МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ВЕБЗАСТОСУНКУ ВІЗУАЛІЗАЦІЇ ДАНИХ ЗАСОБАМИ ANGULARJS ТА NODE.JS»**

Виконав: студент 4 курсу, групи 6.1219-2пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
Д.В. Мельник
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.ф.-м.н. Кудін О.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Матвіїшина Н.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної
інженерії, к.ф.-м.н., доцент

________________ Лісняк А.О.
(підпис)

“ 07 ” 02 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Мельнику Дмитру Вікторовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка вебзастосунку візуалізації даних засобами AngularJS та Node.js

керівник роботи Кудін Олексій Володимирович, к.ф.-м.н, доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 26 » січня 2023 року № 102-с

2. Строк подання студентом роботи 07.06.2023 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1. Постановка задачі, аналіз предметної області.
   2. Проєктування.
   3. Реалізація та тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) ________
презентація за темою докладу

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.02.2023 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	08.02.2023	
2.	Збір вихідних даних.	17.02.2023	
3.	Обробка методичних та теоретичних	10.03.2023	
	джерел.		
4.	Розробка першого та другого розділу.	14.04.2023	
5.	Розробка третього розділу.	15.05.2023	
6.	Оформлення та нормоконтроль	01.06.2023	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	22.06.2023	

Студент ____________ Д.В. Мельник
(підпис) (ініціали та прізвище)

Керівник роботи ____________ О.В. Кудін
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер ____________ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка вебзастосунку візуалізації даних засобами AngularJS та Node.js»: 56 с., 22 рис., 10 джерел, 4 додатки.

ANGULAR, API, CHARTJS, FRAMEWORK, MVC, NODEJS, TYPESCRIPT, UML.

Об'єкт дослідження – вебзастосунок, інструменти для взаємодії Angular та Node.js, засоби взаємодії вебзастосуноку з користувачем.

Мета роботи – розробити вебзастосунок візуалізації даних.

Методи дослідження – моделювання, проєктування, програмний, аналітичний.

В сучасному цифровому світі об'єм і складність даних зростають експоненційно. Великі компанії, дослідницькі установи, урядові органи та інші організації зіштовхуються з необхідністю аналізувати та інтерпретувати величезні масиви даних для прийняття обґрунтованих рішень. Саме тут важливу роль відіграє візуалізація даних.

Одним з основних аспектів актуальності візуалізації даних є здатність швидко розуміти та виявляти закономірності, тенденції та залежності. Людський мозок легше сприймає інформацію, яку можна відобразити в графіках, діаграмах, картах тощо. Візуальні зображення допомагають знайти схожість, розрізнити аномалії та зробити швидкий аналіз даних, що може бути особливо важливим у ситуаціях, коли необхідно швидко прийняти рішення.

Таким чином, за результатами роботи створено зручний та ефективний вебзастосунок візуалізації даних з використанням Angular та Node.js.

SUMMARY

Bachelor's qualifying paper «Development of a Web Application for the Data Visualization using AngularJS and Node.js»: 56 pages, 22 figures, 10 references, 4 supplements.

ANGULAR, API, CHARTJS, FRAMEWORK, MVC, NODEJS, TYPESCRIPT, UML.

The object of the study is the web application, tools for Angular and Node.js interaction, means of user interaction with the web application.

The aim of the study is to develop a data visualization web application.

The methods of research are modeling, design, programming, analytical.

In the modern digital world, the volume and complexity of data are growing exponentially. Large companies, research institutions, government bodies, and other organizations are faced with the need to analyze and interpret vast arrays of data to make informed decisions. This is where data visualization plays a crucial role.

One of the key aspects of the relevance of data visualization is its ability to quickly understand and identify patterns, trends, and dependencies. The human brain better perceives information that can be presented in graphs, charts, maps, and other visual representations. Visual images help find similarities, distinguish anomalies, and conduct rapid data analysis, which can be particularly important in situations where quick decision-making is required.

Thus, as a result of the study, a convenient and efficient data visualization web application has been developed using Angular and Node.js.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Технічне завдання 10
1.1 Терміни та визначення 10
1.1.1 Загальні терміни 10
1.1.2 Технічні терміни 10
1.2 Функціональні вимоги 11
1.2.1 Призначення і цілі створення системи 11
1.2.2 Загальні функціональні можливості системи 11
1.3 Нефункціональні вимоги 11
1.4 Опис предметної області 12
1.5 Опис системи 12
1.6 Огляд інструментів розробки 13
1.6.1 Angular 13
1.6.2 NodeJS 15
1.6.3 ChartJS 16
2 Проєктування 17
2.1 Використання UML під час розробки системи 17
2.2 Діаграма варіантів використання 18
2.2.1 Опис варіантів використання 20
2.3 Діаграма діяльності 24
2.4 Діаграма послідовності 27
2.5 Діаграма розгортання 30
3 Реалізація та тестування 32
3.1 Опис інструментів розробки 32

3.2 Angular-компонент 32
3.3 Angular-сервіс 33
3.4 Node.js-модель 35
3.5 Node.js-контролер 35
3.6 Тестування проєкту 36
3.6.1 Unit-тест 36
3.6.2 Integration-тест 38
3.7 Керівництво користувача 39
3.7.1 Рівень підготовки користувача 39
3.7.2 Підготовка до роботи 39
3.7.3 Керування джерелом вхідних даних 39
3.7.4 Керування діаграмами 40
3.7.5 Візуалізація відображення 42
3.7.6 Фільтрація вхідних даних 44
3.7.7 Експорт даних до pdf 45
Висновки 47
Перелік посилань 48
Додаток А Angular-компонент 49
Додаток Б Angular-сервіс 53
Додаток В Node.js-модель 55
Додаток Г Node.js-контролер 56

ВСТУП

В сучасному цифровому світі об’єм і складність даних зростають експоненційно. Великі компанії, дослідницькі установи, урядові органи та інші організації зіштовхуються з необхідністю аналізувати та інтерпретувати величезні масиви даних для прийняття обґрунтованих рішень. Саме тут важливу роль відіграє візуалізація даних.

Одним з основних аспектів актуальності візуалізації даних є здатність швидко розуміти та виявляти закономірності, тенденції та залежності. Людський мозок легше сприймає інформацію, яку можна відобразити в графіках, діаграмах, картах тощо. Візуальні зображення допомагають знайти схожість, розрізнити аномалії та зробити швидкий аналіз даних, що може бути особливо важливим у ситуаціях, коли необхідно швидко прийняти рішення.

Другим аспектом важливості візуалізації даних є її здатність до комунікації інформації. Люди краще розуміють та запам’ятовують історії, які розповідаються через візуальні елементи. Візуалізація даних дозволяє перетворити нудну табличну інформацію на вражаючі графіки, які здатні зацікавити та залучити аудиторію. Вона створює ефективний засіб комунікації між аналітиками, менеджерами та іншими зацікавленими сторонами, що сприяє кращому розумінню і спільному прийняттю рішень.

Виходячи з цього, було вирішено створити вебзастосунок, котрий би мав простий інтерфейс та був доступний всім бажаючим.

Актуальність дослідження: актуальність теми зумовлена необхідністю візуалізації даних у сферах бізнесу, науки, політики та багатьох інших галузях, де аналіз даних відіграє важливу роль у прийнятті обґрунтованих рішень.

З огляду на це, можна виділити наступні цілі і задачі нашого дослідження:

Мета: розробити вебзастосунок візуалізації даних.

Задачі:

1) сформувати вимоги до системи;

2) спроєктувати та побудувати архітектуру системи;

3) реалізувати вебзастосунок візуалізації даних;

4) протестувати роботу системи.

Об'єкт дослідження: процес візуалізації даних користувачем, інструменти для реалізації вебзастосунку візуалізації даних, функціонал вебзастосунку, необхідний користувачам.

Предмет дослідження: створення вебзастосунку, що дозволяє візуалізувати дані.

Методи дослідження: моделювання, проєктування, програмний, аналітичний.

Перший розділ присвячено збору та аналізуванню вимог до системи, огляду інструментів розробки та опису вебзастосунку.

У другому розділі розглянуто етапи проєктування вебзастосунку, наведено детальний опис прецедентів.

Третій розділ присвячено реалізації та тестуванню роботи вебзастосунку, наведено детальне керівництво користувача, яке описує процес роботи з вебзастосунком візуалізації даних.

1 ТЕХНІЧНЕ ЗАВДАННЯ

1.1 Терміни та визначення

1.1.1 Загальні терміни

Система – вебзастосунок візуалізації даних створений на основі Angular та NodeJS.

Angular – це фреймворк для створення веб-додатків, який базується на TypeScript. Angular дозволяє створювати потужні та масштабовані додатки з використанням компонентної архітектури та забезпечує розробникам ряд інструментів для зручної роботи.

NodeJS – це відкрита серверна платформа для виконання JavaScript коду на стороні сервера.

ДВІ – Діаграма Варіантів Використання чи Use Case Diagram.

ДД – Діаграма Діяльності.

ДП – Діаграма Послідовності.

Користувач – людина, котра перейшла на сторінку веб-застосунку.

1.1.2 Технічні терміни

API – набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

.env – файл, який дозволяє зберігати та налаштовувати змінні середовища.

1.2 Функціональні вимоги

1.2.1 Призначення і цілі створення системи

Функціональне призначення системи – реалізувати можливість візуалізації даних.

Експлуатаційне призначення системи: система може експлуатуватися користувачем системи.

Мета створення системи – розробка системи візуалізації даних.

1.2.2 Загальні функціональні можливості системи

Система має надавати користувачам такі можливості:

- обирати/змінювати спосіб завантаження вхідних даних;
- обирати/змінювати тип діаграми для відображення;
- фільтрувати вхідні дані;
- експортувати результат до pdf.

1.3 Нефункціональні вимоги

Інтерфейс користувача. Система повинна відображати коректно інтерфейс користувача на будь-якому пристрої.

Підтримка браузерів. Система повинна працювати для наступних браузерів останніх версій: Mozilla Firefox, Google Chrome, Safari, Microsoft Edge, Opera.

Вимоги до продуктивності. Система повинна відображати будь-яку сторінку не довше, ніж за 1 секунду.

Система повинна відправляти повідомлення не довше, ніж 2 секунди.

Вимоги до безпеки. Система не повинна дозволяти вводити у поля дані, які можуть використовуватися, як експлойти.

Система не повинна відображати API-ключі та іншу конфедеційну інформацію, пов'язану з підключенням до зовнішніх сервісів.

1.4 Опис предметної області

Предметною областю є розробка системи для візуалізації чисельних або гібритних даних. Дана система повинна надавати користувачам можливість обирати дані та діаграми для їх візуалізації. Процес взаємодії з системою проходить наступним чином. Користувач, повинен ввести адресу сайту в браузері.

Після входу до системи користувач має можливість взаємодіяти з такими розділами як: джерело вхідних даних, вид діаграми для відображення, фільтрація вхідних даних, експорт результату до pdf.

Процесс побудови візуалізації даних проходить наступним чином. Користувач обирає джерело даних з запропанованих або завантажує самостійно, керуючись шаблоном. Система відображує дані, використовуючи діаграму за замовчуванням. Після генерації діаграми, користувач має можливість змінити її тип та/або отфільтрувати вхідні дані.
Також користувач має можливість експортувати результат відображення вхідних даних до pdf.

1.5 Опис системи

Візуалізація даних є важливим інструментом для аналізу та розуміння великих обсягів інформації. Це процес представлення даних у графічному вигляді з метою забезпечення зручності їх сприйняття та аналізу.

Одним з головних принципів візуалізації даних є можливість швидкого виявлення залежностей, трендів та паттернів в даних, які можуть бути невидимі при перегляді у табличному вигляді. Крім того, візуалізація даних може допомогти виявити потенційні проблеми, такі як аномалії в даних або недостатність даних для прийняття важливих рішень.

Візуалізація даних стала особливо актуальною в останні роки зі зростанням кількості даних, які генеруються у різних галузях, таких як наука, медицина, фінанси та бізнес. Інтерактивні візуалізації даних дозволяють користувачам самостійно вибирати параметри для відображення та взаємодіяти з даними у режимі реального часу, що дозволяє зробити аналіз більш ефективним та точним.

Нарешті, візуалізація даних допомагає покращити комунікацію результатів аналізу даних. Використання візуальних елементів, таких як діаграми та графіки, дозволяє легко інтерпретувати та поділитися результатами з іншими користувачами.

Зважаючи на актуальність таких систем, була розроблена нова система з зручним та зрозумілим для користувача інтерфейсом.

Можливості системи:

- вибір джерела вхідних даних;
- вибір діаграми для візуалізації вхідних даних;
- експорт до pdf.

1.6 Огляд інструментів розробки

1.6.1 Angular

Angular є одним з найпопулярніших веб-фреймворків, який використовується для розробки веб-додатків з високою ефективністю та взаємодією. Він створений на мові програмування TypeScript та розроблений компанією Google.

Angular є повністю клієнтським фреймворком, який дозволяє розробникам створювати SPA (Single Page Applications). Angular забезпечує розробникам широкий спектр можливостей, таких як шаблони, директиви, сервіси, інжектори та інші.

Шаблони дозволяють визначати структуру веб-сторінок та компонентів, які вони містять. Директиви дозволяють змінювати поведінку стандартних HTML-елементів та додавати нові. Сервіси використовуються для взаємодії з сервером та збереження даних на клієнтській стороні. Інжектори використовуються для внедрення залежностей сервісів та компонентів в інші компоненти.

Angular дозволяє використовувати спеціальні декоратори, такі як @Component та @NgModule, для створення та організації компонентів та модулів веб-додатків. Angular також підтримує реактивне програмування з використанням RxJS, що дозволяє забезпечити ефективну взаємодію зі станом додатку.

Для розробки Angular додатків необхідно використовувати спеціальний інструментарій, який включає в себе Angular CLI, який дозволяє створювати, налаштовувати та запускати проєкти. Angular також підтримує побудову додатків з використанням серверної рендерінгу, що забезпечує ефективну оптимізацію веб-сторінок та пок ращення їх продуктивності.

Angular має велику спільноту розробників, яка забезпечує підтримку та надає безліч готових компонентів та бібліотек. Також, Angular є досить документованим фреймворком, що сприяє швидкому вивченню та розробці.

Одним з головних переваг Angular є його модульність та масштабованість. Завдяки використанню модулів та компонентів, Angular дозволяє розробляти великі та складні додатки з багатьма функціями та сторінками. Крім того, Angular дозволяє легко розширювати функціональність додатків за допомогою сторонніх бібліотек та модулів.

1.6.2 NodeJS

Node.js – це відкрите середовище для виконання JavaScript на серверній стороні, що дозволяє розробникам створювати високопродуктивні та масштабовані веб-додатки з використанням технологій, які використовуються на клієнтській стороні. Node.js базується на двигуні V8, який створений компанією Google та використовується в браузері Google Chrome.

Node.js дозволяє розробникам створювати веб-додатки з використанням однієї мови програмування - JavaScript, яка дозволяє легко переносити код між клієнтською та серверною сторонами. Він також дозволяє розробникам зосередитись на створенні високопродуктивного та ефективного коду з використанням асинхронних функцій та інших технік програмування.

Однією з головних переваг Node.js є його здатність до швидкого відповідання на запити та обробки багатьох одночасних підключень. Це досягається за допомогою моделі подій та неблокуючих операцій вводу/виводу. Node.js також дозволяє використовувати пакетний менеджер npm, що надає доступ до більш ніж 1,5 мільйонів пакетів, що розширюють можливості розробки.

Node.js використовується в різних областях, таких як розробка веб-додатків, розробка мобільних додатків, розробка інтернет-розсилок, створення музичних та відео стрімінгових сервісів, розробка інтернет-магазинів та багато іншого.

Для розробки Node.js додатків необхідно мати знання JavaScript та знати основні принципи веб-розробки. Також необхідно мати знання пакетного менед жера npm та фреймворків, таких як Express.js та Socket.io, які спрощують процес розробки та дозволяють швидше створювати високоякісні додатки. Для візуалізації даних та створення інтерактивних інтерфейсів можна використовувати бібліотеки, такі як D3.js та React.

Node.js підтримує роботу з різними базами даних, такими як MySQL, MongoDB, PostgreSQL та інші, що дозволяє легко інтегрувати додатки з різними

базами даних.

Окрім того, Node.js дозволяє розробникам використовувати JavaScript для створення розширень для інших додатків та браузерних плагінів, таких як Chrome Extension та Firefox Add-ons.

Node.js є популярним та широко використовуваним середовищем для розробки веб-додатків та має великий потенціал для розширення функціональності та забезпечення високої продуктивності.

1.6.3 ChartJS

Chart.js – це JavaScript-бібліотека для створення красивих та інтерактивних графіків і діаграм. Вона дозволяє швидко та просто відтворювати різні типи діаграм, такі як лінійні, стовпчикові, кругові та інші, з наданням великої кількості кастомізацій та налаштувань.

Однією з головних переваг Chart.js є його легкість використання та налаштування. Вона має простий та зрозумілий API, що дозволяє швидко створювати графіки та діаграми з мінімальними зусиллями. Також бібліотека має широкий спектр налаштувань та можливостей для кастомізації графіків, таких як зміна кольорів, шрифтів, легенд, анімації та багато іншого.

Chart.js підтримує різні типи діаграм, такі як:

– лінійні діаграми, які використовуються для відображення змін у часі;
– стовпчикові діаграми, які використовуються для порівняння значень між різними категоріями;
– кругові діаграми, які використовуються для показу часток у загальній сумі;
– діаграми розкиду, які використовуються для відображення зв’язку між двома змінними;
– гістограми, які використовуються для відображення розподілу даних.

Бібліотека Chart.js є відкритою та безкоштовною. Вона має велику та активну спільноту, що дозволяє швидко знайти відповіді на будь-які запитання щодо використання бібліотеки.

2 ПРОЄКТУВАННЯ

2.1 Використання UML під час розробки системи

UML (Unified Modeling Language) є мовою моделювання, яка широко використовується під час розробки програмного забезпечення для візуалізації, специфікації, конструювання та документування системи.

UML надає набір графічних символів і правил, які дозволяють розробникам описати структуру та поведінку системи.

Розглянемо основні аспекти використання UML під час розробки системи. *Визначення вимог*: UML дозволяє аналізувати та моделювати вимоги до системи шляхом використання діаграм вимог. Це дозволяє комунікувати зі зацікавленими сторонами та уточнювати вимоги перед розробкою системи.

Проєктування структури: UML надає діаграми класів, діаграми компонентів та діаграми пакетів для моделювання структури системи. Ці діаграми допомагають визначити класи, їх взаємозв'язки та інтерфейси між компонентами системи.

Моделювання поведінки: UML надає діаграми послідовності, діаграми станів та діаграми діяльності для моделювання поведінки системи. Ці діаграми дозволяють показати взаємодію об'єктів, послідовність операцій та різні стани, в яких може перебувати система.

Взаємодія з користувачем: UML дозволяє моделювати інтерфейс користувача за допомогою діаграми взаємодії, діаграми активності та інших діаграм. Це допомагає розробникам розуміти та візуалізувати взаємодію користувача з системою.

Архітектурне проєктування: UML може бути використана для моделювання архітектури системи, включаючи діаграм компонентів, діаграми розгортання та діаграми пакетів. Ці діаграми допомагають визначити структуру системи, розміщення компонентів, залежності та взаємодію між ними.

Тестування та валідація: UML може бути використана для моделювання сценаріїв тестування та валідації системи за допомогою діаграм послідовності, діаграм станів та інших діаграм. Це дозволяє розробникам зрозуміти та перевірити правильність роботи системи перед реалізацією.

Документація: UML надає стандартизований спосіб документування системи. Різні діаграми UML можуть бути використані для створення технічної документації, яка описує структуру, поведінку та інші аспекти системи. Це полегшує розуміння та спілкування між розробниками, аналітиками та іншими учасниками проєкту.

Загалом, використання UML під час розробки системи допомагає візуалізувати та формалізувати різні аспекти проєкту, сприяє зрозумінню та спілкуванню між розробниками та зацікавленими сторонами, а також полегшує аналіз, проєктування, тестування та документування системи.

2.2 Діаграма варіантів використання

Діаграма варіантів використання (Use Case Diagram) є одним з основних видів діаграм UML і використовується для моделювання функціональності системи з точки зору її взаємодії з акторами (користувачами або зовнішніми системами). Діаграма варіантів використання допомагає визначити та відобразити основні дії або «варіанти використання», які можуть бути виконані акторами в системі.

Розглянемо основні елементи діаграми варіантів використання.

Актори: представляють зовнішніх користувачі або системи, які взаємодіють з системою, тобто особи або ролі, які мають інтерес до функціональності системи. На діаграмі вони зображаються у вигляді піктограм людини або блоку з назвою.

Варіанти використання: описують конкретні дії або сценарії, які актори можуть виконати в системі. Кожен варіант використання має назву, яка описує

його основну функціональність, та може мати асоційовані акторів, пре- та постумови. На діаграмі вони зображаються у вигляді овалів з назвами.

Взаємодії: показують зв'язки між акторами та варіантами використання. Вони вказують, які актори беруть участь у виконанні конкретного варіанту використання. Зв'язки можуть бути прямими (актор безпосередньо взаємодіє з варіантом використання) або індиректними (через інші актори або системи).

Розширення та умови: деякі варіанти використання можуть мати варіанти розширення (Extension Points) або умови (Conditions), що розширюють або обмежують основний сценарій використання. Вони використовуються для моделювання альтернативних шляхів виконання або умов, за яких варіант використання може бути активований або зупинений.

Діаграма варіантів використання дозволяє візуалізувати основні функціональність системи та взаємодію акторів з нею. Вона допомагає уточнити вимоги до системи, ідентифікувати основні варіанти використання та зв'язки між ними, а також слугує основою для подальшого аналізу, проєктування та реалізації системи.

При створенні діаграми варіантів використання важливо враховувати реальні потреби та цілі користувачів, а також уникати деталей реалізації системи. Вона служить як комунікаційний засіб між розробниками, аналітиками та зацікавленими сторонами для спільного розуміння функціональності системи.

На рисунку 2.1 представлена діаграма варіантів використання.

На діаграмі представлено актора «Користувач», який відображає функціональні можливості користувачів системи.

Виділено 1 основний варіант використання – «Візуалізація даних». Після того, як користувач переходить на сайт системи візуалізації даних, запускаються прецеденти «Ініціалізація вхідних даних» і «Ініціалізація типу діаграми», які встановлюють конфігурацію за замовчуванням та надають змогу відразу отримати відображення даних. Якщо користувачу потрібен інший набор даних або вид діаграми, то він змінює їх, обравши зі списку відповідних полів. Після того, як користувач обрав дані та вид діаграми, він натискає на кнопку

«Візуалізація даних», система відправляє запит до відповідного сервісу та візуалізує отримані дані.

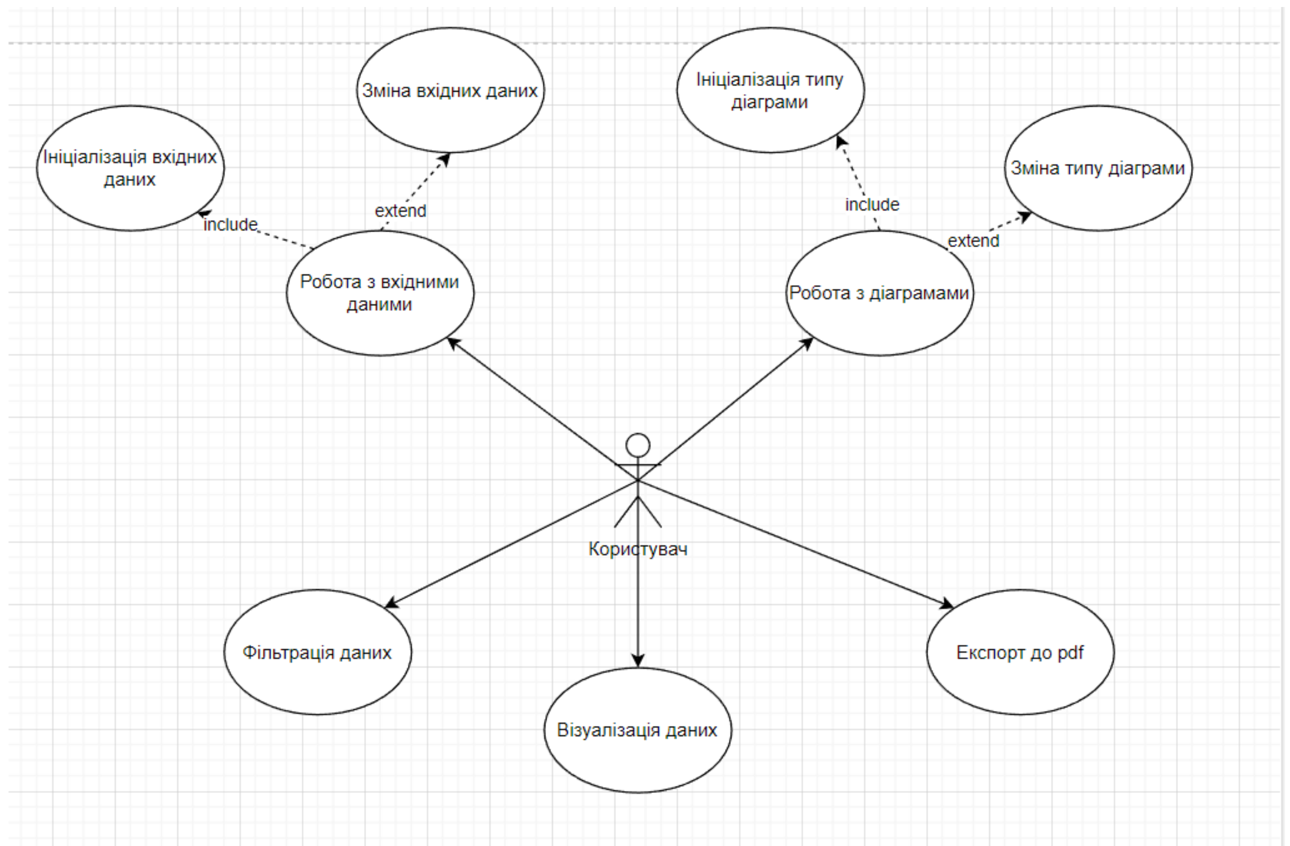

Рисунок 2.1 – Діаграма варіантів використання

Варіанти використання визначають функціональні можливості. Кожен з них представляє певний спосіб використання. Таким чином, кожен варіант використання відповідає послідовності дій для того, щоб користувач міг отримати певний результат.

2.2.1 Опис варіантів використання

Прецедент «Робота з вхідними даними»

Призначення: даний варіант використання надає можливість користувачу взаємодіяти з вхідними даними.

Основний потік подій: даний варіант використання починає виконуватися, коли користувачу потрібно обрати вхідні дані для візуалізації. Система пропонує перелік джерел даних. Після того, як користувач обрав джерело, система розблоковує можливість вибору виду діаграми, а також динамічно змінює перелік фільтрів.

Альтернативний потік: користувач не обрав джерело вхідних даних – система сповіщає про це користувача. Користувач може спробувати ще раз обрати дані.

Передумова: перед початком виконання даного варіанта використання користувач повинен знаходитися на вебсторінці системи.

Прецедент «Робота з діаграмами»

Призначення: даний варіант використання надає можливість користувачу взаємодіяти з діаграмами.

Основний потік подій: даний варіант використання починає виконуватися, коли користувачу потрібно обрати тип діаграми для візуалізації. Система пропонує перелік типів діаграм. Після того, як користувач обрав тип діаграми, система розблоковує кнопку «Візуалізація даних.

Альтернативний потік: користувач не обрав тип діаграми – система сповіщає про це користувача. Користувач може спробувати обрати ще раз.

Передумова: перед початком виконання даного варіанта використання користувач повинен знаходитися на вебсторінці системи та обрати джерело вхідних даних.

Прецедент «Ініціалізація вхідних даних»

Призначення: даний варіант використання надає можливість користувачу відразу приступити до візуалізації даних, використовуючи джерело даних за замовчуванням.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач переходить до вебсторінки системи. Система обирає набір вхідних даних за замовчуванням та відображає це у відповідному полі.

Передумова: перед початком виконання даного варіанта використання

користувач повинен перейти на вебсторінку системи.

Прецедент «Ініціалізація типу діаграми»

Призначення: даний варіант використання надає можливість користувачу відразу приступити до візуалізації даних, використовуючи тип діаграми за замовчуванням.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач переходить до вебсторінки системи. Система обирає тип діаграми за замовчуванням та відображає це у відповідному полі.

Передумова: перед початком виконання даного варіанта використання користувач повинен перейти на вебсторінку системи.

Прецедент «Зміна вхідних даних»

Призначення: даний варіант використання надає можливість користувачу змінити джерело вхідних даних.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач системи натискає на поле вибору джерела вхідних даних, система відображає перелік доступних джерел даних. Користувач обирає нове джерело та натискає на його назву в переліку, система зберігає нове джерело даних.

Альтернативний потік подій: якщо користувач не обрав нове джерело даних, то система не повинна реагувати на цю дію.

Передумова: перед початком виконання даного варіанта використання користувач повинен перейти на вебсторінку системи.

Прецедент «Зміна типу діаграми»

Призначення: даний варіант використання надає можливість користувачу змінити тип діаграми.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач системи натискає на поле вибору типу діаграми, система відображає перелік доступних типів. Користувач обирає новий тип та натискає на його назву в переліку, система зберігає новий тип.

Альтернативний потік подій: якщо користувач не обрав новий тип

діаграми, то система не повинна реагувати на цю дію.

Передумова: перед початком виконання даного варіанта використання користувач повинен перейти на вебсторінку системи.

Прецедент «Візуалізація даних»

Призначення: даний варіант використання надає можливість візуалізувати дані на основі обраних джерела вхідних даних та типу діаграми.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає на кнопку «Візуалізація даних», система відправляє запит до відповідного сервісу та візуалізує отримані дані.

Передумова: перед початком виконання даного варіанта використання користувач повинен обрати джерело вхідних даних та тип діаграми.

Виняткова ситуація 1: не обрано джерело вхідних даних – система відобразить відповідне повідомлення. Користувач може обрати джерело.

Виняткова ситуація 2: не обрано тип діаграми – система відобразить відповідне повідомлення. Користувач може обрати тип діаграми.

Виняткова ситуація 3: сервіс джерела вхідних даних не відповідає – система відобразить відповідне повідомлення. Користувач може змінити джерело вхідних даних.

Прецедент «Фільтрація даних»

Призначення: даний варіант використання надає можливість користувачу фільтрувати вхідні дані.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач системи обирає фільтр та вводить у відповідне поле значення для фільтрації, система фільтрує вхідні дані та змінює відображення.

Передумова: перед початком виконання даного варіанта використання користувач повинен натиснути на кнопку «Візуалізація даних».

Виняткова ситуація 1: введено невалідні дані – система відобразить відповідне повідомлення. Користувач може змінити дані.

Виняткова ситуація 2: не натиснуто кнопку «Візуалізація даних» – система блокує можливість взаємодіяти з фільтрами.

Прецедент «Експорт до pdf»

Призначення: даний варіант використання надає можливість користувачу експортувати результат візуалізації даних у формат pdf.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач системи натискає на кнопку «Експортувати до pdf». Система завантажує поточне відображення на пристрій користувача.

Передумова: перед початком виконання даного варіанта використання користувач повинен натиснути на кнопку «Відобразити дані».

Виняткова ситуація 1: не натиснуто кнопку «Візуалізація даних» – система блокує кнопку «Експортувати до pdf».

2.3 Діаграма діяльності

Діаграма діяльності (Activity Diagram) є одним з основних видів діаграм UML і використовується для моделювання послідовності дій або процесів в системі. Вона дозволяє візуалізувати потік керування, умови, паралельність та інші аспекти виконання дій.

Розглянемо основні елементи діаграми діяльності.

Дії (Actions): дії представляють окремі елементарні кроки або дії, які виконуються в системі. Вони зображаються у вигляді прямокутників з назвами. Прикладами дій можуть бути «Відкрити файл», «Ввести дані», «Обчислити значення» і т.д.

Переходи (Transitions): переходи показують послідовність виконання дій та зв’язки між ними. Вони зображаються у вигляді стрілок і показують напрямок потоку керування. Переходи можуть мати умови або події, що викликають перехід до наступної дії.

Розгалуження (Branching): розгалуження вказує на різні шляхи або умови, які впливають на подальший потік виконання. Вони зображаються у вигляді ромбів і мають умови або події, що визначають шляхи виконання.

Злиття (Merge): злиття показує об'єднання різних шляхів виконання після розгалуження. Вони зображаються у вигляді ромбів з назвами.

Синхронізація (Synchronization): синхронізація вказує на паралельне виконання дій або процесів. Вона зображується у вигляді смуги, що об'єднує різні потоки виконання.

Пул (Swimlane): пул використовується для групування дій або акторів, що виконують д різні дії або процеси. Він представлений у вигляді вертикальних смуг або стовпців з назвами акторів або виконавців.

Контроль (Control Nodes): контрольні вузли використовуються для керування потоком виконання в діаграмі діяльності. Вони включають початковий вузол (Initial Node) для позначення початку виконання, кінцевий вузол (Final Node) для позначення завершення виконання, вузли рішень (Decision Node) для здійснення вибору між альтернативними шляхами, вузли злиття (Merge Node) для об'єднання різних шляхів виконання і багато інших.

Діаграми діяльності дозволяють вам моделювати різні сценарії виконання, включаючи послідовний потік дій, розгалуження, злиття, паралельне виконання та умови. Вони є корисним інструментом для візуалізації процесів в системі та аналізу потоку роботи.

При створенні діаграми діяльності важливо враховувати послідовність дій, умови, які впливають на вибір шляхів виконання, та паралельність, яка може спостерігатися в процесі. Це допомагає зрозуміти поведінку системи і виявити можливі проблеми або неузгодженості.

Використання діаграм діяльності дозволяє команді розробників та зацікавленим сторонам отримати загальне уявлення про взаємодію компонентів системи та послідовність дій в різних сценаріях. Це сприяє кращому розумінню процесу, поліпшує комунікацію між учасниками проєкту та допомагає виявляти й виправляти можливі проблеми у проєкті.

Наведемо діаграму діяльності, що описує модель поведінки варіанта використання «Візуалізація даних». Діаграма представлена на рисунках 2.2 – 2.3.

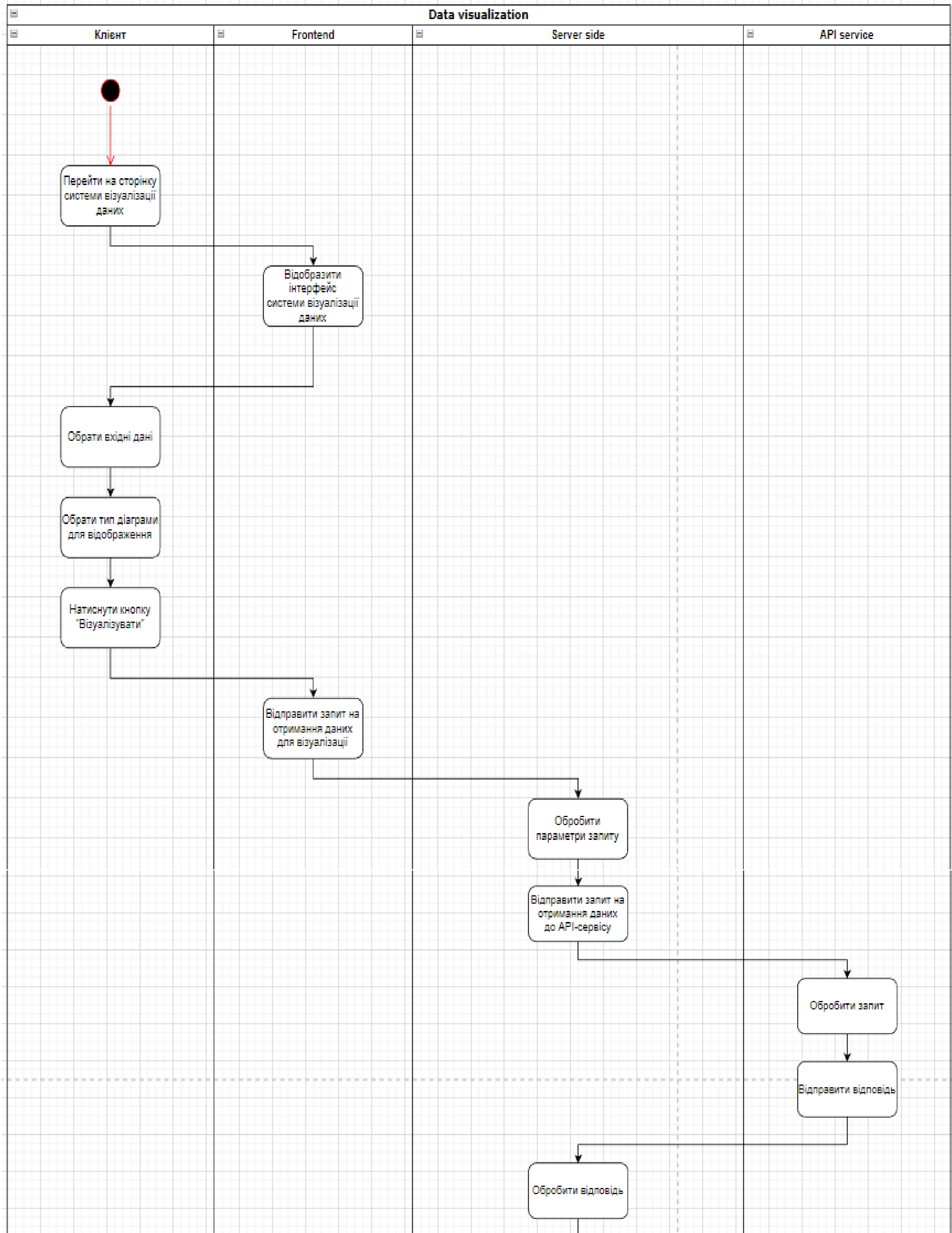

Рисунок 2.2 – Діаграма діяльності

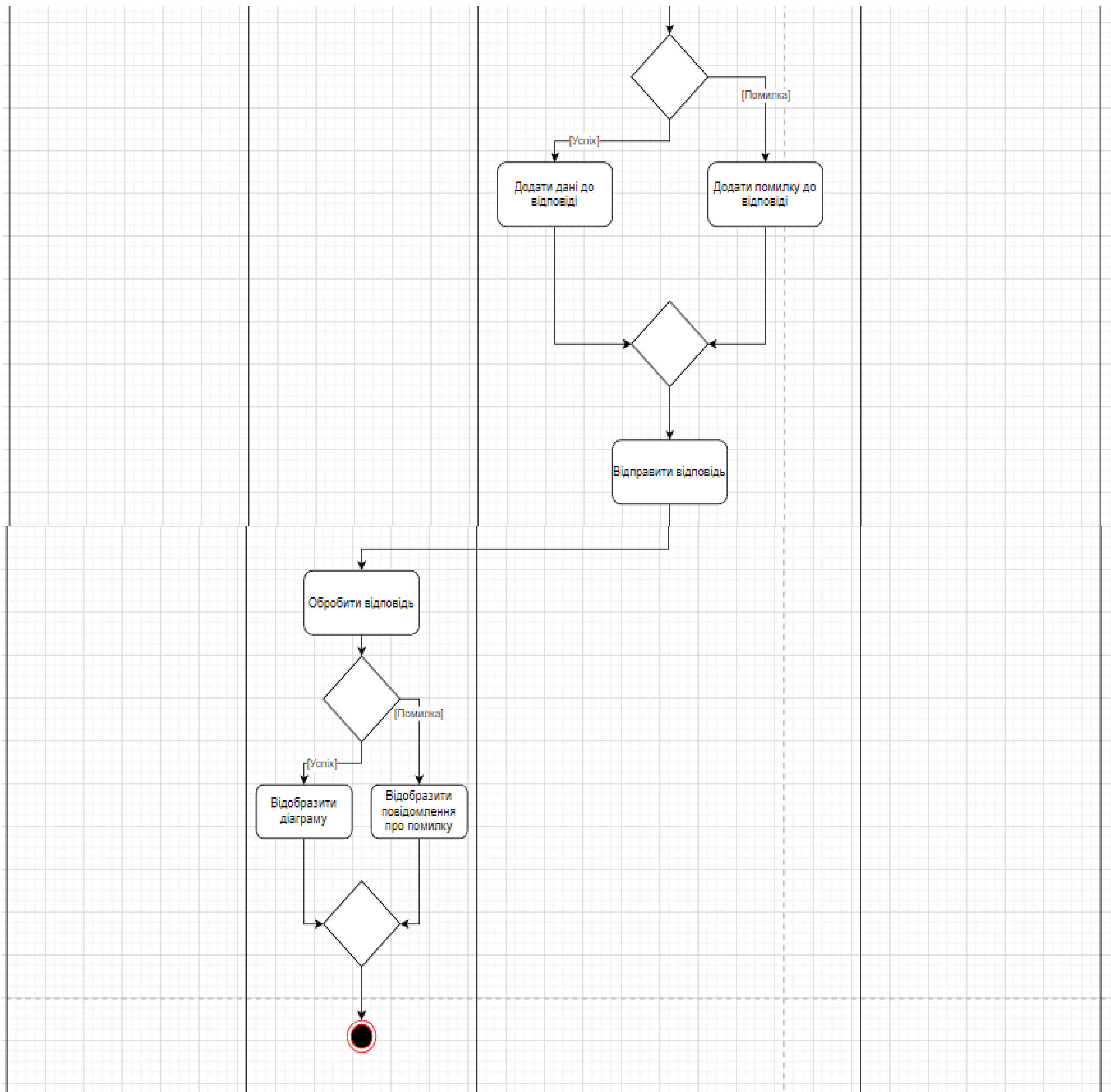

Рисунок 2.3 – Діаграма діяльності

2.4 Діаграма послідовності

Діаграма послідовності (Sequence Diagram) є одним з основних видів діаграм UML і використовується для моделювання взаємодії між об’єктами або компонентами системи в послідовному порядку. Вона дозволяє показати обмін повідомленнями та зміну станів об’єктів протягом певного часового періоду.

На діаграмі послідовності зображаються тільки ті об’єкти, які

безпосередньо беруть участь у взаємодії.

Лінія життя об'єкта зображується пунктирною вертикальною лінією, асоційованою з єдиним об'єктом на діаграмі послідовності. Лінія життя служить для позначення періоду часу, протягом якого об'єкт існує в системі і, отже, може потенційно брати участь у всіх її взаємодіях. Якщо об'єкт існує в системі постійно, то і його лінія життя повинна продовжуватися по всій площині діаграми послідовності від самої верхньої її частини до самої нижньої.

Розглянемо основні елементи діаграми послідовності.

Об'єкти (Objects): об'єкти відображають ролі або елементи системи, які взаємодіють між собою. Вони зображаються у вигляді прямокутників з назвами на верхній стороні. Кожен об'єкт може виконувати певні дії та надсилати повідомлення іншим об'єктам.

Повідомлення (Messages): повідомлення показують обмін інформацією між об'єктами. Вони зображаються у вигляді стрілок, які вказують напрямок передачі повідомлення. Повідомлення можуть бути синхронними (чекають на відповідь) або асинхронними (не чекають на відповідь). Кожне повідомлення має назву, що вказує на дію або метод, який виконується.

Життєвий цикл об'єкта (Object Lifeline): життєвий цикл об'єкта показує часову ось об'єкта та зміни його стану. Він зображується у вигляді вертикальної лінії, яка простягається вздовж діаграми і позначає існування об'єкта протягом часу.

Взаємодія (Interaction): взаємодія показує порядок виконання повідомлень та взаємодію між об'єктами. Вона може включати в себе послідовності, умови, цикли та інші конструкції, які допомагають описати поведінку системи.

Фрагменти (Fragments): фрагменти використовуються для моделювання структури, умов та інших складних сценаріїв в діаграмі послідовності. Вони дозволяють включати умовні вітки, цикли, паралельні виконання та інші структурні елементи.

Ограничення (Constraints): ограничення використовуються для опису додаткових умов або обмежень, які повинні виконуватися під час взаємодії між

об'єктами. Вони можуть бути використані для визначення правил поведінки або обмежень, які повинні бути враховані під час моделювання системи.

Актори (Actors): актори представляють зовнішні сутності, які взаємодіють з системою. Вони можуть бути людьми, іншими системами, апаратурою або будь-якими іншими сутностями, які взаємодіють з системою. Актори зображуються у вигляді піктограм або прямокутників з назвами.

Діаграма послідовності допомагає уточнити взаємодію між об'єктами і компонентами системи, а також відображає часові взаємозв'язки та порядок виконання дій. Вона є потужним інструментом для аналізу та проєктування системи перед реалізацією.

На рисунку 2.4 описана діаграма послідовності прецедента «Візуалізувати дані».

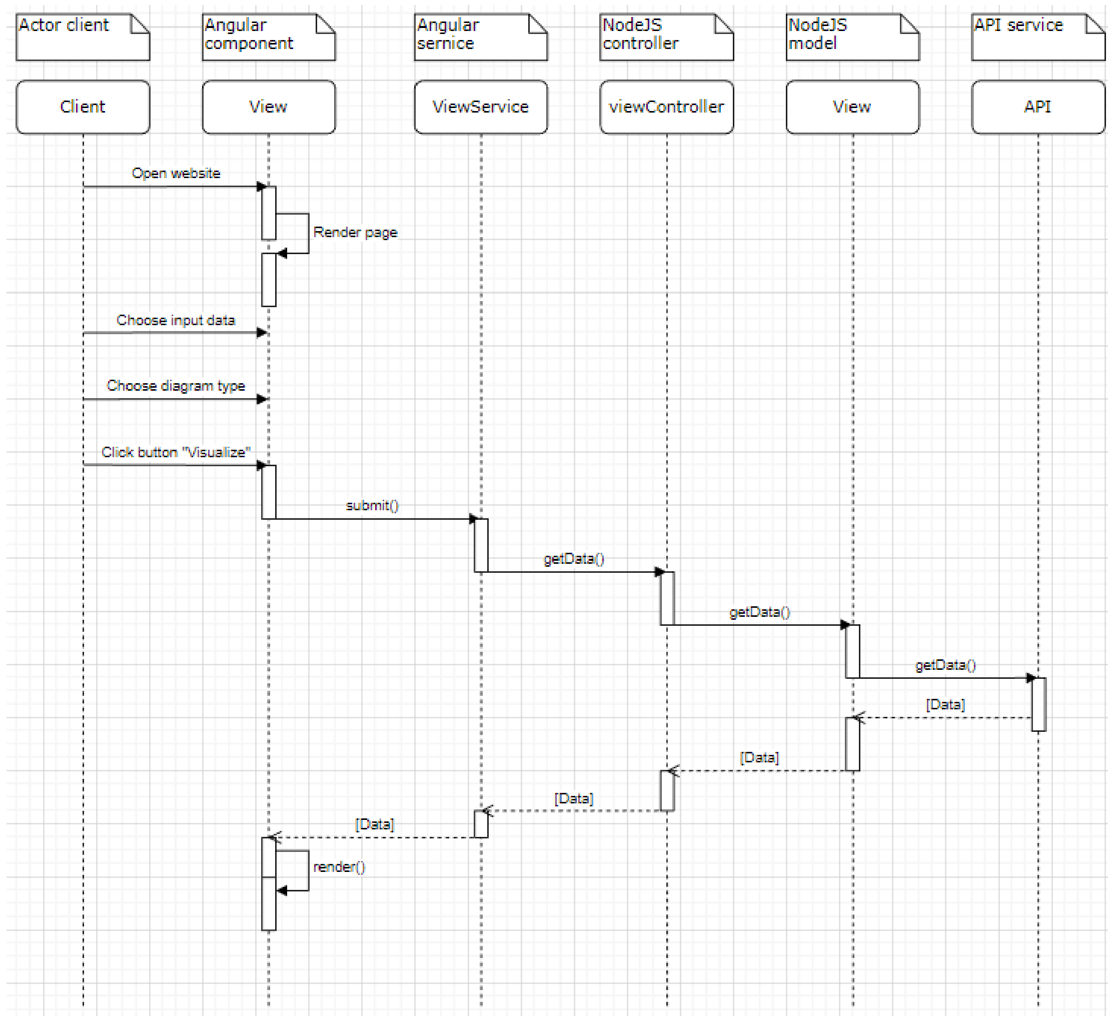

Рисунок 2.4 – Діаграма послідовності

2.5 Діаграма розгортання

Діаграма розгортання (Deployment Diagram) є одним з видів діаграм UML і використовується для моделювання фізичної архітектури системи, її компонентів та залежностей між ними. Вона дозволяє візуалізувати розміщення апаратних та програмних ресурсів системи, таких як сервери, комп'ютери, мережеві вузли, програмні модулі та їх взаємозв'язки.

Розглянемо основні елементи діаграми розгортання.

Вузли (Nodes): вузли представляють фізичні або віртуальні пристрої, на яких розгортується система. Вони можуть бути серверами, комп'ютерами, мобільними пристроями, мережевими вузлами тощо. Кожен вузол зображується у вигляді прямокутника з назвою.

Артефакти (Artifacts): артефакти представляють програмні компоненти або файли, що розгортаються на вузлах. Вони можуть бути виконуваними файлами, бібліотеками, конфігураційними файлами тощо. Артефакти зображуються у вигляді піктограм або прямокутників з назвами.

Зв'язки (Connections): зв'язки показують комунікацію та залежності між вузлами та артефактами. Вони вказують, як артефакти розгортаються на вузлах і як вони спілкуються між собою. Зв'язки зображуються у вигляді ліній або стрілок, які з'єднують вузли та артефакти.

Атрибути вузлів (Node Attributes): атрибути вузлів вказують характеристики або властивості вузлів, такі як обладнання, операційна система, пам'ять, процесор тощо. Вони можуть бути представлені у вигляді списку атрибутів, що прикріплені до вузла.

Залежності (Dependencies): залежності показують, які артефакти використовуються іншими артефактами або вузлами. Вони вказують на наявність залежностей між компонентами системи. Залежності зображуються у вигляді пунктирних ліній зі стрілками, що вказують напрямок залежності.

Діаграма розгортання допомагає розуміти фізичну архітектуру системи та взаємозв'язки між її компонентами. Вона є корисним інструментом для

планування та управління розгортанням програмного забезпечення і інфраструктури системи.

На рисунку 2.5 наведено діаграму розгортання системи візуалізації даних.

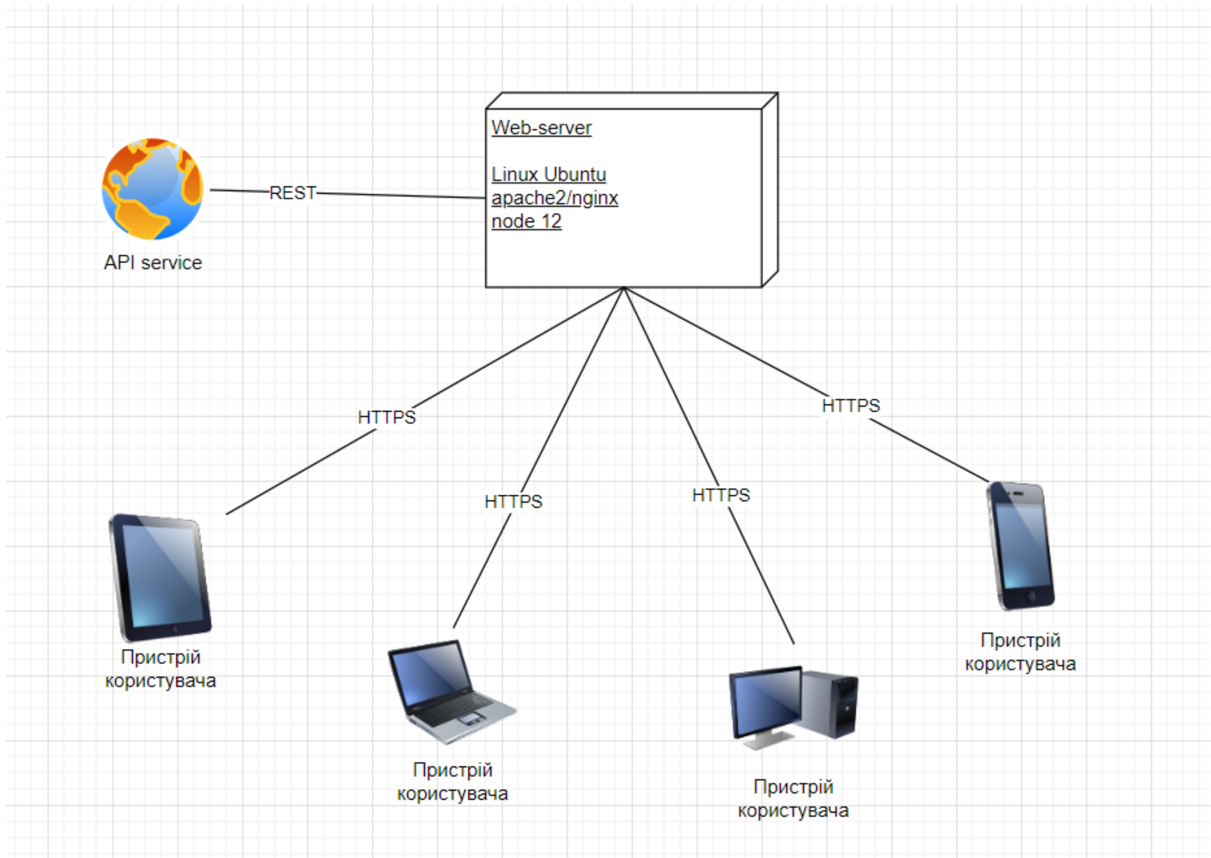

Рисунок 2.5 – Діаграма розгортання

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Опис інструментів розробки

Для реалізації були використані Node.js та front-end фреймворк Angular.

Chart.js – це бібліотека JavaScript для створення інтерактивних та налаштовуваних діаграм і графіків на веб-сторінках. Вона надає простий і гнучкий спосіб візуалізації даних, що допомагає зрозуміти та аналізувати інформацію.

Angular Material – це бібліотека, що включає 30 сучасних компонентів, виготовлених згідно специфікації Google Material Design.

3.2 Angular-компонент

Компонент в Angular відповідає за відображення та поведінку частини користувацького інтерфейсу.

Він включає в себе HTML-шаблон, який визначає структуру та вигляд компонента, а також TypeScript-клас, який відповідає за логіку компонента. Компоненти можуть мати властивості (properties), за допомогою яких дані передаються з батьківського компонента до дочірнього, та можуть генерувати події (events), що сповіщають про зміни або взаємодію з компонентом.

Angular надає потужні механізми для розширення функціональності компонентів, такі як директиви, що додають спеціальні поведінки до компонентів, і сервіси, які надають спільний доступ до даних та функцій між компонентами.

Компоненти в Angular розгортаються в ієрархічній структурі, де батьківські компоненти можуть містити дочірні компоненти. Це дозволяє побудувати складніші інтерфейси шляхом комбінування та повторного

використання компонентів.

Angular-компоненти забезпечують модульну, масштабовану та підтримувану структуру для розробки веб-додатків.

Наведемо приклад компоненту (див. рис. 3.1).

```
export class ViewComponent {

  @ViewChild('pdfContainer') container: ElementRef | undefined;

  form: FormGroup;
  inputData: InputData[] = dataList;
  diagrams: TypeDiagram[] = diagramList;
  selectDiagram: any = this.diagrams[0];

  constructor(
    private viewService: ViewService,
  ) {
    Chart.register(...registerables);
    this.form = new FormGroup({
      data: new FormControl('coin', [Validators.required]),
      diagram: new FormControl('pie', [Validators.required])
    });
  }
}
```

Рисунок 3.1 – Приклад компоненту

Повний код компонента наведено в Додатку А.

3.3 Angular-сервіс

Клас в архітектурі Angular, який надає певну функціональність та може бути використаний для спільного доступу до даних, логіки або зовнішніх ресурсів між компонентами.

Сервіси в Angular використовуються для виконання спільних завдань, таких як отримання та обробка даних, комунікація з сервером, робота з локальним сховищем (локальним сховищем даних браузера, таким як

localStorage), аутентифікація, переклад та багато іншого.

Основна ідея використання сервісів полягає в тому, щоб винести спільну функціональність з компонентів в окремі класи, які можна використовувати в багатьох компонентах. Це сприяє модульності, перевикористанню коду та полегшує тестування.

Сервіси в Angular можуть бути впроваджені за допомогою ін’єкції залежностей. Це означає, що сервіс може бути впроваджений (ін’єктований) в компоненти або інші сервіси, щоб вони могли використовувати його функціональність.

Для створення сервісу в Angular можна використовувати команду ng generate service, а потім відповідний клас сервісу буде створено відповідно до вказаної назви.

Наведемо приклад сервісу (див. рис. 3.2).

```
1   export class ViewService {
2
3     private baseUrl = 'http://localhost:3000/api';
4     headers: any;
5
6     constructor(private http: HttpClient) {
7       this.headers = new HttpHeaders({
8         Accept: 'application/json',
9         'Content-Type': 'application/json',
10        'Access-Control-Allow-Origin': '*',
11        'Access-Control-Allow-Headers': 'Content-Type',
12      'Access-Control-Allow-Methods': 'GET,POST,OPTIONS,DELETE,PUT',
13      });
14    }
15
16    getCoins(): Promise<any> {
17      return this.http
18        .get(this.baseUrl + '/coin', {headers: this.headers})
19        .toPromise()
20        .then((data) => {
21          return data;
22        });
23    }
24  }
```

Рисунок 3.2 – Приклад сервісу

Повний код сервісу наведено в Додатку Б.

3.4 Node.js-модель

Модель відповідає за обробку та управління даними додатку. Це можуть бути взаємодії з базою даних, зберігання та маніпуляція даними, а також бізнес-логіка, пов'язана з цими даними.

Модель може включати схеми, моделі даних та методи для отримання/збереження/оновлення/видалення даних, включаючи правила валідації, взаємодію з базою даних та зовнішніми сервісами.

Наведемо приклад моделі (див. рис. 3.3).

```
1  module.exports = class Coin {
2    static async getData() {
3      return new Promise( (resolve, reject) => {
4        axios
5          .get(URL, options)
6          .then( (response) => {
7            resolve(response.data.data.coins);
8          }).catch((error) => {
9            reject(error);
10         });
11     }).catch(err => console.log(err));
12   }
13
14 };
15
```

Рисунок 3.3 – Приклад моделі

Повний код моделі наведено в Додатку В.

3.5 Node.js-контролер

Відповідає за обробку вхідних запитів та взаємодію з моделлю та поданням. Контролер містить логіку, яка обробляє запити користувача, виконує необхідні операції з моделлю та відправляє дані до відповідного подання для

відображення результату.

Основна задача контролера полягає в тому, щоб приймати запити, зчитувати параметри, викликати відповідні методи моделі для обробки даних та підготовки результату, а потім передавати цей результат до відповідного подання для відображення. Контролер також може відповідати за маршрутизацію, визначення шляхів та керування потоком програми.

Наведемо приклад контролеру (див. рис. 3.4).

```
1  exports.getCoin = async (req, res) => {
2    res.send(await Coin.getData().then(res => res).catch(err => err));
3  };
4
```

Рисунок 3.4 – Приклад контролеру

Повний код контроллеру наведено в Додатку Г.

3.6 Тестування проєкту

3.6.1 Unit-тест

Unit-тестування є процесом перевірки, чи працюють окремі компоненти програмного забезпечення (функції, класи, модулі) вірно і очікувано.

Це важлива частина розробки програмного забезпечення, оскільки добре написані та надійні тести допомагають виявити помилки та сприяють збереженню коректної роботи програми при змінах в коді.

Приклади такого тестування наведено на рисунках 3.5 та 3.6.

```
describe('ViewComponent', () => {
  let component: ViewComponent;
  let fixture: ComponentFixture<ViewComponent>;
  let viewService: ViewService;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ViewComponent],
      imports: [ReactiveFormsModule],
      providers: [ViewService],
    }).compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(ViewComponent);
    component = fixture.componentInstance;
    viewService = TestBed.inject(ViewService);
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should initialize form with default values', () => {
    expect(component.form.get('data')?.value).toEqual('coin');
    expect(component.form.get('diagram')?.value).toEqual('pie');
  });
```

Рисунок 3.5 – Тестування ViewComponent

```
  it('should initialize form with default values', () => {
    expect(component.form.get('data')?.value).toEqual('coin');
    expect(component.form.get('diagram')?.value).toEqual('pie');
  });

  it('should call viewService.getCoins when form is submitted with "coin" data', () => {
    spyOn(viewService, 'getCoins').and.returnValue(
      Promise.resolve([{ name: 'Coin1', price: 10 }, { name: 'Coin2', price: 20 }])
    );
    component.form.patchValue({ data: 'coin' });
    component.onSubmit();
    expect(viewService.getCoins).toHaveBeenCalled();
    expect(component.labels).toEqual(['Coin1', 'Coin2']);
    expect(component.dataAPI).toEqual([10, 20]);
  });

  it('should call viewService.getExchangeRate when form is submitted with "exchange" data', () => {
    spyOn(viewService, 'getExchangeRate').and.returnValue(
      Promise.resolve([{ txt: 'Exchange1', rate: 1 }, { txt: 'Exchange2', rate: 2 }])
    );
    component.form.patchValue({ data: 'exchange' });
    component.onSubmit();
    expect(viewService.getExchangeRate).toHaveBeenCalled();
    expect(component.labels).toEqual(['Exchange1', 'Exchange2']);
    expect(component.dataAPI).toEqual([1, 2]);
  });

  it('should call buildChart method when form is submitted', () => {
    spyOn(component, 'buildChart');
    component.form.patchValue({ data: 'coin' });
    component.onSubmit();
    expect(component.buildChart).toHaveBeenCalled();
  });
});
```

Рисунок 3.6 – Тестування ViewComponent

3.6.2 Integration-тест

Інтеграційне тестування в Angular використовується для перевірки взаємодії між різними компонентами, сервісами, директивами та іншими частинами вашої програми. Це дозволяє перевірити, чи працюють ці компоненти разом належним чином і чи виконуються очікувані результати.

Основою для інтеграційного тестування в Angular є фреймворки для тестування, такі як Jasmine або Jest, разом з допоміжними бібліотеками, такими як @angular/platform-browser та @angular/router/testing. Розглянемо приклад інтеграційного тесту для Angular сервісу (див. рис. 3.7).

```
import { TestBed, inject } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';
import { ViewService } from './view.service';

describe('ViewService', () => {
  let service: ViewService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [ViewService]
    });

    service = TestBed.inject(ViewService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    httpMock.verify();
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('should make a GET request to retrieve coins', () => {
    const mockResponse = [{ name: 'Coin1', price: 10 }, { name: 'Coin2', price: 20 }];

    service.getCoins().then((data) => {
      expect(data).toEqual(mockResponse);
    });

    const req = httpMock.expectOne('http://localhost:3000/api/coin');
    expect(req.request.method).toBe('GET');
```

Рисунок 3.7 – Тестування сервісу

3.7 Керівництво користувача

3.7.1 Рівень підготовки користувача

Користувач сайту повинен володіти певною кваліфікацією.

Навички користувача для роботи з ПК, та навички роботи з web-браузером.

Знайомство з Керівництвом користувача.

3.7.2 Підготовка до роботи

Запуск системи.

Доступ до сайту здійснюється через мережу Інтернет за допомогою звичайного web-браузера. Адреса сайту в мережі Інтернет: https://localhost:4200. Для коректної роботи клієнтської частини повинен використовуватися браузер Google Chrome, Mozilla Firefox, Opera, Safari.

При вході на Сайт користувач потрапляє на сторінку з інструментами для генерації відображення.

На Сайті розрізняються наступні групи користувачів:

Користувач – особа, що має доступ до функцій користувача на сторінці сайту.

3.7.3 Керування джерелом вхідних даних

Після входу на сайт, система відображає форму для вибору параметрів візуалізації даних (див. рис. 3.8). Перший селектор відповідає за джерело вхідних даних. На вибір пропонується два джерела даних, пов’язаних з курсами валют різних країн та криптовалюти (див. рис. 3.9).

Рисунок 3.8 – Форма параметрів візуалізації даних

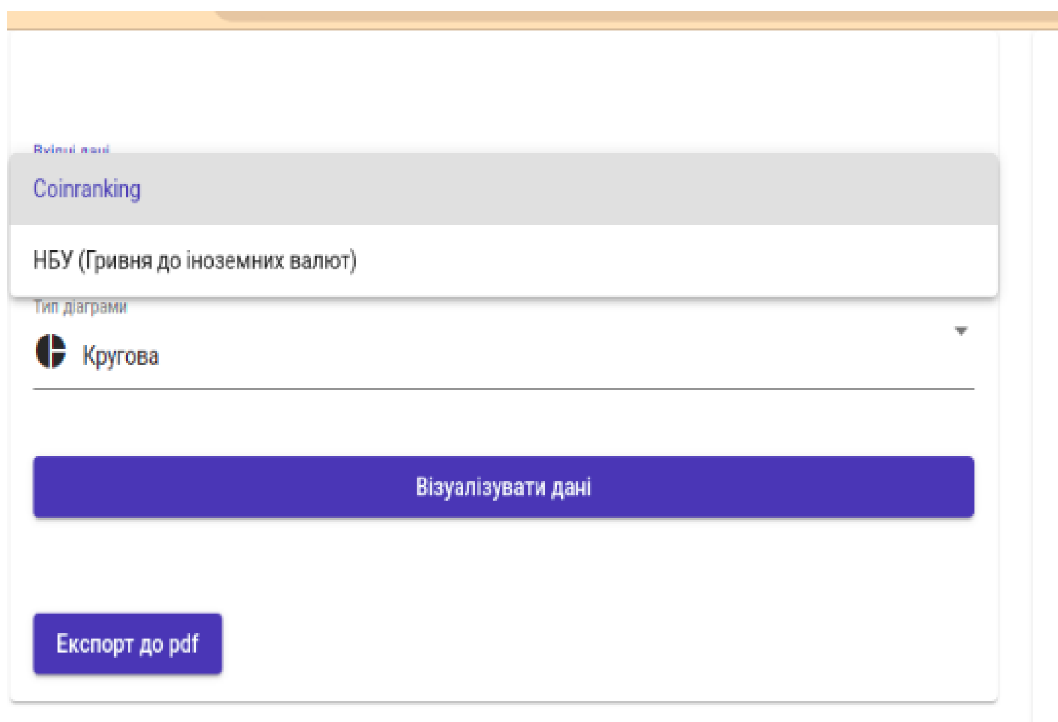

Рисунок 3.9 – Перелік джерел вхідних даних

3.7.4 Керування діаграмами

Після вибору джерела вхідних даних, необхідно визначитися з типом діаграми для їх візуалізації. На вибір пропонуються шість видів діаграм (див.

рис. 3.10), що надають можливість відображувати та аналізувати дані з різних кутів:

– кругова діаграма (pie chart) використовується для графічного відображення даних у вигляді кругової частки або сектора, також вона дозволяє ілюструвати пропорції часток даних відносно цілого;
– стовпчикова діаграма (bar chart) використовується для графічного відображення даних у вигляді стовпців, які представляють різні категорії та їх відносні значення, також вона дозволяє порівнювати значення різних категорій та аналізувати їх відносність один до одного;
– діаграма-графік (line chart) використовується для візуалізації залежності між значеннями на осі X (наприклад, час) та відповідними значеннями на осі Y, також вона дозволяє відслідковувати зміну значень впродовж певного періоду часу або відображати будь-яку іншу прогресію;
– бульбашкова діаграма (bubble chart) використовується для візуалізації трьох вимірних даних, вона подібна до точкової діаграми, але додає третій вимір – розмір (або вагу) елементу, бо кожен елемент діаграми представляється у вигляді круга (бульбашки), розмір якого відображає третій вимір даних;
– полярна діаграма (polar chart), також відома як кругова діаграма або вітрова розсіяння, використовується для візуалізації даних, які мають циклічну або кругову структуру, вона зображає дані на круговій сітці з полюсами, що відповідають різним значенням на осі X, а радіуси кіл відображають значення на осі Y;
– діаграма-радар, також відома як полігональна діаграма або спайдер-діаграма, використовується для візуалізації багатовимірних даних, вона складається зі сполучених ліній, які виходять з центральної точки і представляють значення різних змінних, бо кожна лінія відображає одну змінну, а форма, розмір та положення полігону показують, як ці значення співвідносяться між собою.

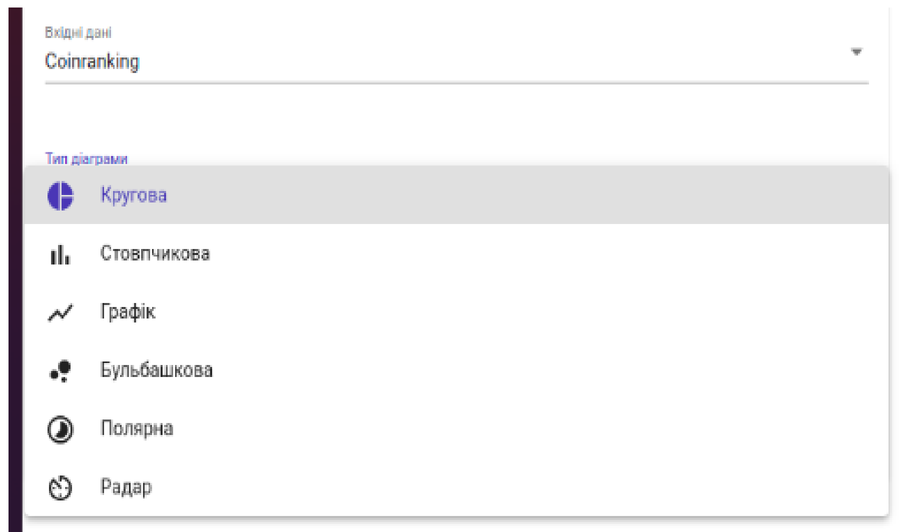

Рисунок 3.10 – Перелік типів діаграм

3.7.5 Візуалізація відображення

Після вибору джерела вхідних даних та типу діаграми для відображення, потрібно натиснути на кнопку «Візуалізувати дані». Система відобразить обраний тип діаграми, опираючись на джерело вхідних даних, в правій частині сторінки (див. рис. 3.11).

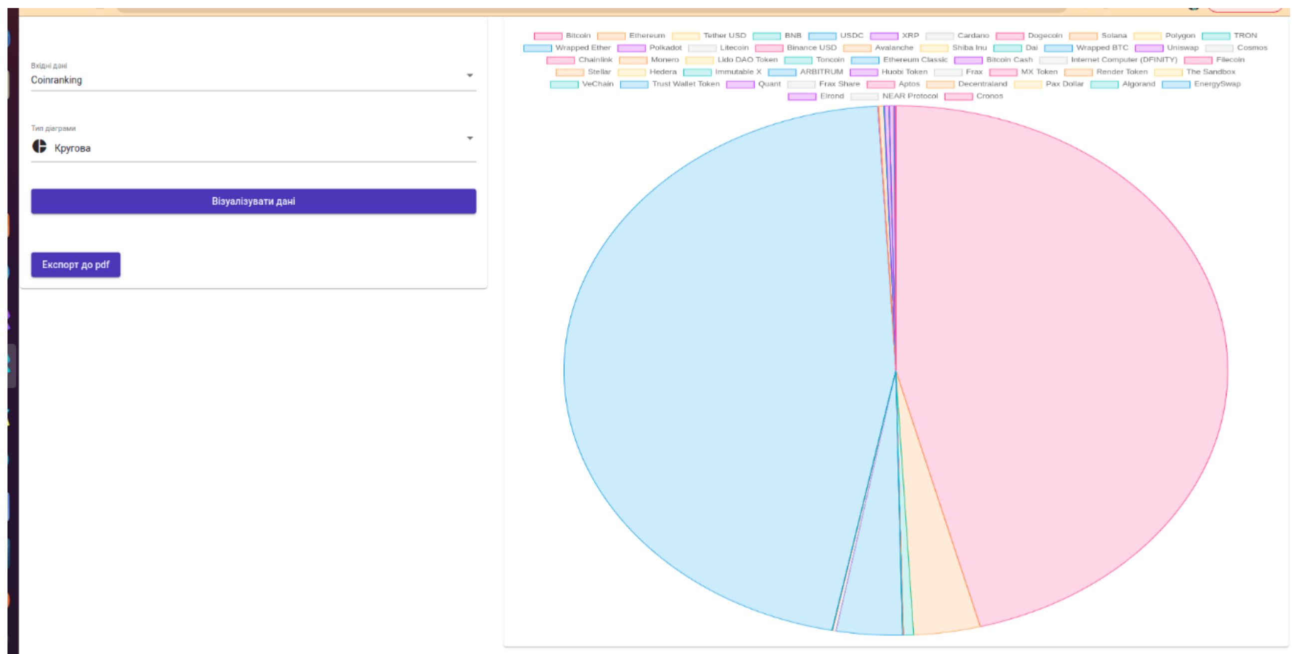

Рисунок 3.11 – Приклад візуалізації даних за допомогою кругової діаграми

Залежно від потреб та актуальності результату відображення, можна змінити параметри та проаналізувати, який вид діаграми найбільш вдало візуалізує вхідні дані. На рисунках 3.12 – 3.14 зображено приклади візуалізації наборів даних за допомогою різних типів діаграм.

Рисунок 3.12 – Візуалізація даних за допомогою стовпчикової діаграми

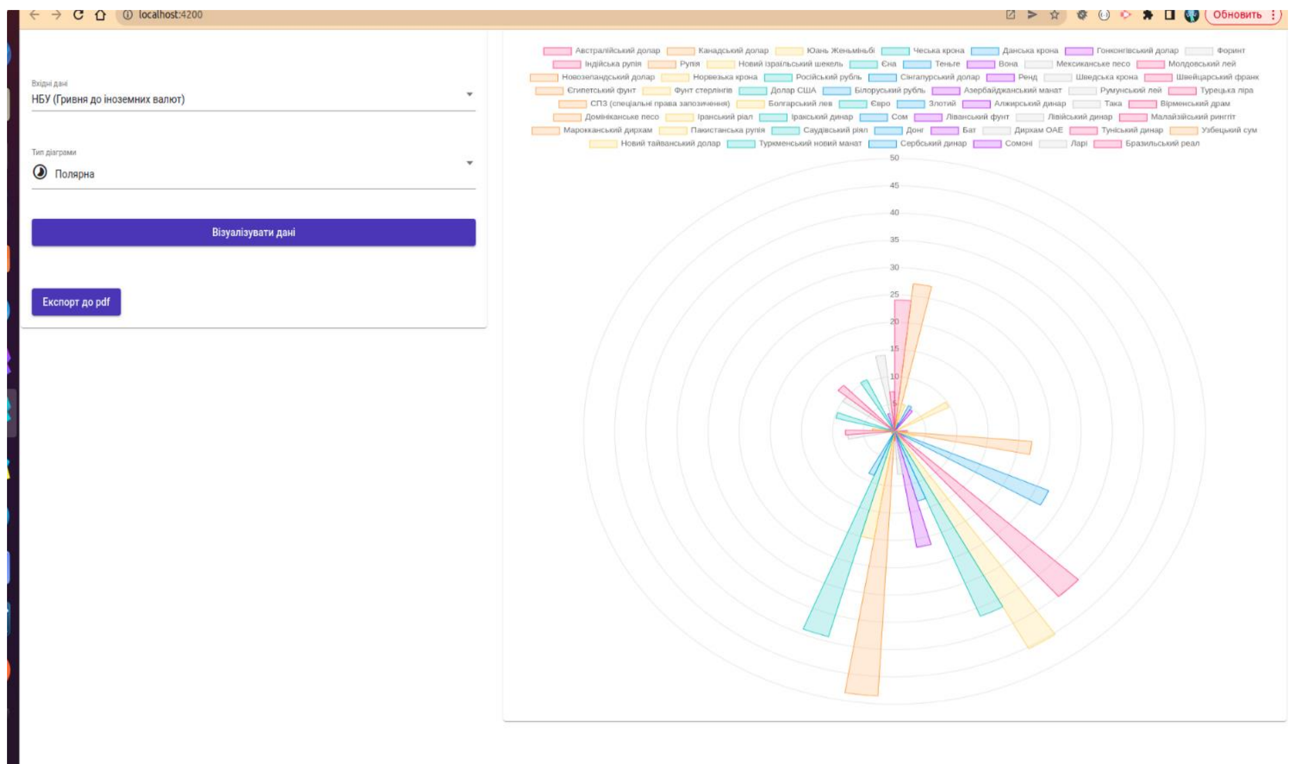

Рисунок 3.13 – Візуалізація даних за допомогою полярної діаграми

Рисунок 3.14 – Візуалізація даних за допомогою графіку

3.7.6 Фільтрація вхідних даних

Також система надає можливість фільтрації вхідних даних у кругових, полярних да радарних діаграмах. Для цього необхідно натиснути на назву елементу, який необхідно приховати або показати. Прихований елемент не приймає участь у відображенні та його назва перекреслена (див. рис. 3.15). Це дає можливість порівняти та проаналізувати конкретні елементи або з меншою вагою.

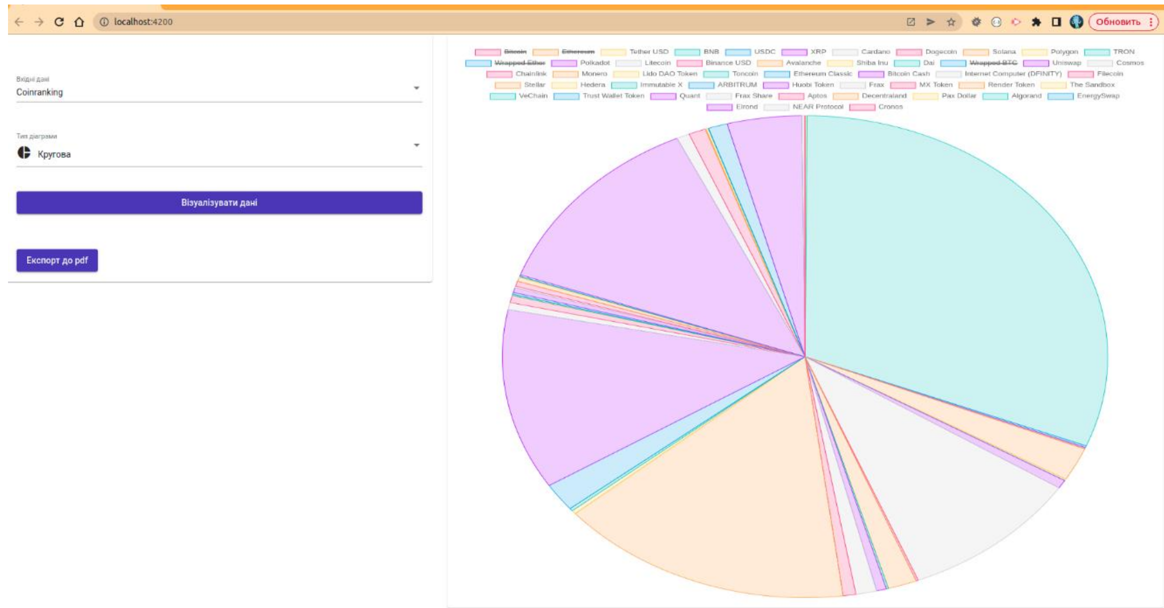

Рисунок 3.15 – Фільтрація даних

3.7.7 Експорт даних до pdf

Для експорту даних до pdf, необхідно натиснути на кнопку «Експорт до pdf», система завантажить файл з короткою назвою джерела даних (рис. 3.16 та 3.17).

Рисунок 3.16 – Завантаження файлу

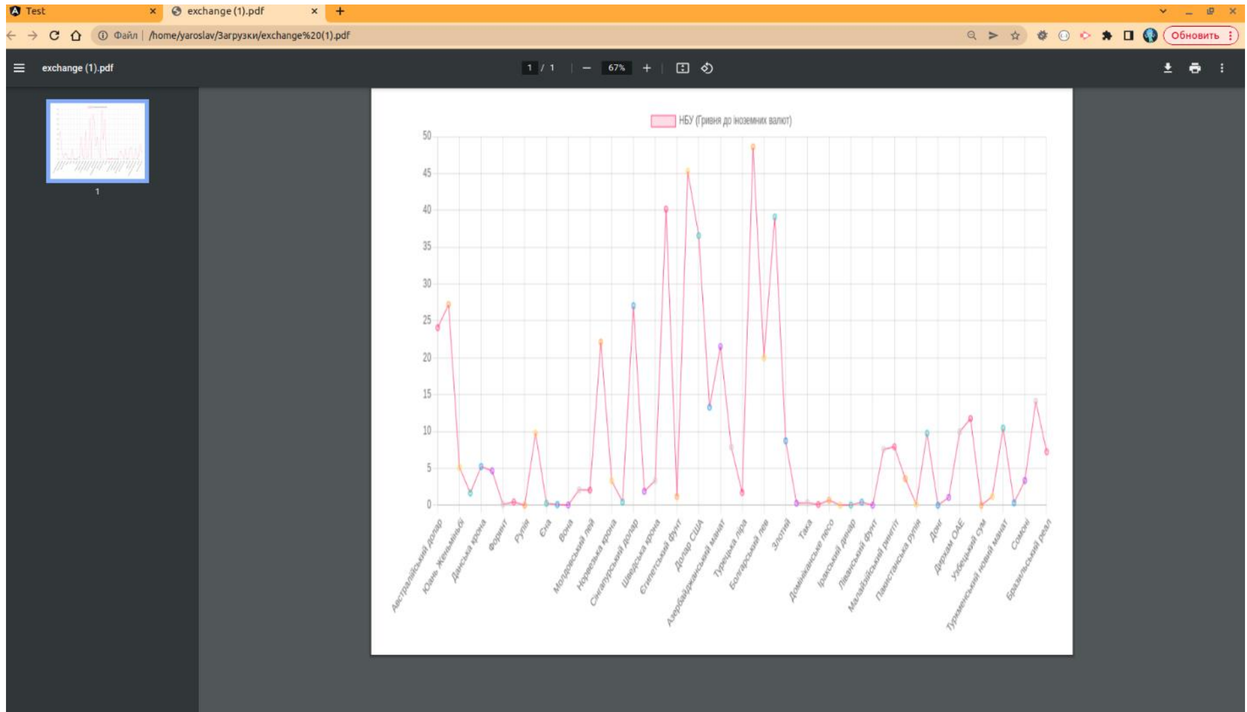

Рисунок 3.17 – Перегляд файлу

ВИСНОВКИ

В результаті роботи було написано технічне завдання на розробку вебзастосунку візуалізації даних. Для створення цієї системи були обрані фреймворк Angular зі сторони клієнта та Node.js зі сторони сервера, за їх широкі можливості у сфері створення web-систем.

У відповідності з метою кваліфікаційної роботи було розроблено вебзастосунок візуалізації даних із застосуванням наступних технологій:

- Node.js для реалізації backend API;
- Angular для реалізації front end частини, а також відправки запитів до серверу.

У відповідності з поставленими задачами були виконані наступні етапи створення системи:

- сформовані вимоги до системи (функціональні та нефункціональні (інтерфейс, кросбраузерність, безпека, продуктивність)), а також проведено огляд предметної області та інструментів розробки;
- спроєктована та побудована структура системи (побудовані діаграми прецедентів, діяльності, послідовності та розгортання; надано детальний опис прецедентів);
- реалізовано вебзастосунок візуалізації даних (наведена інструкція по створенню компонентів системи, надано керівництво користувача та структура проєкту);
- протестована робота системи.

ПЕРЕЛІК ПОСИЛАНЬ

1. Angular Developer Documentation. URL: https://angular.io/docs/ (дата звернення: 11.03.2023).
2. Bampakos A. Angular Projects: Build modern web apps by exploring Angular 12 with 10 different projects and cutting-edge technologies. Birmingham : Packt Publishing, 2021. 344 p.
3. Bampakos A., Deeleman P. Learning Angular: A no-nonsense guide to building web applications with Angular 15. Birmingham : Packt Publishing, 2023. 446 p.
4. Chart.js Developer Documentation. URL: https://www.chartjs.org/docs/ (дата звернення: 19.03.2023).
5. Chebbi L. Reactive Patterns with RxJS for Angular: A practical guide to managing your Angular application’s data reactively and efficiently using RxJS 7. Birmingham : Packt Publishing, 2022. 224 p.
6. Fain Y., Moiseev A. Angular Development with TypeScript. New York : Manning Publications, 2019. 560 p.
7. Freeman A. Pro Angular: Build Powerful and Dynamic Web Apps. London: Apress, 2022. 905 p.
8. Li D. Introduction to Backend Development with Node.js and Express: Building Robust Web Applications with Node.js and Express Framework. Vancouver : Kindle Edition, 2023. 94 p.
9. Node.js Developer Documentation. URL: https://nodejs.org/en/docs (дата звернення: 01.04.2023).
10. Rappl F. Modern Frontend Development with Node.js: A compendium for modern JavaScript web development within the Node.js ecosystem. Birmingham : Packt Publishing, 2022. 208 p.

ДОДАТОК А

Angular-компонент

```
import {Component, ElementRef, Inject, OnInit, ViewChild} from '@angular/core';
import {ViewService} from '../../services/view.service';
import {FormControl, FormGroup, Validators} from '@angular/forms';
import {InputData} from '../../core/models/input.data';
import {dataList} from '../../core/data/data.list';
import {TypeDiagram} from '../../core/models/type.diagram';
import {diagramList} from '../../core/data/diagram.list';
import { Chart, registerables } from 'chart.js';
import {DOCUMENT} from '@angular/common';
import html2canvas from 'html2canvas';
import {jsPDF} from 'jspdf';

@Component({
  selector: 'app-view',
  templateUrl: './view.component.html',
  styleUrls: ['./view.component.scss']
})
export class ViewComponent implements OnInit {

  @ViewChild('pdfContainer') container: ElementRef | undefined;

  form: FormGroup;
  inputData: InputData[] = dataList;
  diagrams: TypeDiagram[] = diagramList;
  selectDiagram: any = this.diagrams[0];
  labels: any;
  dataAPI: any;
  chart!: Chart;

  constructor(
    private viewService: ViewService,
    @Inject(DOCUMENT) private document: Document,
  ) {
```

```
    Chart.register(...registerables);
    this.form = new FormGroup({
      data: new FormControl('coin', [Validators.required]),
      diagram: new FormControl('pie', [Validators.required])
    });
  }

  ngOnInit(): void {
  }

  onSubmit(): void {
    if (this.form.valid) {
      switch (this.form.get('data')?.value) {
        case 'coin':
          this.viewService.getCoins().then(
            (res) => {
              this.dataAPI = res.map((coins: any) => coins.price);
              this.labels = res.map((coins: any) => coins.name);
              this.buildChart(this.dataAPI, this.labels, this.selectDiagram.value, this.inputData.find(item =>
item.value === this.form.get('data')?.value)?.name);
            },
            error => {
              console.log(error);
            }
          );
          break;
        case 'exchange':
          this.viewService.getExchangeRate().then(
            (res) => {
              this.dataAPI = res.map((exchange: any) => exchange.rate);
              this.labels = res.map((exchange: any) => exchange.txt);
              this.dataAPI.splice(-4, 4);
              this.labels.splice(-4, 4);
              this.buildChart(this.dataAPI, this.labels, this.selectDiagram.value, this.inputData.find(item =>
item.value === this.form.get('data')?.value)?.name);
            },
            error => {
              console.log(error);
```

```
        }
      );
      break;
    }
  }
}

onDiagramChange(event: any): void {
  this.selectDiagram = this.diagrams.find(item => item.value === event);
}

buildChart(dataAPI: any, label: any, types: any, title: any): void {
  if (this.chart)
  {
    this.chart.destroy();
  }
  this.chart = new Chart('canvas', {
    type: types,
    data: {
      labels: label,
      datasets: [
        {
          data: dataAPI,
          label: title,
          backgroundColor: [
            'rgba(255, 99, 132, 0.2)',
            'rgba(255, 159, 64, 0.2)',
            'rgba(255, 205, 86, 0.2)',
            'rgba(75, 192, 192, 0.2)',
            'rgba(54, 162, 235, 0.2)',
            'rgba(153, 102, 255, 0.2)',
            'rgba(201, 203, 207, 0.2)'
          ],
          borderColor: [
            'rgb(255, 99, 132)',
            'rgb(255, 159, 64)',
            'rgb(255, 205, 86)',
            'rgb(75, 192, 192)',
```

```
            'rgb(54, 162, 235)',
            'rgb(153, 102, 255)',
            'rgb(201, 203, 207)'
          ],
          borderWidth: 1,
        },
      ],
    },
  });
}

print(): void {
  console.log(this.container);
  const htmlWidth = this.container?.nativeElement.offsetWidth;
  const htmlHeight = this.container?.nativeElement.offsetHeight;

  const pdfWidth = htmlWidth + 60;
  const pdfHeight = htmlHeight + 60;

  const canvasImageWidth = htmlWidth;
  const canvasImageHeight = htmlHeight;

  const pdfData = this.document.getElementById('canvas') as HTMLElement;

  html2canvas(pdfData, {allowTaint: true}).then(canvas => {
    canvas.getContext('2d');
    const imgData = canvas.toDataURL('image/png', 2.0);
    const pdf = new jsPDF('l', 'pt', [pdfWidth, pdfHeight]);
    pdf.addImage(imgData, 'png', 30, 30, canvasImageWidth, canvasImageHeight);

    pdf.save(`${this.form.get('data')?.value}.pdf`);
  });
}
}
```

ДОДАТОК Б

Angular-сервіс

```
import { Injectable } from ‘@angular/core’;
import {HttpClient, HttpHeaders} from ‘@angular/common/http’;
import {Observable} from ‘rxjs’;

@Injectable({
  providedIn: ‘root’
})
export class ViewService {

  private baseUrl = ‘http://localhost:3000/api’;
  headers: any;

  constructor(private http: HttpClient) {
    this.headers = new HttpHeaders({
      Accept: ‘application/json’,
      ‘Content-Type’: ‘application/json’,
      ‘Access-Control-Allow-Origin’: ‘*’,
      ‘Access-Control-Allow-Headers’: ‘Content-Type’,
      ‘Access-Control-Allow-Methods’: ‘GET,POST,OPTIONS,DELETE,PUT’,
    });
  }

  getCoins(): Promise<any> {
    return this.http
      .get(this.baseUrl + ‘/coin’, {headers: this.headers})
      .toPromise()
      .then((data) => {
        return data;
      });
  }

  getExchangeRate(): Promise<any> {
    return this.http
```

```
      .get(this.baseUrl + ‘/nbu’, {headers: this.headers})
      .toPromise()
      .then((data) => {
        return data;
      });
  }
}
```

ДОДАТОК В

Node.js-модель

```
const axios = require(‘axios’);
require(‘dotenv’).config()

const URL = process.env.API_URL_COIN
const APIKEY = process.env.API_KEY_COIN

const options = {
  headers: {
    ‘Content-Type’: ‘application/json’,
    ‘x-access-token’: APIKEY,
  },
}


module.exports = class Coin {
  static async getData() {
    return new Promise( (resolve, reject) => {
      axios
        .get(URL, options)
        .then( (response) => {
          resolve(response.data.data.coins);//обрабатываем ответ
        }).catch((error) => {
          reject(error);
      });
    }).catch(err => console.log(err));
  }
};
```

ДОДАТОК Г

Node.js-контролер

```
const Coin = require("../models/coin.model.js");
const NBU = require('../models/nbu.model.js');

exports.getCoin = async (req, res) => {
   res.send(await Coin.getData().then(res => res).catch(err => err));
};

exports.getExchangeRate = async (req, res) => {
   res.send(await NBU.getData().then(res => res).catch(err => err));
};
```