({
      type: errorsRegistry.bff.invalidMethod,
      msg: `Method '${method}' not found`,
    });
```

```
    }

    return right(agentTopic);
  }

  async processRequest(
    request: SocketRequest, wsClient:
WebSocketClient,
  ): AsyncEitherWithErr<[AgentTopicProcessed,
SocketProcessedRequest]> {
    const timeLogLabel = `PROCESS REQUEST
${request.id}`;

    const timeAtStart = Number(Date.now());
    console.time(timeLogLabel);

    const processedRequest = {
      ...request,
      clientId: wsClient.clientId,
      authInfo: null,
      topicResponse: process.env.CLIENT_ID,
    } as unknown as SocketProcessedRequest;

    const agentEither =
this.resolveAgent(processedRequest.agent,
processedRequest.method);
    if (agentEither.isLeft()) {
      return (agentEither as unknown) as
AsyncEitherWithErr<[AgentTopicProcessed,
SocketProcessedRequest]>;
    }

    const { value: agent } = agentEither;

    // const registerConnectionEither = await
this.userRedisRepository
    //
.registerConnectionRequest(processedRequest.agent,
processedRequest.method, wsClient.clientId);
```

```
    //
    // if (registerConnectionEither.isLeft()) {
    //   return (
    //     registerConnectionEither as unknown
    //   ) as AsyncEitherWithErr<[AgentTopicProcessed,
SocketProcessedRequest]>;
    // }

    const validationBaseEither =
this.ajvWrapper.validate({ method: 'base', data:
request });
    if (validationBaseEither.isLeft()) {
      return (validationBaseEither as unknown) as
AsyncEitherWithErr<[AgentTopicProcessed,
SocketProcessedRequest]>;
    }
    const validationEither = this.ajvWrapper.validate({
method: agent.topic, data: request.params });
    if (validationEither.isLeft()) {
      return (validationEither as unknown) as
AsyncEitherWithErr<[AgentTopicProcessed,
SocketProcessedRequest]>;
    }

    const authInfoEither = await this.auth
      .isMethodAllowed(request.method,
agent.accessLevel, request);
    if (authInfoEither.isLeft()) {
      return (authInfoEither as unknown) as
AsyncEitherWithErr<[AgentTopicProcessed,
SocketProcessedRequest]>;
    }

    const { value: { authorized, info } } = authInfoEither;
    if (!authorized) {
      return left({
        type: errorsRegistry.permissions.notAllowed,
        msg: 'Not authorized',
      });
    }

    if (authorized) {
      if (processedRequest.authInfo) wsClient.token =
processedRequest.clientId;
      this.onConnectUserCallbacks.forEach((cb) => cb(
        processedRequest.clientId,
        (info as { [k: string]: string }).userId,
```

```
        wsClient,
        (info as { [k: string]: string
})[`${this.keycloakNamespace}/timezone`],
        (info as { [k: string]: string
})[`${this.keycloakNamespace}/country`],
      ));
    }

    processedRequest.authInfo = {
      userId: info.hasOwnProperty('userId') ? (info as {
[k: string]: string }).userId : null,
      ipAddress: wsClient.ipAddress,
    };

    console.timeEnd(timeLogLabel);
    const processTimeDiff = Number(Date.now()) -
timeAtStart;

    if (this.highestRequestProcessTime <
processTimeDiff) {
      this.highestRequestProcessTime =
processTimeDiff;
      this.highestRequest = JSON.stringify(request);
    }

    return right([agent, processedRequest]);
  }

  onConnectUser(cb: ConnectUserCallback) {
    this.onConnectUserCallbacks.push(cb);
  }
}
```

Важливим є WebSocket Server, ніжче наведено приклад його роботи із додатку:

```
import {
EventEmitter
} from
'events';
                import { Logger, LoggerCategorized } from
                '@realisnetwork/logger';
                import { StatusCodes } from 'http-status-codes';
                import {
```

```
  App as uWSApp, TemplatedApp, AppOptions,
WebSocketBehavior,
} from 'uWebSockets.js';
import { ConfigService } from '@realisnetwork/configurator';
import {
  MicroserviceResponse,
  Request,
  EventBroadcast,
} from '@realisnetwork/types';
import { v4 as uuidV4 } from 'uuid';
import { singleton } from 'tsyringe';
import { Nats } from '@realisnetwork/nats-streaming';
import { TextDecoder } from 'util';
import * as process from 'process';
import { EitherHealthCheckError } from '@realisnetwork/errors-
registry';
import {
  ErrorMessage, HasNecessaryHeaders, WSClient, WsTopics,
} from '../../common/types/ws';
import { WebSocketClient } from './WebSocketClient';
import { WebServer } from '../http/WebServer';
import { isNatsHealthy } from '../../helpers/healthCheckCallbacks';

@singleton()
export class WebSocketServer extends EventEmitter {
  private app: TemplatedApp;
  private readonly port: number;
  private readonly keepAlive: number;
  private readonly wsOptions: AppOptions;

  private connectionsByClient = new Map<string, WSClient>();
  private connectionsByUser = new Map<string, WSClient>();

  private readonly logger: LoggerCategorized;
```

```
  constructor(
    private readonly nats: Nats,
    private readonly webServer: WebServer,
    private readonly configService: ConfigService,
    private readonly loggerFactory: Logger,
  ) {
    super();
    this.logger =
loggerFactory.getLogger(`(${this.constructor.name})`);

    this.wsOptions =
this.configService.get<AppOptions>('ws.options');

    this.port = this.configService.get<number>('ws.port');
    this.keepAlive = this.configService.get<number>('ws.keepAlive');

    this.init();

    setTimeout(() => this.logger.debug('ACTIVE CONNECTIONS
COUNT: ', this.connectionsByClient.size), 2000);

    return this;
  }

  init(): void {
    this.app = uWSApp()
      .ws('/*', {
        upgrade: this.onUpgrade.bind(this) as
WebSocketBehavior['upgrade'],
        open: this.onClientConnection.bind(this) as
WebSocketBehavior['open'],
        message: this.onClientMessage.bind(this) as
WebSocketBehavior['message'],
        close: this.onClientDisconnection.bind(this) as
WebSocketBehavior['close'],
        pong: this.onPong.bind(this) as WebSocketBehavior['pong'],
      }).get('/health', (res) => {
        const callbacks = [isNatsHealthy(this.nats)];
```

```
        const errors = callbacks.reduce((acc, callback) => {
          const either = callback();
          if (either.isLeft()) acc.push(either.value);

          return acc;
        }, [] as EitherHealthCheckError[]);

        if (!errors.length) {

res.writeStatus(StatusCodes.OK.toString()).write(JSON.stringify({
message: 'ok', code: StatusCodes.OK }));
          res.end();

          return res;
        }


res.writeStatus(StatusCodes.INTERNAL_SERVER_ERROR.toString()
).write(
          JSON.stringify({
            message: 'Something went wrong!',
            code: StatusCodes.INTERNAL_SERVER_ERROR,
            healthy: false,
            errors,
          }),
        );
        res.end();

        return res;
      }).listen(this.port, (token: boolean) => {
        if (token) return this.logger.info(`webSocket: listening to port
${this.port}`);

        this.logger.error('Server is down');
        return process.nextTick(() => {
          process.exit();
        });
      });
  }
```

```
  private onUpgrade: WebSocketBehavior['upgrade'] = (res, req,
context) => {
    console.log('x-original-forwarded-for', req.getHeader('x-original-
forwarded-for'));
    console.log('x-forwarded-for', req.getHeader('x-forwarded-for'));
    console.log('x-real-ip', req.getHeader('x-real-ip'));
    res.upgrade<HasNecessaryHeaders>(
      {
        ipAddress:
          req.getHeader('x-original-forwarded-for') || req.getHeader('x-
real-ip'),
        userAgent: req.getHeader('user-agent'),
        origin: req.getHeader('origin'),
      },
      req.getHeader('sec-websocket-key'),
      req.getHeader('sec-websocket-protocol'),
      req.getHeader('sec-websocket-extensions'),
      context,
    );
  };

  // eslint-disable-next-line consistent-return
  private onClientMessage: WebSocketBehavior['message'] =
(wsClient: WSClient, buf) => {
    const decoder = new TextDecoder();
    const json = decoder.decode(buf);

    let message: Request<unknown>;

    try {
      message = JSON.parse(json) as Request<unknown>;
      if (message === null || message.constructor !== Object) {
        this.logger.warn('invalid ws message received');
        return this.app.publish(
          `${WsTopics.TO}${wsClient.clientId}`,
          JSON.stringify({ error: 'Message should be a valid JSON object'
}),
        );
      }
    } catch (e) {
      this.logger.warn('invalid ws message received', json);
      return this.app.publish(
        `${WsTopics.TO}${wsClient.clientId}`,
```

```
        JSON.stringify({ error: 'Message should be a valid JSON object'
}),
      );
    }

    message.clientId = wsClient.clientId;

    this.emit('message', message, wsClient.client);
  };

  private onClientConnection: WebSocketBehavior['open'] = (ws:
WSClient) => {
    ws.clientId = uuidV4();

    const client = new WebSocketClient(
      ws, this.keepAlive, this.loggerFactory,
    );

    ws.client = client;

    this.connectionsByClient.set(ws.clientId, ws);

    ws.subscribe(WsTopics.ALL);
    ws.subscribe(`${WsTopics.TO}${ws.clientId}`);

    this.logger.info('new connection is opened', { clientId: ws.clientId,
ip: ws.ipAddress });
  };

  private onClientDisconnection: WebSocketBehavior['close'] = (ws:
WSClient, code, buf) => {
    const decoder = new TextDecoder();
    const msg = decoder.decode(buf);

    const { clientId, client: { userId } } = ws;
```

```
    this.connectionsByClient.delete(clientId);
    this.connectionsByUser.delete(userId);
    ws.client.onClose();
    this.emit('close', clientId, code, msg); // TODO: get on close here
  };

  private onPong: WebSocketBehavior['pong'] = (ws: WSClient) => {
    ws.client.heartbeat();
  };

  notifyAll(message: EventBroadcast) {
    this.app.publish(WsTopics.ALL, JSON.stringify(message));
  }

  sendTo(clientId: string, message: MicroserviceResponse<unknown,
unknown> | ErrorMessage) {
    this.logger.debug(`${WsTopics.TO}${clientId}`, { msg:
JSON.stringify(message) });
    this.app.publish(`${WsTopics.TO}${clientId}`,
Buffer.from(JSON.stringify(message), 'utf-8'));
  }

  sendToUserId(userId: string, message:
MicroserviceResponse<unknown, unknown> | ErrorMessage) {
    this.logger.debug(`${WsTopics.USER_ID}${userId}`, { msg:
JSON.stringify(message) });
    this.app.publish(`${WsTopics.USER_ID}${userId}`,
Buffer.from(JSON.stringify(message), 'utf-8'));
  }

  findClientByClientId(clientId: string) {
    return this.connectionsByClient.get(clientId);
  }

  mapClientIdToUserId(clientId: string, userId: string) {
    const client = this.connectionsByClient.get(clientId);
    if (!client) return;
```

```
    this.connectionsByUser.set(userId, client);
  }

  disconnectUser(userId: string) {
    const connectionByUserId = this.connectionsByUser.get(userId);
    this.logger.debug('CONNECTION BY USER FOR DISCONNECT:
', connectionByUserId, userId);
    if (connectionByUserId) {
      connectionByUserId.close();
      this.connectionsByUser.delete(userId);
    }
  }
}
```

ВИСНОВКИ

У ході виконання роботи розроблено систему верифікації контрольних робіт на основі мікросервісної архітектури та програмного застосунку в рамках реалізації хмарних сервісів кіберуніверситету, що дає можливість підвищити якість збирання, збереження й обробки інформації для обліку успішності студентів, а також знизити часові витрати на перевірку.

Виконано аналіз сучасних технологій розробки серверних додатків. Виконано порівняння мікро-сервісної архітектури з монолітом, визначено переваги та недоліки.

Для написання та розробки архітектури проекту використано досить новітніх технологій та функцій. Створення складних додатків за своєю суттю є важко справою. Монолітна архітектура має сенс лише для простих, легких додатків. Мікросервісна архітектура, незважаючи на недоліки та проблеми впровадження, є кращим вибором для складних продуктів що постійно розвиваються.

Якщо моноліт розділити на окремі частини і не брати до уваги ключові принципи проектування архітектури, орієнтованої на мікросервіси, це може створити проблеми, а не елегантне рішення. В даній роботі було розглянуто основні принципи проектування мікросервісних програмних систем, деякі ключові моменти та основні паттерни для успішного розгортання MSA. Було детально розглянуто деякі компоненти мікросервісної архітектури, до яких відносяться API Gateway, балансувальник навантаження, автовимикач.

Значну роль відіграє у розробці програмної системи необхідне середовище, яке забезпечується системами віртуалізації та постійної інтеграції. Було розглянуто систему автоматичного контейнерного розгортання Docker. Були продемонстровані ключові моменти безпеки мікросервіснох архітектури. Було розглянуто масштабування мікросервісів, мета, проблеми, паттерни їх вирішення. Розглянуто деякі підходи для

переходу з моноліту на мікросервіси.

Як підсумок можна визначити наступне: мікросервісна архітектура не є панацея, але вирішує широке коло проблем, які не міг вирішити моноліт.

Мікросервіси є більш гнучкий спосіб представлення програмного забезпечення, але мікросервіси не слід використовувати без аналізу предметної області. Мікросервісна архітектура стає дедалі все популярніше, багато провідних компаній використовують мікросервіси для розробки своїх продуктів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Хаханов В.І. Кіберсоціальная система – розумний кіберуніверситет / В.І. Хаханов, Е.І. Литвінова, С.В. Чумаченко, А.С. Мищенко // Радіоелектронні і комп'ютерні системи. – 2016. – No 5 (79). – С.187-194.

2. Розумний дім [Електронний ресурс] Режим доступа: https://uk.wikipedia.org/wiki/%D0%A0%D0%BE%D0%B7%D1%83%D0%BC%D0%BD%D0%B8%D0%B9_%D0%B4%D1%96%D0%BC

3. Уінг Ч. Как работает ваш дом [Текст] / Ч. Уінг – К. : ДМК-Пресс, 2016. – 206 с.

4. Пєтін В. В. Создание умного дома на базе Arduino [Текст] / В. В. Пєтін – К. : ДМК-Пресс, 2017. – 180 с.

5. Умный Дом MiMi SMART. Що вміє Розумний будинок [Електронний ресурс] Режим доступа: https://www.smarthouse.ua/ua/chto_umeet_umnyj_dom.html

6. Блум Д. Изучаем Arduino. Инструменты и методы технического волшебства [Текст] / Д. Блум – К. : БХВ-Петербург, 2021. – 544 с.

7. Пєтін В. А. Практическа энциклопедия Arduino [Текст] / В. А. Петин, Біняковский А. А. – К. : ДМК Пресс, 2020. – 166 с.

8. Сардін І. Проблеми функціонування бездротових пристроїв Bluetooth та IEEE 802.11 неліцензійному діапозоні ISM 2,4 ГГц та шляхи їх вирішення, 2006

9. Introduction to microservices. – Режим доступу: https://nginx.com/blog/introduction-to-microservices/ – Дата доступу: 29.05.2019

10. Using an API Gateway. – Режим доступу: https://nginx.com/blog/buildingmicroservices-using-an-api-gateway/ – Дата доступу: 27.05.2019

11. Service Discovery. – Режим доступу: https://nginx.com/blog/service-discovery-ina-microservices-architecture/ – Дата доступу: 27.05.2019

12. Офіційний сайт Docker. – Режим доступу: https://docker.com/ – Дата доступу: 27.05.2019

13. Офіційний сайт C++ Micro Services. – Режим доступу: http://cppmicroservices.org/ – Дата доступу: 29.05.2017

14. Офіційний сайт Pistache framework. – Режим доступу: http://pistache.io/ – Дата доступу: 29.05.2019

15. Офіційний сайт Spring framework. – Режим доступу: https://spring.io – Дата доступу: 29.05.2019

16. Офіційний сайт Spark framework. – Режим доступу: https://sparkjava.com/ – Дата доступу: 29.05.2019