МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ВБУДОВАНОГО ПРОГРАМНОГО ЗАСТОСУНКУ ДЛЯ ЦИФРОВИХ БІЛБОРДІВ»**

Виконала: студентка 4 курсу, групи 6.1211-1пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
Н.Г. Удинська
(ініціали та прізвище)

Керівник декан математичного факультету, д.т.н., професор Гоменюк С.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної та прикладної математики, д.т.н., професор Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

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)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Удинській Наталії Геннадіївні
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка вбудованого програмного застосунку для цифрових білбордів

керівник роботи Гоменюк Сергій Іванович, д.т.н., професор
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 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.
2. Основні теоретичні відомості.
3. Програмна реалізація застосунку для збору інформації цифрових моніторів.

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

6. Консультанти розділів роботи

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

7. Дата видачі завдання 26.12.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	08.01.2025	
2.	Збір вихідних даних.	20.01.2025	
3.	Обробка методичних та теоретичних	17.02.2025	
	джерел.		
4.	Розробка першого та другого розділу.	24.03.2025	
5.	Розробка третього розділу.	19.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	17.06.2025	

Студент _____ Н.Г. Удинська
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.І. Гоменюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка вбудованого програмного застосунку для цифрових білбордів»: 64 с., 44 рис., 4 табл., 15 джерел, 2 додатки.

БІБЛІОТЕКА, ЗАСТОСУНОК, МОНІТОР, CSS, ELECTRON, HTML, JAVASCRIPT, VISUAL STUDIO CODE.

Об’єкт дослідження – отримання та аналіз інформації про цифрові монітори в умовах віддаленого доступу.

Предмет дослідження – програмне забезпечення для дистанційного моніторингу стану комп’ютера та цифрових моніторів.

Мета роботи – розробка програми, яка дистанційно надає інформацію про стан комп’ютера та підключених до нього моніторів.

Метод дослідження – методи збору та аналізу вимог до програмного забезпечення, а також підходи до проєктування, розробки та тестування програмного забезпечення.

При розробці застосунку було проведено аналіз та визначено актуальність предметної області. Реалізовано web-застосунок з використанням таких технологій як NPM System information, HTML, CSS, JavaScript, Electron.js, ElectronBuilder.js, для збору інформації про стан комп’ютеру і підключених до нього цифрових моніторів за допомогою Visual Studio Code .

В ході виконання роботи отримано застосунок, який надає користувачам інформацію про стан комп’ютера та підключених моніторів у дистанційному режимі.

SUMMARY

Bachelor’s qualifying paper “Development of an Embedded Software Application for Digital Billboards”: 64 pages, 44 figures, 4 tables, 15 references, 2 supplements.

LIBRARY, APPLICATION, MONITOR, CSS, ELECTRON, HTML, JAVASCRIPT, VISUAL STUDIO CODE.

Object of the research is obtaining and analyzing information about digital monitors under conditions of remote access.

Subject of the research is software for remote monitoring of the computer’s condition and connected digital monitors.

Purpose of the work is the development of a program that remotely provides information about the condition of a computer and the monitors connected to it.

Research method involves gathering and analyzing software requirements, as well as approaches to software design, development, and testing.

During the development of the application, an analysis was conducted and the relevance of the subject area was determined. A web application was implemented using technologies such as NPM System Information, HTML, CSS, JavaScript, Electron.js, and ElectronBuilder.js to collect information about the computer and connected digital monitors, using Visual Studio Code.

As a result of the work, an application was created that provides users with remote access to information about the condition of the computer and connected monitors.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Аналіз предметної області та вибір інструментальних засобів 10
1.1 Поняття цифрових білбордів і особливості їх використання 10
1.2 Аналіз існуючих рішень для моніторингу цифрових екранів 10
1.3 Постановка задачі на розробку застосунку 11
1.4 Обґрунтування вибору технологій та інструментів розробки 12
1.5 Висновки до розділу 1 14
2 Проєктування програмного забезпечення 16
2.1 Визначення функціональних та нефункціональних вимог 16
2.1.1 Функціональні вимоги 16
2.1.2 Нефункціональні вимоги 17
2.2 Архітектура за стосунку 17
2.3 UML-діаграми та їх опис 18
2.3.1 Use Case Diagram 18
2.3.2 Sequence Diagram 20
2.3.3 Component Diagram 22
2.4 Вибір технологій 25
2.5 Висновки до розділу 2 26
3 Реалізація програмного забезпечення 28
3.1 Структура проєкту 28
3.2 Реалізація головного процесу (main.js) 29
3.2.1 Імпорт бібліотек 29
3.2.2 Функція createFirstWindow() 30
3.2.3 Запуск застосунку 32

3.2.4 Завершення роботи застосунку 33
3.3 Реалізація preload.js 34
3.4 Взаємодія з бібліотекою systeminformation для збору системних даних 35
3.4.1 Обробка незмінної системної інформації 35
3.4.2 Динамічна інформація про систему 36
3.4.3 Інформація про підключені монітори 37
3.5 Реалізація механізму авто оновлення 38
3.6 Інтерфейс користувача (HTML + CSS + renderer.js) 40
3.7 Створення скриншотів екрана 41
3.8 Tray-меню як елемент навігації 43
3.8.1 Реалізація Tray-меню 43
3.8.2 Результат реалізації Tray-меню 45
3.9 Збірка, установка та запуск застосунку 46
3.10 Запуск за стосунку 48
3.11 Демонстрація роботи та результати 49
3.12 Висновки до розділу 52
Висновки 54
Перелік посилань 55
Додаток А Корисні сервіси та посилання 57
Додаток Б Приклади роботи за стосунку 58
Б.1 Приклади роботи за стосунку за допомогою HDMI-кабелю 58
Б.2 Приклади роботи за стосунку через віддалене підключення до комп’ютера 61

ВСТУП

Сучасний світ стрімко змінюється під впливом новітніх технологій, що охоплюють практично всі сфери . Особливо це помітно у сфері маркетингу та реклами, де дедалі більше компаній переходять на цифрові рішення відмовляючись від традиційних паперових носіїв на користь сучасних цифрових моніторів. Це необхідністю для оперативного оновлення рекламного контенту, підвищення ефективності взаємодії з аудиторією та оптимізації витрат на розповсюдження інформації.

Одним із яскравих прикладів таких змін є діяльність німецької компанії Heico Media & Co GmbH, яка спеціалізується на встановленні цифрових моніторів та займається розробкою власного програмного забезпечення "tv2b", що дозволяє транслювати рекламні оголошення та інформаційні повідомлення в різних містах не тільки Німеччини, а також Австрії.

Актуальність обраної теми зумовлена потребою у віддаленому моніторингу цифрових моніторів та комп’ютерів, до яких вони підключені.

Мета роботи – розробка програми, яка дистанційно надає інформацію про стан комп’ютера та підключених до нього моніторів.

Досягнення поставленої мети передбачає реалізацію наступних завдань:

– проаналізувати предметну область цифрового моніторингу;
– дослідити можливості використання Electron.js для створення десктопного застосунку;
– розробити інтерфейс користувача для відображення стану системи;
– спроєктувати логіку автоматичного збору технічних даних;
– реалізувати механізм створення та збереження скриншотів;
– протестувати застосунок у середовищі Windows.

Об’єкт дослідження – отримання та аналіз інформації про цифрові монітори в умовах віддаленого доступу.

Предмет дослідження – програмне забезпечення для дистанційного моніторингу стану комп’ютера та цифрових моніторів.

Метод дослідження – методи збору та аналізу вимог до програмного забезпечення, а також підходи до проєктування, розробки та тестування програмного забезпечення.

Програму було розроблено з використанням наступних технологій:

– JavaScript – основна мова програмування;
– HTML та CSS – для візуального представлення даних;
– Electron.js – для реалізації логіки та взаємодії з операційною системою;
– NPM Systeminformation – бібліотека для збору інформації про обладнання.

У результаті розроблено програму, що працює у фоновому режимі, збирає системну інформацію, створює скриншоти, зберігає дані у форматі JSON і забезпечує зручний інтерфейс. Реалізовано tray-меню, автозапуск і автоматичне оновлення без участі користувача.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, переліку використаних джерел та двох додатків. Загальний обсяг роботи становить 64 сторінки. Додатки викладено на 8 сторінках. У списку використано 15 джерел.

У першому розділі розглянуто предметну область, проаналізовано існуючі рішення та обґрунтовано вибір технологій. У другому розділі описано проєктування програмного забезпечення, вимоги до нього та UML-діаграми. У третьому розділі представлено реалізацію застосунку, його структуру, функціонал, інтерфейс і механізми оновлення. У додатках наведено приклади роботи програми.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ

1.1 Поняття цифрових білбордів і особливості їх використання

Цифрові білборди – це великоформатні екрани, які використовуються для трансляції рекламного, інформаційного або соціального контенту у публічних просторах. На відміну від традиційних друкованих носіїв, цифрові білборди мають низку переваг: можливість динамічного оновлення інформації, інтерактивність, гнучке керування контентом та віддалене адміністрування.

Ці системи зазвичай складаються з таких компонентів:

– дисплейний пристрій (LED або LCD екран);
– керуючий комп’ютер або мікроконтролер, підключений до дисплею;
– мережеве підключення (Ethernet, Wi-Fi, 4G/5G);
– програмне забезпечення для трансляції та моніторингу стану пристрою.

Оскільки цифрові білборди зазвичай розташовані у віддалених або важкодоступних місцях, важливу роль відіграє дистанційний моніторинг їхнього технічного стану. Це дозволяє зменшити кількість виїздів технічного персоналу та швидко реагувати на збої у роботі.

1.2 Аналіз існуючих рішень для моніторингу цифрових екранів

Існує багато готових рішень для управління контентом на білбордах. Розглянемо їх детальніше.

tv2b (Heico Media GmbH) – комерційне рішення для цифрових білбордів у Німеччині та Австрії. Система дає змогу транслювати

оголошення у різних містах, працює за централізованим принципом, має вбудований контент-менеджер. Втім, можливості технічного моніторингу реалізовано частково, або інтегровано через окремі інструменти.

Xibo – open-source платформа для керування дисплеями, яка дозволяє створювати контент, планувати розклад і переглядати статистику. Має часткові можливості моніторингу стану клієнтських пристроїв через логування або API [1].

Screenly OSE / Pro – рішення для Raspberry Pi, розроблене з акцентом на простоту використання. Професійна версія дозволяє централізовано керувати пристроями, хоча базова open-source версія не включає функцій глибокого технічного моніторингу [2].

У більшості рішень моніторинг технічного стану – лише додаткова функція. Саме тому розробка окремого застосунку, що фокусується саме на зборі системної інформації з комп'ютерів білбордів (наприклад, температура процесора, завантаження, підключені дисплеї, скриншоти) – є актуальним завданням, особливо для компаній, які використовують нестандартне обладнання або прагнуть кастомізувати контрольні механізми.

1.3 Постановка задачі на розробку застосунку

Завданням даної роботи є створення вбудованого програмного забезпечення (застосунку), який працюватиме у фоновому режимі на комп'ютерах, підключених до цифрових білбордів, та буде виконувати наступні функції:

- збір інформації про стан системи: температура процесора, використання ОЗП, мережеві інтерфейси, статус підключених моніторів;
- фіксація скриншотів дисплея через задані інтервали;
- збереження логів та попереджень про несправності;

– віддалена передача або відображення даних для технічного персоналу;
– автозапуск при включенні ПК, мінімізація до tray-меню.

Кінцевий продукт повинен бути:

– кросплатформенним (Windows, Linux);
– енергоефективним, щоб не заважати роботі основного ПЗ для трансляції;
– мінімалістичним за інтерфейсом, з акцентом на функціональність.

1.4 Обґрунтування вибору технологій та інструментів розробки

Electron – це фреймворк з відкритим кодом, який дозволяє створювати кросплатформені настільні застосунки з використанням вебтехнологій: HTML, CSS та JavaScript [3]. Він з'єднує браузер Chromium із Node.js, що даю можливість створювати повноцінні десктопні застосунки з доступом до системних ресурсів.

Переваги:

– працює на Windows, Linux, macOS;
– підтримує npm-бібліотеки, зокрема systeminformation;
– дозволяє працювати з файловою системою, створювати автозапуск, tray-меню;
– створення інтерфейсу за допомогою знайомих вебтехнологій.

Недоліки:

– великий розмір виконуваного файлу через вбудований Chromium;
– вища витрата ресурсів порівняно з нативними застосунками;
– не підходить для систем із дуже обмеженими можливостями (наприклад, старі SBC-пристрої).

JavaScript – основна мова програмування для веб та Electron-застосунків. У поєднанні з Node.js даю можливість створювати не тільки інтерфейсну, а й і серверну логіку [4].

Переваги:

– популярність та велика кількість бібліотек;
– динамічний підхід до розробки;
– підтримка асинхронних операцій;
– проста інтеграція з HTML/CSS та Electron.

Недоліки:

– динамічна типізація може призводити до помилок без ретельного тестування;
– менш ефективна для складних математичних обчислень або задач низького рівня.

HTML – мова розмітки інтерфейсу, а CSS – мова стилів для оформлення. Разом формують основу UI в Electron-застосунках [5, 6].

Переваги:

– швидке створення зрозумілих та адаптивних інтерфейсів;
– широка підтримка UI-бібліотек;
– гнучкість стилізації без потреби у складному кодуванні;
– візуальна простота для користувача.

Недоліки:

– потреба в ретельному тестуванні на різних розмірах екранів;
– складність при реалізації дуже складного або нестандартного UI без додаткових фреймворків.

Systeminformation – бібліотека для Node.js, яка надає можливість отримувати широкий набір технічних параметрів про систему: CPU, RAM, монітори, температура тощо [7].

Переваги:

– працює без сторонніх драйверів;
– широкий набір доступної інформації;
– підтримка всіх основних ОС;
– асинхронний API, що не блокує виконання.

Недоліки:

– для деяких даних потрібен запуск з правами адміністратора;

– певні функції (наприклад, доступ до відеокарти або температури) залежать від платформи.

Visual Studio Code – багатоплатформове середовище розробки з відкритим кодом, створене компанією Microsoft [8].

Переваги:

– підтримка широкого спектру мов програмування;
– велика кількість розширень для Electron, Node.js, Git;
– вбудовані засоби дебагінгу та автодоповнення;
– простий та зручний інтерфейс.

Недоліки:

– високе споживання ресурсів при роботі з великими проєктами;
– не всі розширення працюють однаково стабільно на різних платформах.

Electron-builder – інструмент для створення інсталяційних пакетів для Electron-застосунків [9].

Переваги:

– підтримка різних форматів файлів: .exe, .AppImage, .deb тощо;
– простота налаштування пакування;
– можливість автоматичного оновлення застосунку;
– створення цифрових підписів для захисту інсталяцій.

Недоліки:

– деякі функції можуть некоректно працювати в різних ОС без додаткових скриптів;
– створення пакунків для Linux вимагає роботи з терміналом і знань про формати .deb або .rpm.

1.5 Висновки до розділу 1

У першому розділі було проаналізовано предметну область цифрових білбордів та визначено їхню роль у сучасній рекламній індустрії. Зростання

популярності цифрових засобів інформування зумовило потребу в розробці спеціалізованого програмного забезпечення, яке дозволяє здійснювати дистанційний моніторинг технічного стану пристроїв.

На підставі аналізу існуючих рішень встановлено, що більшість платформ зосереджуються на управлінні контентом, але не забезпечують повноцінний технічний моніторинг обладнання. Це створює потребу у розробці окремого вбудованого застосунку, який би дозволяв отримувати системну інформацію в реальному часі та оперативно реагувати на потенційні збої.

Було сформульовано основну мету проєкту – створити настільний застосунок, який дистанційно надає дані про стан комп'ютера та підключених моніторів, фіксує знімки екрану, логування помилок і підтримує роботу у фоновому режимі.

Також було обґрунтовано вибір стеку технологій для реалізації проєкту:

- Electron.js – як основний фреймворк для створення кросплатформених застосунків;
- JavaScript – для реалізації логіки роботи та взаємодії з системними компонентами;
- HTML і CSS – для створення зручного інтерфейсу користувача;
- systeminformation – як основна бібліотека для збору технічної інформації;
- Visual Studio Code – як середовище розробки;
- Electron-builder – для пакування застосунку в інсталяційний файл.

Поряд із перевагами, було також проаналізовано недоліки кожного з інструментів. Це дозволяє усвідомлено підходити до розробки, враховувати потенційні обмеження і планувати рішення для їх подолання.

Таким чином, у розділі закладено теоретичне та технологічне підґрунтя для подальшого проєктування та реалізації застосунку, що стане предметом наступних етапів роботи.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Визначення функціональних та нефункціональних вимог

Проєктування програмного забезпечення для моніторингу стану цифрових білбордів почалося з формулювання чітких вимог. Основними вимогами до системи стали функціональні та нефункціональні вимоги, що забезпечують ефективну та стабільну роботу програми.

2.1.1 Функціональні вимоги

Функціональні вимоги описують конкретні функції, які повинно виконувати програмне забезпечення.

Збір технічної інформації про пристрій:

– отримання інформації про комп’ютер: його назву, серійний номер, операційну систему, мережеві налаштування (IP, MAC, SSID);
– моніторинг стану процесора, його завантаження та температури;
– збір даних про підключені монітори, їхні характеристики та поточний стан.

Автоматичне оновлення даних: система повинна оновлювати дані про пристрій з заданим інтервалом, наприклад, кожні 5 хвилин.

Можливість створення скриншотів: програма повинна автоматично створювати скриншоти з підключених дисплеїв щохвилини.

Відображення інформації в графічному інтерфейсі: дизайн інтерфейсу має бути мінімалістичним і зручним для користувача. Інформація повинна бути зрозумілою і доступною для швидкого перегляду.

Робота у фоновому режимі: застосунок повинен працювати у фоновому режимі та підтримувати tray-меню, яке дозволяє користувачеві швидко взаємодіяти з програмою.

Автоматичне оновлення програми: програма повинна мати механізм автоматичного оновлення, який завантажує та встановлює нові версії без втручання користувача.

Збереження логів: усі події та зібрані дані мають зберігатися у форматі JSON для подальшого аналізу та діагностики.

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають характеристики системи, які стосуються її продуктивності, безпеки та надійності:

- кросплатформенність: застосунок має працювати на різних операційних системах, таких як Windows та Linux, для забезпечення широкого використання;
- надійність і стабільність: застосунок повинен стабільно працювати у фоновому режимі, без суттєвого навантаження на систему та без переривання основних функцій пристрою;
- енергоефективність: програма має мінімально впливати на енергоефективність пристроїв, не споживаючи зайві ресурси;
- безпека: доступ до системних API має бути забезпечений через захищені канали IPC та preload- процес, щоб уникнути потенційних вразливостей;
- масштабованість: система має бути спроєктована таким чином, щоб можна було легко інтегрувати її з іншими програмними рішеннями, наприклад, для централізованого моніторингу через хмарні сервіси.

2.2 Архітектура за стосунку

Застосунок побудований на основі Electron.js, що дозволяє створювати кросплатформенні застосунки.

Він розділений на три основні компоненти: головний процес, рендер-процес і preload.

Головний процес (file main.js):

– відповідає за ініціалізацію вікон застосунку та запуск автооновлень;
– використовує electron-updater для перевірки оновлень і автоматичної їх установки;
– організовує обробку системних запитів через IPC [10].

Рендер-процес (file renderer.js):

– відповідає за відображення отриманої інформації користувачеві;
– збирає дані через запити до головного процесу та представляє їх у вигляді таблиць чи графіків.

Preload (file preload.js):

– безпечний канал між головним і рендер-процесами через contextBridge;
– використовує IPC для передачі даних між процесами.

2.3 UML-діаграми та їх опис

Unified Modeling Language (UML) – це стандартна мова моделювання. Вона використовується для візуалізації, специфікації, конструювання та документування програмних систем [11].

Тож для кращого розуміння процесів та взаємодії компонентів, у розробці застосунку використовувалися UML-діаграми.

2.3.1 Use Case Diagram

Діаграма варіантів використання (Use Case Diagram) – це вид поведінкової діаграми UML, що описує функціональність системи з точки

зору користувача. Вона показує, які дії користувач може виконувати в системі, не заглиблюючись у технічні деталі реалізації [12].

На рисунку 2.1 представлена діаграма варіантів виконання застосунку MONITOR.

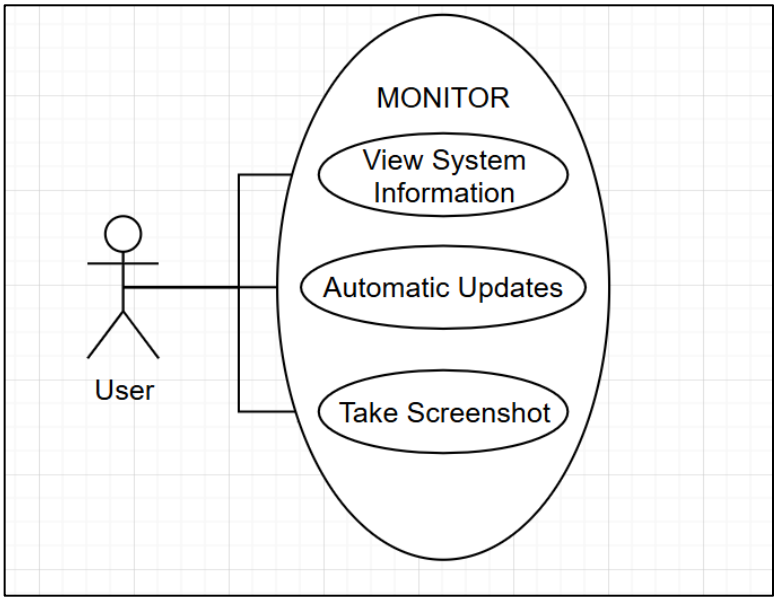


Рисунок 2.1 – Діаграма варіантів використання

У таблиці 2.1 наведено опис елементів та їх тип з діаграми варіантів використання.

Таблиця 2.1 – Опис елементів та їх тип

№	Сценарій (Use Case)	Актор	Опис дії
1	View System Information	User	Користувач переглядає системну інформацію: модель ПК, CPU, RAM, IP, MAC тощо.
2	Automatic Updates	User	Користувач автоматично отримує оновлення застосунку через сервер.
3	Take Screenshot	User	Користувач викликає функцію створення скриншота з підключених моніторів.

Ця таблиця демонструє ключову функціональність, доступну кінцевому користувачу. Основні сценарії: перегляд системної інформації, автоматичні оновлення та створення скриншотів. Отже, вони охоплюють базові потреби в адмініструванні та моніторингу пристроїв, а також ці сценарії повністю відповідають функціональним вимогам системи та відображають логіку взаємодії з користувачем.

2.3.2 Sequence Diagram

Діаграми послідовності (Sequence Diagram) – це діаграми взаємодії, які детально описують, як виконуються операції. Вони відображають взаємодію між об’єктами в контексті спільної роботи.

Діаграми послідовності зосереджені на часовому аспекті та наочно показують порядок взаємодії: які повідомлення передаються і коли. Вертикальна вісь діаграми використовується для представлення часу [13].

На рисунку 2.2 представлена діаграма послідовності застосунку MONITOR.

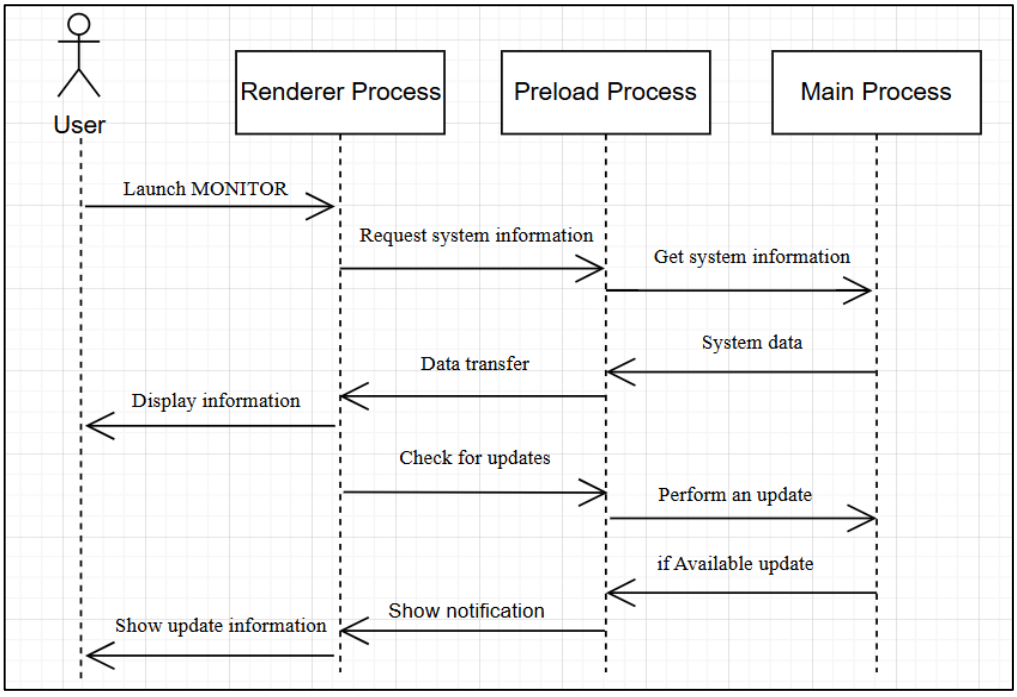


Рисунок 2.2 – Діаграма послідовності

У таблиці 2.2 наведено опис елементів та їх тип з діаграми послідовності.

Таблиця 2.2 – Опис елементів та їх дії

№	**Етап взаємодії**	**Учасник (джерело)**	**Учасник (одержувач)**	**Опис дії**
1	Launch MONITOR	User	Renderer Process	Користувач запускає застосунок MONITOR.
2	Request system information	Renderer Process	Preload Process	Рендер-процес надсилає запит на отримання системної інформації.
3	Get system information	Preload Process	Main Process	Preload передає запит до головного процесу.
4	System data	Main Process	Preload Process	Головний процес повертає системну інформацію (CPU, IP, модель тощо).
5	Data transfer	Preload Process	Renderer Process	Передача зібраної інформації назад до інтерфейсу.
6	Display information	Renderer Process	User	Рендер показує інформацію користувачу.
7	Check for updates	Renderer Process	Preload Process	Рендер ініціює перевірку наявності оновлень.
8	Perform an update	Preload Process	Main Process	Preload передає команду перевірки до головного процесу.
9	if Available update	Main Process	Preload Process	Якщо оновлення доступне – головний процес сповіщає preload.

Продовження табл. 2.2

№	Етап взаємодії	Учасник (джерело)	Учасник (одержувач)	Опис дії
10	Show notification	Preload Process	Renderer Process	Preload передає сигнал для виводу сповіщення.
11	Show update information	Renderer Process	User	Користувач бачить сповіщення про доступне оновлення.

Ця таблиця відображає чіткий маршрут передачі даних від користувача до системи і назад. Детально описано етапи від запуску застосунку до виведення інформації та перевірки оновлень. Це дозволяє простежити логіку функціонування системи в динаміці та переконатися у коректному розподілі ролей між компонентами програми (Renderer, Preload, Main).

2.3.3 Component Diagram

Діаграма компонентів (Component Diagram**)** – це тип діаграми, що використовується для моделювання фізичних аспектів об’єктно-орієнтованих систем. Вона слугує для візуалізації, специфікації, документування компонентних систем, а також для створення виконуваних систем за допомогою прямого та зворотного проєктування.

По суті, діаграма компонентів є різновидом діаграми класів, яка зосереджена на компонентах системи – незалежних, замінюваних частинах програмного забезпечення, що реалізують певну функціональність. Вона використовується для моделювання статичної структури реалізації системи [14].

На рисунку 2.3 представлена діаграма компонентів застосунку MONITOR.

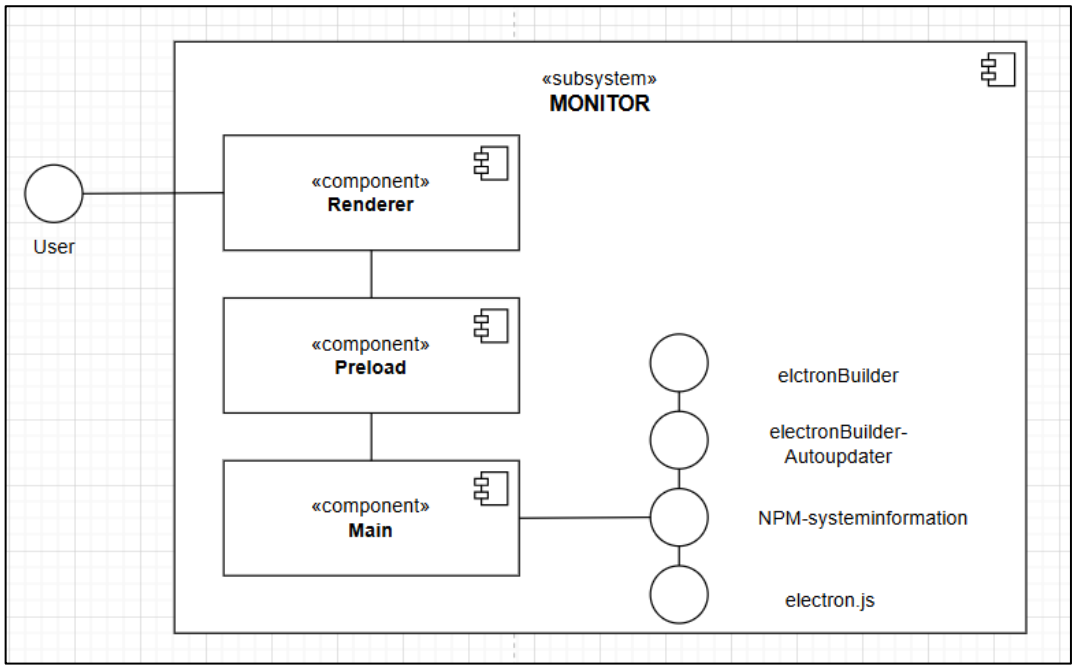


Рисунок 2.3 – Діаграма компонентів

У таблиці 2.3 наведено опис елементів та їх тип з діаграми компонентів.

Таблиця 2.3 – Опис елементів та їх тип

№	Компонент	Тип	Зв’язок	Опис
1	Renderer	<<component>>	Зв’язок з користувачем і preload	Відповідає за інтерфейс користувача. Виводить дані, оновлення, скриншоти.
2	Preload	<<component>>	Міст між Renderer та Main	Забезпечує безпечний обмін даними між рендер-процесом та головним процесом.

Продовження табл. 2.3

№	Компонент	Тип	Зв’язок	Опис
3	Main	<<component>>	Працює з зовнішніми модулями	Обробляє логіку збору даних, оновлення, створення вікон.
4	electron.js	NPM бібліотека	Підключена до Main	Основний фреймворк для побудови застосунку.
5	NPM-systeminformation	Зовнішній модуль	Підключений до Main	Збирає системну інформацію про ПК (CPU, RAM, MAC, IP тощо).
6	electronBuilder	Зовнішній інструмент	Працює з Main	Формує інсталяційний пакет застосунку.
7	electronBuilder-Autoupdater	Зовнішній модуль	Інтегрується в Main	Реалізує перевірку оновлень та їх установку.

Ця таблиця демонструє архітектурну побудову системи MONITOR. Виділено ключові програмні модулі (Renderer, Preload, Main) та їх взаємодію з зовнішніми бібліотеками (systeminformation, electron-updater, electron-builder). Завдяки такій структурі забезпечується модульність, гнучкість та масштабованість застосунку. Це дозволяє легко модернізувати або розширювати функціональність системи в майбутньому.

2.4 Вибір технологій

У процесі розробки програмного забезпечення MONITOR було обрано середовище Visual Studio Code (VS Code), яке є кросплатформеним, безкоштовним, має відкритий код і підтримує широкий спектр технологій, зокрема JavaScript, Node.js та Electron.js.

Цей вибір був зумовлений такими чинниками:

– гнучкість і масштабованість: можливість адаптації під різні типи проєктів та розширення функціоналу за рахунок плагінів;
– інтеграція з системою контролю версій Git: зручна робота з репозиторіями та історією змін;
– підтримка Electron та Node.js API: завдяки плагінам, середовище надає автодоповнення, інтегрований дебаг, live preview тощо;
– наявність розширень ESLint, Prettier, Electron Snippets: забезпечують автоматичне форматування та перевірку якості коду;
– можливість запуску та налагодження проєкту напряму з редактора: підтримка вбудованого терміналу, інтерактивної консолі, швидких сценаріїв.

Структура проєкту реалізована у вигляді типового Node.js-застосунку й її можна побачити у таблиці 2.4.

Таблиця 2.4 – Файли/катологи та їх призначення

Файл / Каталог	**Призначення**
main.js	Головний процес: керування вікнами, IPC-зв’язок, auto-updater
renderer.js	Відображення інтерфейсу, обробка подій користувача
preload.js	Міст між рендер-процесом та головним процесом через contextBridge
one.html	Основна розмітка вікна з даними

Продовження табл. 2.4

Файл / Каталог	**Призначення**
styles.css	Стилізація інтерфейсу (кольори, шрифти, структурування виводу)
package.json	Конфігурація залежностей, сценарії запуску, метадані про застосунок
dist/	Папка з фінальною збіркою застосунку (.exe, .deb та інші формати дистрибуції)

Таким чином, вибір VS Code як основного інструменту розробки забезпечив комфортне середовище для написання, тестування та налагодження коду, що суттєво підвищило ефективність розробки та якість кінцевого продукту.

2.5 Висновки до розділу 2

У цьому розділі було проведено комплексне проєктування програмного забезпечення MONITOR, що дозволило сформувати чітке бачення архітектури, функціональності та технологічної основи майбутнього застосунку.

На етапі визначення вимог було сформульовано ключові функціональні та нефункціональні параметри, серед яких – збір системної інформації, фоновий режим роботи, автоматичне оновлення, кросплатформеність, енергоефективність і безпечна IPC-комунікація між процесами.

Подальше формування архітектури застосунку на базі Electron.js забезпечило чітке розділення на головний процес, рендер-процес та міст preload.js, що відповідає сучасним підходам до десктопної розробки. Було підібрано актуальні технології та інструменти, включаючи Electron, JavaScript, systeminformation, electron-updater та electron-log. Всі вони

органічно інтегруються в проєктну структуру та підтримуються обраним середовищем – Visual Studio Code.

Окрему увагу приділено вибору середовища розробки – Visual Studio Code. Завдяки його функціоналу (підтримка розширень, інтеграція з Git, відлагодження IPC-комунікації) були створені зручні умови для повноцінної роботи над проєктом.

Також, для кращої візуалізації логіки роботи застосунку, структури та взаємодії компонентів було розроблено UML-діаграми, зокрема:

– діаграму варіантів використання (Use Case) – для демонстрації взаємодії користувача із системою;
– діаграму послідовності (Sequence) – для відображення передачі запитів між процесами;
– діаграму компонентів (Component) – для структурування логіки застосунку та зовнішніх залежностей.

Ці діаграми дозволили формалізувати ключові процеси розробки та краще спроєктувати структуру майбутньої системи.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Структура проєкту

Програмне забезпечення MONITOR реалізоване як десктопний застосунок на базі фреймворку Electron.js. Основна структура зображена на рисунку 3.1 та побудована за принципом поділу на головний процес (main.js), інтерфейсний (renderer.js) та міст між ними (preload.js).

```
MONITOR
│
├── binaryAssets
├── dist
│   └── win-unpacked
│       ├── locales
│       └── resources
├── node_modules
├── forge.config.js
├── main.js
├── one.html
├── package.json
├── package-lock.json
├── preload.js
├── renderer.js
├── styles.css
├── tray.png
└── two.html
```

Рисунок 3.1 – Структура каталогів проєкту

Це забезпечує як надійність, так і безпечну взаємодію між логікою та виведенням даних. Тобто, preload-процес розроблений задля безпеки й є посередником між renderer-процесом та main-процесом, який оброблює інформацію й передає її з renderer.js до main.js і навпаки.

3.2 Реалізація головного процесу (main.js)

3.2.1 Імпорт бібліотек

На початку головного файлу main.js відбувається імпорт усіх необхідних бібліотек, які забезпечують роботу застосунку MONITOR. Цей процес зображено на рисунку 3.2. Імпортовані бібліотеки охоплюють як базовий функціонал Electron, так і зовнішні рішення для збору даних та оновлення.

```
//All imported bibliothek
const {autoUpdater, AppUpdater} = require("electron-updater");
const { app, Tray, nativeImage, Menu, BrowserWindow, ipcMain, screen, desktopCapturer,
dialog, BaseWindow } = require('electron');
const path = require('node:path');
const sys = require('systeminformation');
```

Рисунок 3.2 – Імпорт бібліотек

Розглянемо детельно всі бібліотеки.

Electron-updater – зовнішня бібліотека, яка реалізує механізм автоматичного оновлення застосунку. Об’єкт autoUpdater використовується для перевірки оновлень, завантаження та встановлення нових версій.

Модулі electron:

- app – контролює життєвий цикл застосунку (ініціалізація, закриття тощо);
- BrowserWindow – створює вікна з HTML-інтерфейсом;
- Tray, nativeImage, Menu – реалізують системне трей-меню;
- ipcMain – забезпечує міжпроцесну комунікацію між головним і рендер-процесами;
- screen – дозволяє отримувати дані про всі підключені монітори;
- desktopCapturer – використовується для створення скриншотів з екрану;
- dialog – дає змогу відкривати системні діалогові вікна.

Path – вбудований модуль Node.js для роботи з файловими шляхами. Його використання гарантує коректну побудову шляхів незалежно від операційної системи.

Systeminformation (sys) – NPM-бібліотека для збору системної інформації: про процесор, оперативну пам’ять, мережу, диски, температуру, Wi-Fi та інше.

Таким чином, на основі імпортованих бібліотек формуються всі ключові функції застосунку від виводу інформації до обробки оновлень і скриншотів, що дозволяє реалізувати MONITOR як повноцінну багатокомпонентну систему.

3.2.2 Функція createFirstWindow()

У реалізації застосунку MONITOR передбачено створення окремих вікон для відображення інформації. Основним є перше вікно, яке завантажує інтерфейс з файлу one.html і відображає ключові параметри системи. Для його створення розроблена функція createFirstWindow(). Цей процес зображено на рисунку 3.3.

Можна зазначити, що у функції розроблено наступне:

– firstWin = new BrowserWindow(...) для створення нового вікна за допомогою API Electron. Вікно має фіксований розмір 800×600 пікселів;
– webPreferences.preload передбачає підключення скрипта preload.js, що реалізує безпечну комунікацію між головним процесом та інтерфейсом;
– removeMenu() вимикає стандартне верхнє меню, що забезпечує більш компактний та мінімалістичний вигляд вікна;
– loadFile('one.html') завантажує HTML-сторінку, яка слугує інтерфейсом користувача;

– center() розміщує вікно по центру екрану;
– n = 1 опоміжна змінна, яка фіксує, що активним є саме перше вікно;
– Menu.setApplicationMenu(contextMenu) застосовує попередньо створене контекстне меню до цього вікна (меню дублює функції системного трею).

```
// variables for all win an functions and automotation
let firstWin;
let secondWin;
let n = 0;
let tray;

//first window wih system info
function createFirstWindow()
{
   firstWin = new BrowserWindow({
      width: 800,
      height: 600,
      webPreferences:
      {
         preload: path.join(__dirname, 'preload.js')
      },
   });
   firstWin.removeMenu();
   firstWin.loadFile('one.html');
   firstWin.center();
   n = 1;

   Menu.setApplicationMenu(contextMenu);
}
```

Рисунок 3.3 – Головне меню з системною інформацією

Таким чином, функція createFirstWindow() є основним інструментом створення GUI вікна для перегляду системної інформації користувачем. Вона інтегрована з preload-логікою, дозволяє безпечно отримувати дані з головного процесу, і повністю відповідає принципам архітектури Electron.

3.2.3 Запуск застосунку

Для коректного запуску застосунку MONITOR після ініціалізації модуля app використовується подія whenReady(), яка зображена на рисунку 3.4. Ця подія гарантує, що середовище Electron повністю підготоване до створення вікон і обробки подій.

```
//start a web app
app.whenReady().then(() => {

   createFirstWindow();
      firstWin.hide();
      n = 4;

   const icon = nativeImage.createFromPath('tray.png');
   tray = new Tray(icon);

   tray.setToolTip('Open a Monitor App');
   tray.setContextMenu(contextMenu);

  autoUpdater.checkForUpdatesAndNotify();
```

Рисунок 3.4 – Запуск застосунку

Можна зазначити, що було розроблено наступне:

– app.whenReady() – це подія, яка сигналізує, що Electron повністю готовий до роботи (наприклад, рендерінгу вікон);
– createFirstWindow() виклик функції, що створює головне вікно з інтерфейсом;
– firstWin.hide() після створення вікно ховається, щоб застосунок стартував у фоновому режимі, а не одразу показував інтерфейс;
– n = 4 – це допоміжна логічна змінна для визначення стану вікна (наприклад, приховане);
– nativeImage.createFromPath('tray.png') для завантаження іконки для системного трею;

– tray = new Tray(icon) для створення елемента трею, який дозволяє взаємодіяти з програмою навіть при закритому вікні;
– tray.setToolTip(...) для відображення підказки при наведенні на іконку трею;
– tray.setContextMenu(contextMenu) для прикріплення контекстного меню, яке дозволяє керувати програмою через іконку в треї;
– autoUpdater.checkForUpdatesAndNotify() для ініціалізація перевірки оновлень одразу після запуску.

Такий підхід до запуску програми дозволяє реалізувати безперервну роботу у фоновому режимі, з мінімальним втручанням з боку користувача. Застосунок не нав’язливо показується, але залишається доступним через системний трей.

3.2.4 Завершення роботи застосунку

Для правильного завершення роботи застосунку MONITOR у випадках, коли всі вікна були закриті користувачем, використовується подія window-all-closed, яка є стандартною в Electron для обробки такого сценарію. Цей процес зображено на рисунку 3.5.

```
//logic for close the web app
app.on('window-all-closed', function ()
{
   if (process.platform !== 'darwin') app.quit();
})
```

Рисунок 3.5 – Закриття застосунку

Можна зазначити, що було розроблено наступне:

– app.on('window-all-closed', ...) є обробником глобальної події, яка викликається, коли користувач закриває всі відкриті вікна застосунку;

– if (process.platform !== 'darwin') використовується як перевірка операційної системи. У даній реалізації вона дозволяє продовжити завершення застосунку лише для платформ, що його підтримують;
– app.quit() – це метод, який виконує повне завершення роботи застосунку.

Така реалізація забезпечує зручну й передбачувану поведінку: після закриття всіх вікон MONITOR автоматично завершує свою роботу, що відповідає очікуванням користувачів у більшості середовищ.

3.3 Реалізація preload.js

Відповідно до архітектури Electron, для безпечної передачі даних між головним процесом (main.js) і рендер-процесом (renderer.js) застосовується preload-процес (preload.js). У проєкті MONITOR цей процес реалізує міст на базі contextBridge, який дозволяє надати користувачу доступ лише до певного функціоналу. Цей процес зображено на рисунку 3.6.

```
contextBridge.exposeInMainWorld('systemInfo', {
  getSystemInfo1:  async () => await ipcRenderer.invoke('get-system-info1'),
  getSystemInfo2: async () => await ipcRenderer.invoke('get-system-info2'),
  getInfoMonitor: async () => await ipcRenderer.invoke('get-system-info-about-monitor'),
});
```

Рисунок 3.6 – Запит в main для отримання системної інформації

Можна зазначити, що було розроблено наступне:

– contextBridge.exposeInMainWorld(...) – метод, що дозволяє відкрити обмежений API в глобальному об’єкті window у рендер-процесі. Це забезпечує безпечну взаємодію, не даючи прямого доступу до всього Node.js API;

– getSystemInfo1 – функція, яка викликає ipcRenderer.invoke(...) і надсилає запит до головного процесу на отримання статичної системної інформації, такої як назва пристрою, версія ОС, MAC-адреса, серійний номер тощо;
– getSystemInfo2 – функція для отримання динамічної інформації, що змінюється з часом: температура CPU, навантаження, доступна пам’ять;
– getInfoMonitor – запит до головного процесу, який повертає дані про підключені монітори, включаючи роздільну здатність, тип підключення, серійний номер і чи є монітор основним.

Така реалізація preload-процеса дозволяє надійно ізолювати інтерфейсну частину застосунку від системних API, дотримуючись сучасних вимог до безпеки в Electron.

3.4 Взаємодія з бібліотекою systeminformation для збору системних даних

3.4.1 Обробка незмінної системної інформації

Для того щоб інтерфейс користувача міг отримувати актуальну інформацію про стан системи, у головному процесі використовуються асинхронні обробники викликів IPC (ipcMain.handle(...)), яку зображено на рисунку 3.7. Вони реагують на запити, які надсилає рендер-процес через preload.js.

Цей обробник збирає системну інформацію, яка змінюється рідко або є статичною. Наприклад, назва пристрою, версія ОС, серійний номер, IP-адреси та MAC-адреси. Дані збираються за допомогою окремих асинхронних функцій, які викликають методи з бібліотеки systeminformation.

Результат повертається у вигляді об’єкта systemInfo та надсилається назад у рендер-процес.

```
//Data which do not change in time and only once called
ipcMain.handle('get-system-info1', async () =>
{
    const  systemInfo =
        {

            'Information about machine: ': '',
            'Machine Name': (await machineName()) || "N/A",
            'Manufacurer': (await manufacturer()) || "N/A",
            'Machine model': (await machineModel()) || "N/A",
            'Serial Number from Machine': ( await sNFM()) || "N/A",
            'OS Version 1': (await osVerOne()) || "N/A",
            'OS Version 2' : (await osVerTwo()) || "N/A",
            'Current Time': (await curTime()) || "N/A",
            'OS Time': (await osTime()) || "N/A",
            'MAC-Adress LAN-SSH': (await  macAderessLAN_SSH(0)) || "N/A",
            'MAC Adress WIFI-SSH': (await macAdressWIFI_SSH(0)) || "N/A",
            'Intern IP LAN/WIFI IP 4': ( await internIP4(0)) || "N/A",
            'Intern IP LAN/WIFI IP 6': ( await internIP6(0)) || "N/A",
            'Exrern IP LAN/WIFI IP 4 ': (await internIP4(1)) || "N/A",
            'Exrern IP LAN/WIFI IP 6':  ( await internIP6(1)) || "N/A",
            'SSID': (await ssID()) || "N/A",
            'WIFI Signal level 1': (await wifiSigLevel1()) || "N/A",
            'WIFI Signal level 2' : (await wifiSigLevel2()) || "N/A",
            'WIFI Transfer Rate ' : (await transferRate()) || "N/A",
        };

    console.log(systemInfo);
    return  systemInfo;
});
```

Рисунок 3.7 – Обробка незмінної системної інформації

3.4.2 Динамічна інформація про систему

Обробку та повернення динамічної інформації про систему реалізовано за допомогою обробника подій, який зображено на рисунку 3.8.

Цей обробник запитує динамічну, тобто змінну у реальному часі, інформацію: температуру процесора, поточне навантаження системи, об’єм вільної пам’яті.

```
//Change data with timer
  ipcMain.handle('get-system-info2', async () =>
    {
       const systemInfo =
       {
          'CPU Temperature': (await cpuTem()) || "N/A",
          'Curent Load': (await curLoad()) || "N/A",
          'Free memory ': (await freeMem(0))  || "N/A"
       };
       return systemInfo;
    });
```

Рисунок 3.8 – Динамічна інформація про систему

Ці дані можуть оновлюватися автоматично за таймером у рендері (через setInterval) кожні 5 хвилин.

3.4.3 Інформація про підключені монітори

На рисунку 3.9 зображено запит інформації про підключені монітори, а саме визначається кількість дисплеїв та формується відповідний об’єкт із даними.

```
//change data with timer
ipcMain.handle('get-system-info-about-monitor', async () =>
{
   const nMonitor = screen.getAllDisplays().length;
   if(nMonitor > 1)
   {
      const systemInfo =
      {
         '1 Monitor' : (await monitorInfo(0))  || "N/A",
         '2 Monitor' :  (await monitorInfo(1)) || "N/A",
         '3 Monitor' :  (await monitorInfo(2)) || "N/A",
         '4 Monitor' :  (await monitorInfo(3)) || "N/A"
      };
      return systemInfo;
   }
   else
```

```
  {
    const systemInfo =
    {
      '1 Monitor' : (await monitorInfo(0))  || "N/A",
    };
    return systemInfo;
  }
});
```

Рисунок 3.9 – Інформація про підключені монітори

Цей блок обробляє запит на отримання даних про монітори. Визначається кількість дисплеїв за допомогою модуля screen, після чого для кожного з них викликається функція monitorInfo(n), яка повертає роздільну здатність, тип підключення, серійний номер, статус «основного» тощо.

3.5 Реалізація механізму авто оновлення

Для зручності користувача у застосунку MONITOR реалізовано автоматичне оновлення. Цей проуес зображено на рисунку 3.10.

```
contextBridge.exposeInMainWorld('updateMessage', {
  checkUpdate: (callback) => ipcRenderer.on('checking-update', callback),
  updateAvailable: (callback) => ipcRenderer.on('update-available', (event, info1) =>
callback(info1)),
  noUpdate: (callback) => ipcRenderer.on('update-not-available', (event, info2) =>
callback(info2)),
  updatErr: (callback) => ipcRenderer.on('update-error', (event, error) => callback(error)),
  downloadProgress: (callback) => ipcRenderer.on('download-progress', (event, progressObj)
=> callback(progressObj)),
  updateOnDownload: (callback) => ipcRenderer.on('update-downloaded', (event, info3) =>
callback(info3))
 });
```

Рисунок 3.10 – Автоматичне оновлення

Функціонал побудований на основі бібліотеки electron-updater, а передача подій до інтерфейсу користувача здійснюється через IPC-зв’язок за допомогою preload.js.

У preload-процесі реалізовано API updateMessage, який передає оновлення від головного процесу до рендер-процесу:

– checkUpdate – відстежує подію початку перевірки наявності оновлень;
– updateAvailable – реагує на виявлення нової версії;
– noUpdate – повідомляє, якщо оновлення не знайдено;
– updatErr – обробляє помилки оновлення (наприклад, відсутність інтернету);
– downloadProgress – передає прогрес завантаження у відсотках;
– updateOnDownload – сигналізує, що оновлення готове до встановлення.

Також, на рисунку 3.11 зображено додатковий реалізований метод quitAndInstall, який ініціює перезапуск застосунку для завершення оновлення.

```
  contextBridge.exposeInMainWorld('api', {
  quitAndInstall: () => ipcRenderer.send('install-update'),
});
```

Рисунок 3.11 – Перезапуск застосунку

Така архітектура дозволяє реалізувати повністю автоматизовану систему оновлень, при цьому користувач бачить лише результат, тобто повідомлення про доступне оновлення та сам результат автовновлення. А саме коли користувач закриває застоснок і потім відкриває його, то застосунок відкривається вже у новії версії.

3.6 Інтерфейс користувача (HTML + CSS + renderer.js)

Графічний інтерфейс користувача у програмі MONITOR реалізований за допомогою HTML, CSS і JavaScript-файлу renderer.js, що зображено на рисунку 3.12. Основною сторінкою є one.html, яка відображає всі ключові системні показники в зручному форматі.

```
<body>
   <h1>MONITOR System Information App</h1>
   <p id = "info-Update"></p>
   <p id = "info-machine1"></p>
   <p id = "info-machine2"></p>
   <p id = "info-machine3"></p>
   <p id = "info-Screenshot"></p>
   <script src="./renderer.js">
      require = "./rendere.js";
   </script>
</body>
```

Рисунок 3.12 – Ключові системні показник

Далі розглянемо детально реалізацію коду:

а) елемент <h1> – заголовок застосунку;

б) п’ять блоків <p> з унікальними id призначені для виведення:

1) info-machine1 – статична системна інформація;
2) info-machine2 – динамічні дані (CPU, RAM);
3) info-machine3 – інформація про монітори;
4) info-Update – статус оновлення;
5) info-Screenshot – результат скриншотів.

На рисунку 3.13 зображено реалізоване CSS-оформлення (styles.css).

```
h1{
   color: orange;
}
```

Рисунок 3.13 – CSS-оформлення

Отже, використовується стиль мінімалістичного дизайну, який підкреслює технічність, monospace дозволяє красиво відображати форматовані JSON-дані, а також pre-wrap дозволяє зберігати й переноси рядків при форматуванні JSON.stringify(...).

Вивід інформації в інтерфейсі реалізовано через асинхронні виклики до preload.js. На рисунку 3.14 зображено основні блоки.

```
const info = await window.systemInfo.getSystemInfo1();
        getInfo1.innerText = JSON.stringify(info, null, 2);
        console.log("First Block: " + getInfo1.innerText);

        const infoWithChange = await window.systemInfo.getSystemInfo2();
        getInfo2.innerText = JSON.stringify(infoWithChange, null, 2);
        console.log( "Second Block: " + getInfo2.innerText);

        const infoWithChange2 = await window.systemInfo.getInfoMonitor();
        getInfo3.innerText = JSON.stringify(infoWithChange2, null, 2);
        console.log("Third Block: " + getInfo3.innerText);

        const info2 = await window.screenshot.capturescreenshot();
        getScreenshot4.innerText = JSON.stringify(info2, null, 2);
        console.log("Screen Block: " + getScreenshot4.innerText);
```

Рисунок 3.14 – Асинхронні виклики

Таким чином, інтерфейс MONITOR є простим, легким і достатньо інформативним. Користувач може одразу бачити ключові параметри свого комп’ютера в зручному вигляді, без зайвих кліків.

3.7 Створення скриншотів екрана

У застосунку MONITOR реалізовано можливість знімати скриншоти екранів користувача. Для цього використовується вбудований модуль Electron desktopCapturer, який надає API для захоплення зображення з усіх активних дисплеїв.

Основна функція, яка відповідає за створення скриншота, має назву captureScreen(n), де n – індекс монітора, з якого потрібно зробити знімок. Цей процес зображена на рисунку 3.15.

```
//screenshot function
async function captureScreen(n)
{
   //take displays
   const pDisplay = screen.getAllDisplays();
   //select display for the future screenshot
   const selectedDisplay = pDisplay[n];
   //check the size width and height
   const {width, height} = selectedDisplay.size;
   const options = {
      types: ['screen'],
      thumbnailSize: {width, height},
   };
   //do the screenshot and save in PNG format than translate in base64 or binary
   const source = await desktopCapturer.getSources(options);
   const primarySource = source.find(({display_id}) => display_id == selectedDisplay.id);
   const image = primarySource.thumbnail;
   const pngBuffer = image.toPNG();
   const ImgBase = pngBuffer.toString('base64');
   const jsonCreate = ImgBase;
     return jsonCreate;
}
```

Рисунок 3.15 – Функція створення скриншота

Далі розглянемо логіку реалізації коду:

– screen.getAllDisplays() – визначає всі активні екрани, підключені до системи;
– desktopCapturer.getSources(...) – повертає зображення (мініатюри) з кожного екрана;
– зображення конвертується у **PNG** та кодується у **base64**, що дозволяє відображати його у HTML через тег <img src="...">.

Скриншот у вигляді base64 передається з main.js до renderer.js, де вставляється в DOM або зберігається у змінній. Це дозволяє:

– переглядати скриншоти без потреби зберігати файли;
– обробляти кілька моніторів;
– оновлювати знімки автоматично через інтервал (наприклад, кожні 60 секунд).

Таким чином, функціонал знімків екрану є особливо актуальним для моніторингових систем, що працюють у фоні. Він дозволяє фіксувати реальний стан дисплея або використовувати цю інформацію для подальшої діагностики.

3.8 Tray-меню як елемент навігації

3.8.1 Реалізація Tray-меню

Оскільки застосунок MONITOR працює у фоновому режимі, його основним елементом навігації є іконка в системному треї (Tray). Користувач може взаємодіяти з програмою через Tray-меню, не відкриваючи її вручну після запуску.

На рисунку 3.16 зображено обробник події click по іконці в треї, який відіграє особливо важливу роль.

Далі розглянемо логіку реалізації коду, а саме змінна n, яка використовується як індикатор стану:

– 1 – відкрите вікно з системною інформацією;
– 2 – відкрите інше вікно (наприклад, для реєстрації);
– 3 – активне вікно вже відкрите, але в прихованому режимі;
– 4 – після запуску програма у фоні (без видимого вікна).

Коли користувач натискає на іконку в треї:

– якщо вікно було приховане – показується заново (firstWin.show());

– якщо активне якесь вікно – воно закривається або перезавантажується (перестворюється).

```
tray.on('click', () => {
   switch(n)
   {
      case 1:
      {
         firstWin.close();
         createFirstWindow();
             n = 3;
      };
      break;
      case 2:
      {
         secondWin.close();
         createFirstWindow();
         n = 3;
      };
      break;
      case 3:
      {
         createFirstWindow();
         //firstWin.show();
         n = 3;
      };
      break;
      case 4:
      {
         firstWin.show();
         n = 3;
      };
      break;
   }
 });
});
```

Рисунок 3.16 – Tray-меню

Таким чином, Tray-меню виконує роль універсального перемикача між станами застосунку, забезпечуючи швидкий доступ і зручне керування без зайвих дій.

3.8.2 Результат реалізації Tray-меню

Як результат реалізації Tray-меню на рисунку 3.17 продемонстровано зовнішній вигляд іконки, яка знаходиться в області сповіщень операційної системи (в правому нижньому куті екрана у Windows).

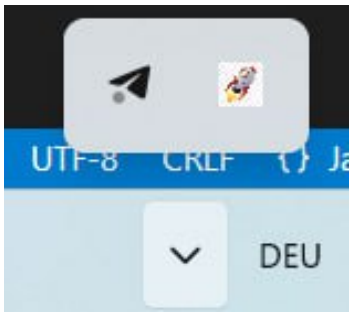


Рисунок 3.17 – Іконка Tray-меню

Після натиснення правою кнопкою мишки по іконці користувч отримує можливість переглянути параметри Tray-меню. Результат цієї дії зображено на рисунку 3.18.

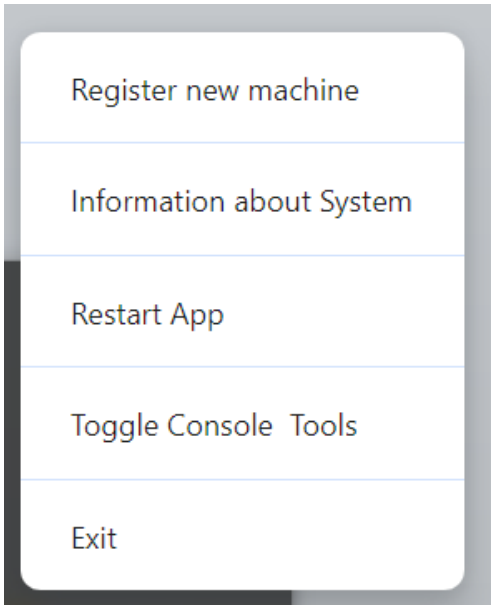


Рисунок 3.18 – Параметри Tray-меню

Після натиснення правої клавіши користувач має можливість обрати один із парамнтів, а саме:

– реєстрація нової машини;

– інформація про систему;

– перезапуск застосунку;

– перемикання інструментів консолі;

– закриття застосунку.

Таким чином, Tray-меню дозволяє залишити застосунок активним у фоновому режимі без відкритого вікна, забезпечуючи зручний доступ до його функцій у будь-який момент.

3.9 Збірка, установка та запуск застосунку

Після реалізації всіх функціональних частин проєкту, наступним етапом є збірка, установка та запуск застосунку. Роблю збірку застосунку в командному рядку прописуючи команду npm run dist, яка збирає програму в дистрибутивну (інсталяційну) версію. Результат команди зображено на рисунку 3.19.

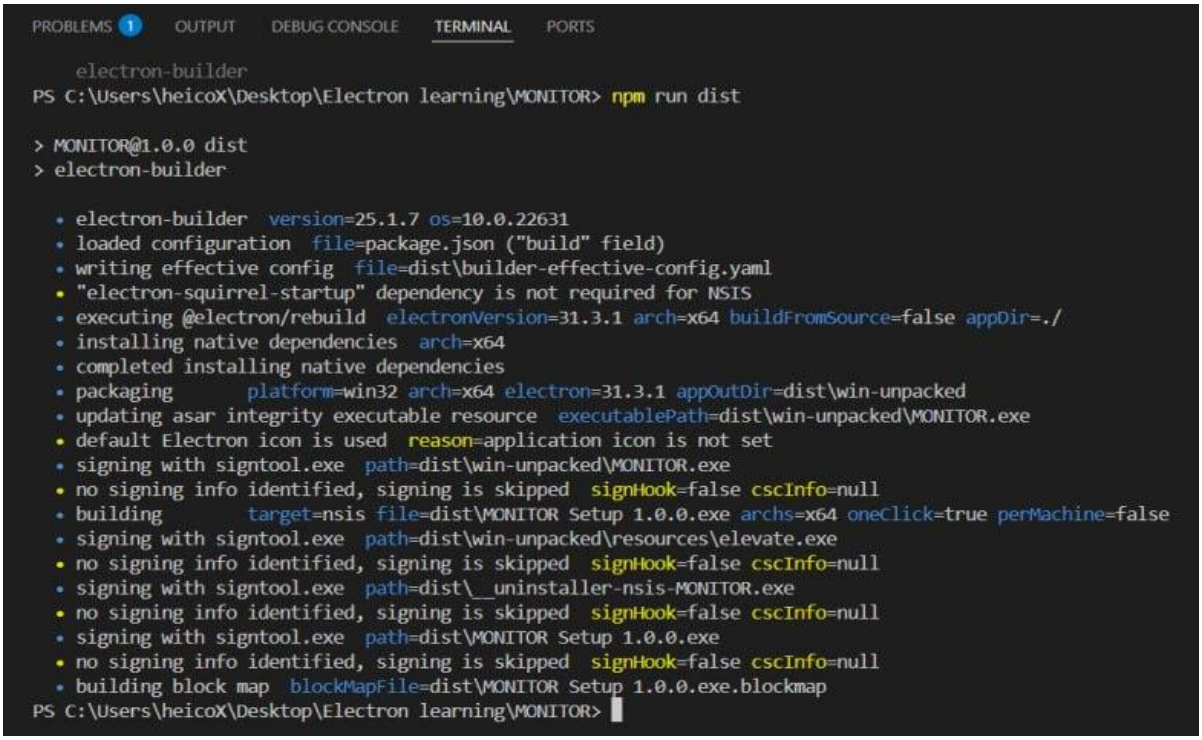


Рисунок 3.19 – Команда збірки застосунку та її результат

Далі як результат прописаної раніше команди npm run dist на рисунку 3.20 продемонстровано вже зібрану програму у форматі exe файлу, який готовий для встановлення програми на комп’ютер.



Рисунок 3.20 – Зібрана програма

Натискаємо на exe файл і починається установка застосунку, яку можна побачити на рисунку 3.21.

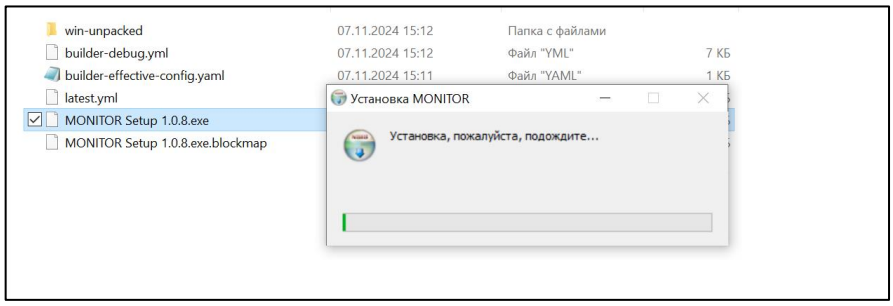


Рисунок 3.21 – Установка за стосунку

Після установки застосунку відкривається програма, яка починає працювати у фоновому режимі і відображається у нижньому правому куті сповіщень операційної системи. Результат продемонстровано на рисунку 3.22.

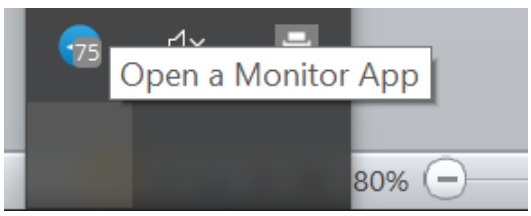


Рисунок 3.22 – Відображення за стосунку

Таким чином, у результаті успішної збірки та встановлення застосунку MONITOR, програма коректно запускається у фоновому режимі, інтегруючись у системний трей операційної системи. Це забезпечує зручний доступ до її функцій без необхідності постійної взаємодії з вікном застосунку.

3.10 Запуск за стосунку

Застосунок MONITOR можна запустити два способам, а саме через командний рядок за допомогою команди npm start, яка ініціалізує виконання головного процесу Electron та відкриває вікно за стосунку. Результат можна побачити на рисунку 3.23. А також, як альтернатива це запуск, який можна здійснити й через ярлик на робочому столі, що є зручним для кінцевого користувача. Результат продемонстровано на рисунку 3.24.

```
PS C:\Users\heicoX\Desktop\Electron learning\MONITOR> npm start
```

Рисунок 3.23 – Запуск з командного рядка



Рисунок 3.24 – Запуск через ярлик

Таким чином, користувач має змогу обрати зручний для себе спосіб запуску: або через термінал для розробки й налагодження, або через графічний інтерфейс після встановлення. Обидва способи гарантують коректну ініціалізацію застосунку та подальше виконання усіх його функціональних модулів.

3.11 Демонстрація роботи та результати

Після запуску застосунку MONITOR, користувач отримує можливість переглядати детальну системну інформацію про пристрій у зручному форматі. На рисунку 3.25 представлено результат відображення таких даних у графічному інтерфейсі.

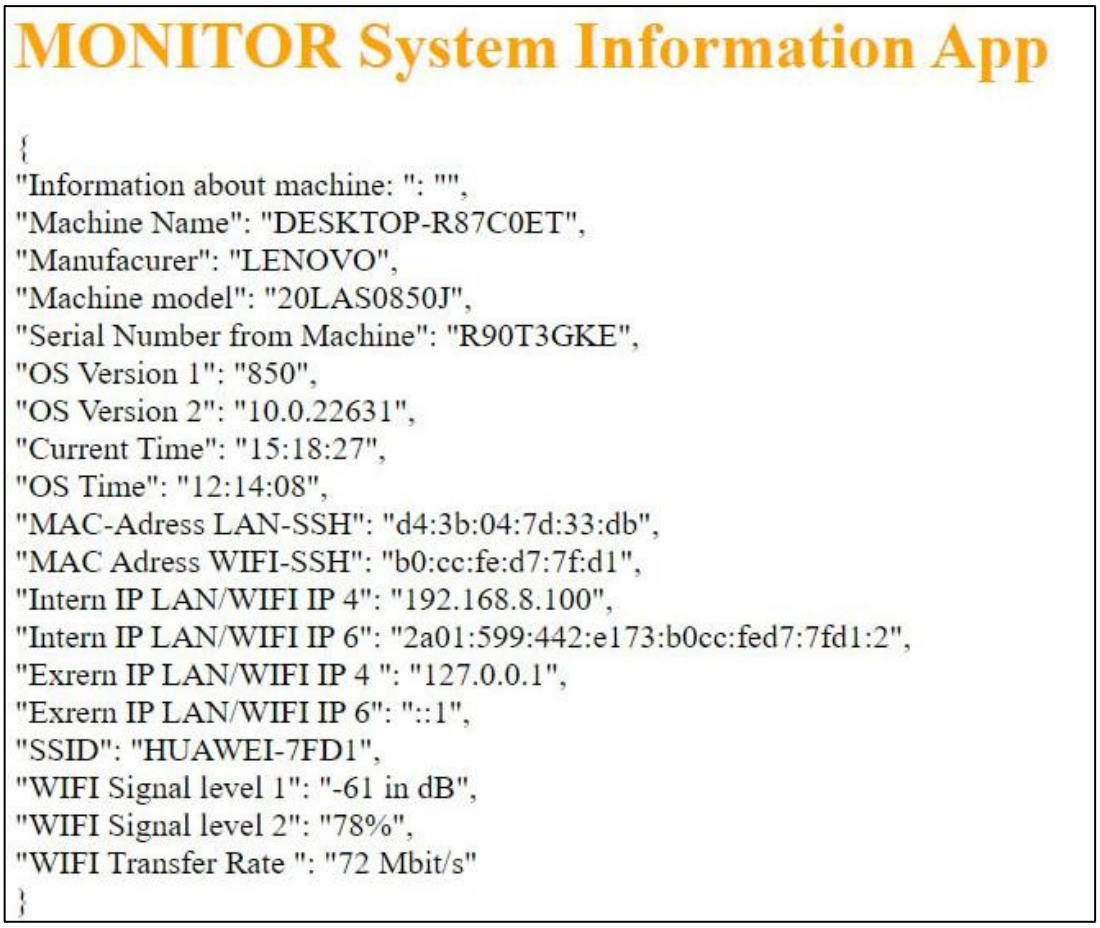


Рисунок 3.25 – Системна інформація

Розглянемо, яка саме інформація вивелась:

а) назва пристрою (Machine Name) – наприклад, "DESKTOP-R87C0ET" – допомагає ідентифікувати систему;

б) виробник і модель пристрою (Manufacturer / Machine model) – "LENOVO", "20LAS0850J";

в) серійний номер пристрою (Serial Number) – корисний для техпідтримки або інвентаризації;

г) інформація про операційну систему, включаючи дві версії та поточний і системний час;

д) мережеві дані:

– MAC-адреси для LAN і Wi-Fi;

– внутрішні та зовнішні IP-адреси (IPv4 / IPv6);

– назва мережі Wi-Fi (SSID), рівень сигналу у dB і %;

– швидкість передачі даних (Transfer Rate).

Далі на рисунках 3.26 та 3.27 продемонстровано результати збору динамічної системної інформації, яка змінюється в режимі реального часу, а також дані про монітор(и)**,** підключені до комп’ютера.


```
{
"CPU Temperature": "N/A",
"Curent Load": 2.9487673515859782,
"Free memory ": 444209901568
}
```


Рисунок 3.26 – Поточні системні ресурси

Розглянемо, яка саме інформація вивелась:

– температура CPU: у даному випадку відсутня ("N/A"), можливо через відсутність відповідного сенсора або обмеження доступу;

– завантаження процесора (Current Load): відображає рівень використання процесора у відсотках (2.95%);

– вільна пам’ять (Free memory): показує обсяг доступної оперативної пам’яті в байтах.


```
{
"1 Monitor": [
"Resolution X 1920 px",
" Current Resolution X 1536 px",
" Resolution Y 1080 px",
" Current Resolution Y 864 px",
" Monitor connection INTERNAL",
" Monitor serial number undefined",
" Main monitor true"
]
}
```


Рисунок 3.27 – Параметри монітора

Розглянемо, яка саме інформація вивелась:

- фізична та поточна роздільна здатність (X/Y);
- тип підключення (INTERNAL – внутрішній монітор ноутбука);
- серійний номер – якщо система його не бачить, виводиться “undefined”;
- позначка основного монітора – “true”, що означає, що саме цей дисплей є головним для системи.

На рисунку 3.28 представлено результат функціональності створення скриншота екрану, яка реалізована в застосунку MONITOR.

{
"1 Monitor Screenshot":
"iVBORw0KGgoAAAANSUhEUgAABgAAAANgCAIAAABlQDTBAAAgAElEQVR4nOzdd3wU5504/s/M7M5sb9
}

Рисунок 3.28 – Створення скриншота екрану

Розглянемо, що саме відображається:

- автоматично захоплене зображення активного дисплею;
- результат збережений у вигляді бінарного рядка (base64), що дозволяє легко інтегрувати його в інтерфейс.

На рисунку 3.29 зображено сайт, де можна декодувати бінарний рядок base64 у вигляд зображення, тобто скриншот, який робить застосунок [15].

У процесі демонстрації було підтверджено, що застосунок MONITOR успішно виконує свою основну функцію таку, як збір і вивід системної інформації. Користувач отримує у зручному форматі повні дані про комп’ютер: назву машини, модель, параметри операційної системи, IP- та MAC-адреси, рівень сигналу Wi-Fi, температуру, навантаження процесора, обсяг вільної пам’яті, а також характеристики підключених моніторів.

Крім того, реалізована можливість автоматичного створення скриншоту активного монітора, що дозволяє зафіксувати поточний стан екрану для подальшого аналізу, звітності або проблем роботи монітору, який

знаходиться віддалено. Це розширює функціональність програми та додає зручності в роботі користувача.

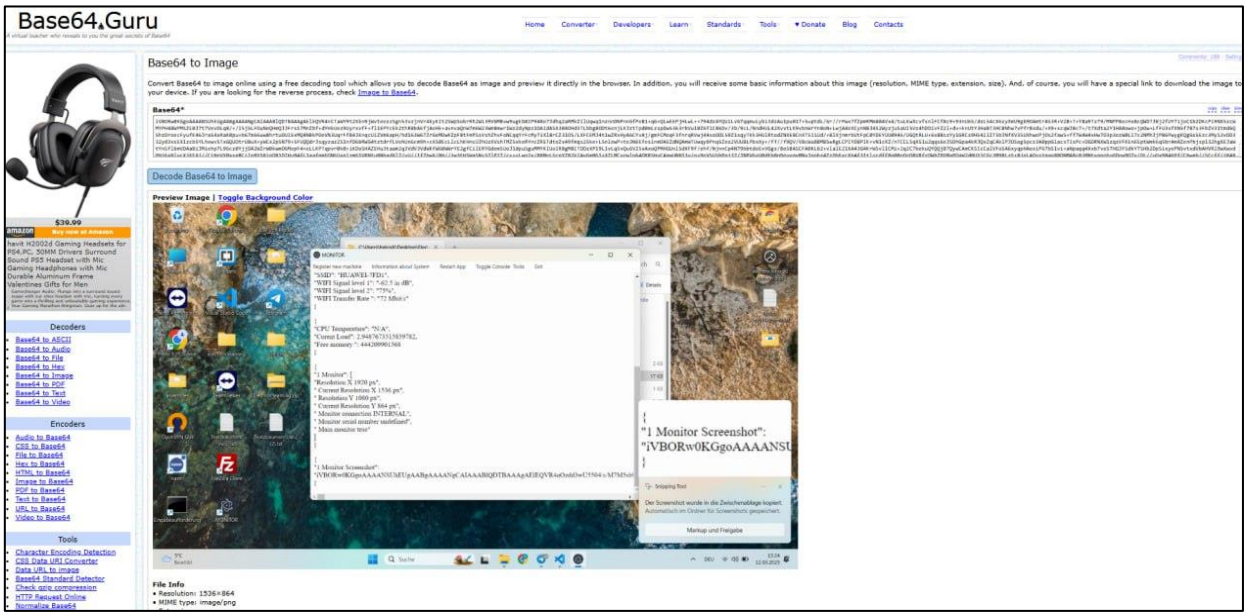


Рисунок 3.29 – Декодування base64 в зображення

Таким чином, застосунок забезпечує наочне, швидке й ефективне отримання повної системної інформації без використання сторонніх утиліт.

3.12 Висновки до розділу

У межах третього розділу було здійснено повну програмну реалізацію десктопного застосунку MONITOR на базі фреймворку Electron.js. Розробку виконано згідно з вимогами, сформованими на етапі проєктування, із дотриманням сучасних принципів модульності, безпеки та зручності користування.

Зокрема, реалізовано:

– головний процес для керування вікнами, системним треєм і логікою IPC;
– preload-скрипт, який забезпечує безпечну взаємодію між інтерфейсом і системним API;

- інтерфейс користувача з HTML, CSS і JavaScript, що відображає системну інформацію в структурованому вигляді;
- функціонал для збору апаратних та мережевих параметрів через бібліотеку systeminformation;
- механізм автоматичного оновлення застосунку через electron-updater;
- створення скриншотів з дисплеїв та можливість відображення їх у інтерфейсі;
- tray-меню для зручної навігації та запуску в фоновому режимі;
- механізм збірки дистрибутива, що дозволяє запускати застосунок як повноцінну інсталяційну програму.

Результати тестування та демонстрації підтвердили стабільну роботу програми: всі модулі взаємодіють коректно, інтерфейс зручний, а дані виводяться точно й своєчасно.

ВИСНОВКИ

У межах дипломної роботи було створено десктопний застосунок MONITOR, який дозволяє ефективно здійснювати збір, обробку та виведення системної інформації користувача в режимі реального часу. Програма автоматично відображає дані про апаратне забезпечення, мережеві параметри, навантаження системи, роботу моніторів та забезпечує створення скриншотів для візуального контролю стану пристрою.

Однією з ключових переваг застосунку є його можливість працювати у фоновому режимі з доступом через системний трей, що робить його зручним і ненав’язливим у повсякденному використанні. Крім того, реалізований механізм автоматичного оновлення дозволяє швидко впроваджувати нові функції або виправлення без участі користувача.

Програма розроблена спеціально для потреб компанії, з урахуванням необхідності віддаленого моніторингу цифрових біл-бордів, які знаходяться у Німеччині та Австрії. Це дозволить IT-відділу оперативно виявляти потенційні проблеми (перегрівання, перевантаження, нестабільне підключення, некоректна інформація тощо.) без фізичного доступу до пристроїв, що значно оптимізує процес технічної підтримки та підвищує стабільність роботи в корпоративному середовищі.

Таки чином, застосунок MONITOR є ефективним, адаптивним та готовим до впровадження у робочі процеси компанії як інструмент системного контролю.

ПЕРЕЛІК ПОСИЛАНЬ

1. Xibo Open Source Digital Signage. URL: https://xibosignage.com/ (дата звернення: 12.03.2025).

2. Screenly – Digital Signage for the Raspberry Pi. URL: https://www.screenly.io/ (дата звернення: 12.03.2025).

3. Electron – Build cross-platform desktop apps with JavaScript, HTML, and CSS. URL: https://www.electronjs.org/ (дата звернення: 17.03.2025).

4. JavaScript documentation – Mozilla Developer Network. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (дата звернення: 17.03.2025).

5. HTML documentation – Mozilla Developer Network. URL: https://developer.mozilla.org/en-US/docs/Web/HTML (дата звернення: 20.03.2025).

6. CSS documentation – Mozilla Developer Network. URL: https://developer.mozilla.org/en-US/docs/Web/CSS (дата звернення: 22.03.2025).

7. systeminformation – Node.js system information library. URL: https://systeminformation.io/ (дата звернення: 24.03.2025).

8. Visual Studio Code – Code editing. Redefined. URL: https://code.visualstudio.com/ (дата звернення: 28.03.2025).

9. Electron-builder – Complete solution to package and build a ready-for-distribution Electron app. URL: https://www.electron.build/ (дата звернення: 03.04.2025).

10. Electron AutoUpdater Electron Documentation**.** URL: https://www.electronjs.org/docs/latest/api/auto-updater (дата звернення: 09.04.2025).

11. Що таке UML- діаграма? URL: https://foxminded.ua/uml-diagramy/ (дата звернення: 11.04.2025).

12. Що таке Use Case діаграма? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/ (дата звернення: 12.04.2025).

13. Що таке Sequence діаграма? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/ (дата звернення: 13.04.2025).

14. Що таке Component діаграма? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/ (дата звернення: 14.04.2025).

15. Декодування base64 в зображення. URL: https://base64.guru/converter/decode/image (дата звернення: 20.04.2025).

ДОДАТОК А

Корисні сервіси та посилання

Сайт інтерактивного навчання JavaScript, HTML, CSS. Містить приклади, редактор коду онлайн, пояснення синтаксису. URL: https://www.w3schools.com/js/ (дата звернення: 14.02.2025).

Офіційний вебсайт Visual Studio Code – середовища розробки, використаного у проєкті. URL: https://code.visualstudio.com/ (дата звернення: 18.03.2024).

Онлайн-редактор для створення UML-діаграм (використовувався для діаграм проєкту). URL: https://www.drawio.com/ (дата звернення: 14.04.2024).

Репозиторій Electron Forge для збирання застосунків. URL: https://www.electronforge.io/ (дата звернення: 20.04.2024).

ДОДАТОК Б

Приклади роботи за стосунку

Для повноти демонстрації роботи застосунку MONITOR планується також показати декілька додаткових прикладів його функціонування:

– при підключенні пристрою до зовнішнього монітора за допомогою HDMI-кабелю;

– під час роботи через віддалене підключення до комп’ютера.

Ці приклади дозволять переконатися, що застосунок правильно збирає та відображає системну інформацію як у випадку фізичного розширення робочого простору, так і за умови віддаленої взаємодії з пристроєм.

Б.1 Приклади роботи за стосунку за допомогою HDMI-кабелю

На рисунку Б.1 показано базову інформацію про комп’ютер: ім’я пристрою, виробник, модель, серійний номер, версії ОС, час, MAC-адреси, IP-адреси, стан Wi-Fi.

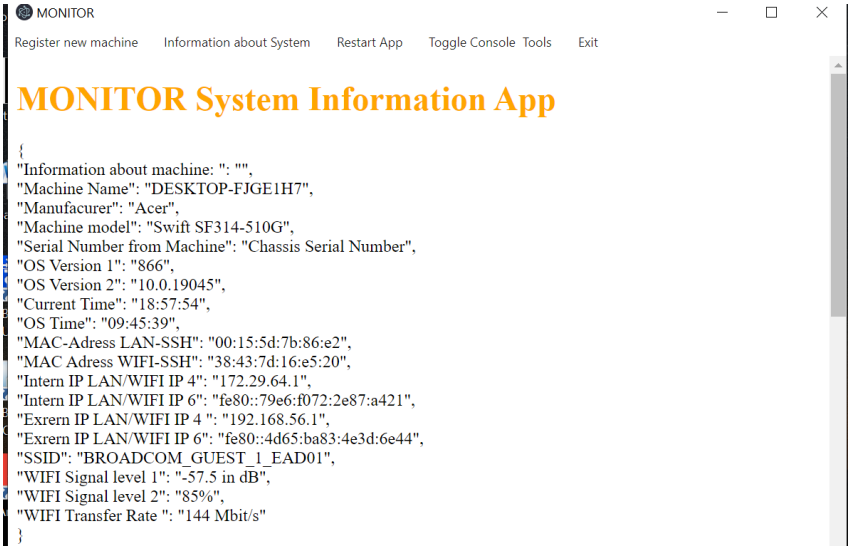


Рисунок Б.1 – Виведення системної інформації на головному вікні застосунку

На рисунку Б.2 показані змінні в реальному часі: завантаження процесора, температура CPU та обсяг доступної оперативної пам’яті.

{
"CPU Temperature": "N/A",
"Curent Load": 2.3997094808436117,
"Free memory ": 28555665408
}

Рисунок Б.2 – Вивід динамічної інформації про систему

На рисунку Б.3 виводяться дані про монітори: роздільна здатність, тип підключення (INTERNAL або HDMI), серійний номер монітора (якщо є) і чи є він основним дисплеєм.

{
"1 Monitor": [
"Resolution X 1280 px",
" Current Resolution X 1280 px",
" Resolution Y 720 px",
" Current Resolution Y 720 px",
" Monitor connection INTERNAL",
" Monitor serial number undefined",
" Main monitor true"
],
"2 Monitor": [
"Resolution X 1280 px",
" Current Resolution X 1280 px",
" Resolution Y 720 px",
" Current Resolution Y 720 px",
" Monitor connection HDMI",
" Monitor serial number undefined",
" Main monitor false"
],
"3 Monitor": "N/A",
"4 Monitor": "N/A"
}

Рисунок Б.3 – Інформація про підключені монітори

На рисунку Б.4 показані отримані скриншоти моніторів у форматі Base64, що можна декодувати для перегляду зображення першого та другого монітора.

{
"1 Monitor Screenshot":
"iVBORw0KGgoAAAANSUhEUgAABQAAAALQCAIAAABAH0oBAAAgAElEQVR4nOydd3wcxfn/P7O7V1VO
"2 Monitor Screenshot":
"iVBORw0KGgoAAAANSUhEUgAABQAAAALQCAIAAABAH0oBAAAgAElEQVR4nEy9V5MkyXU1GB5aR2p
}

Рисунок Б.4 – Виведення скриншотів екрану

На рисунку Б.5 продемонстровано декодування Base64 у зображення, яке показує вміст першого монітора, а саме робочий стіл користувача з ярликами та відкритими вікнами.

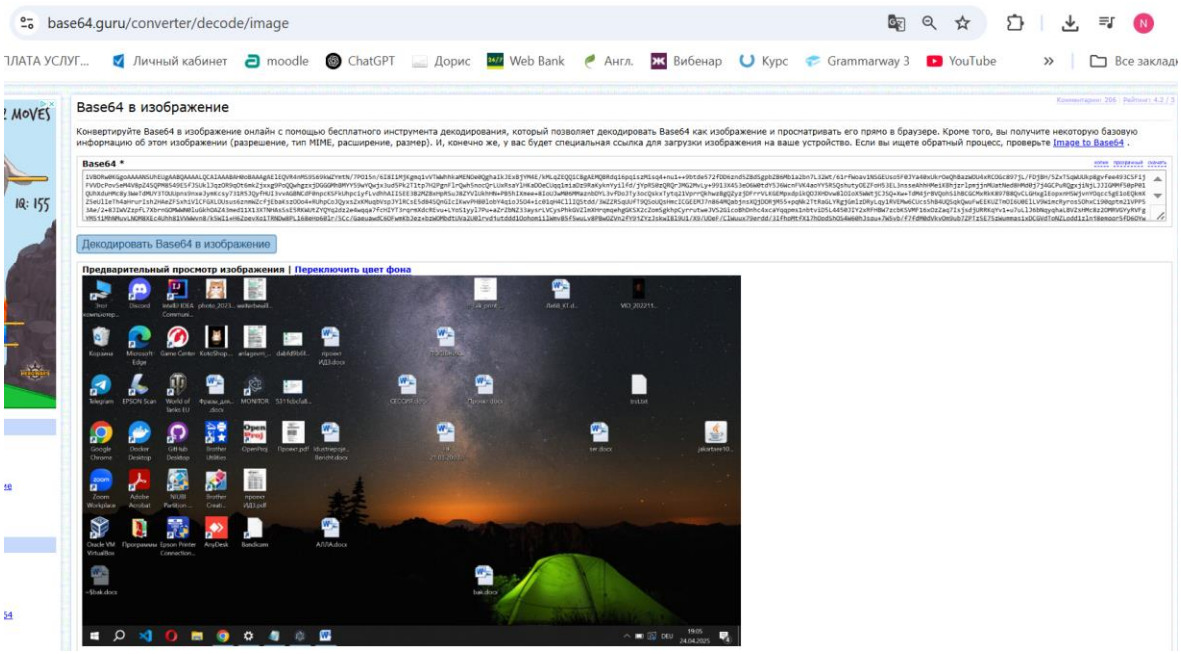


Рисунок Б.5 – Попередній перегляд скриншота першого монітора

На рисунку Б.6 продемонстровано декодування Base64 у зображення, яке показує вміст другого монітора, де відкрито вікно з текстовим документом з текстом “Hello monitor 2!!”.

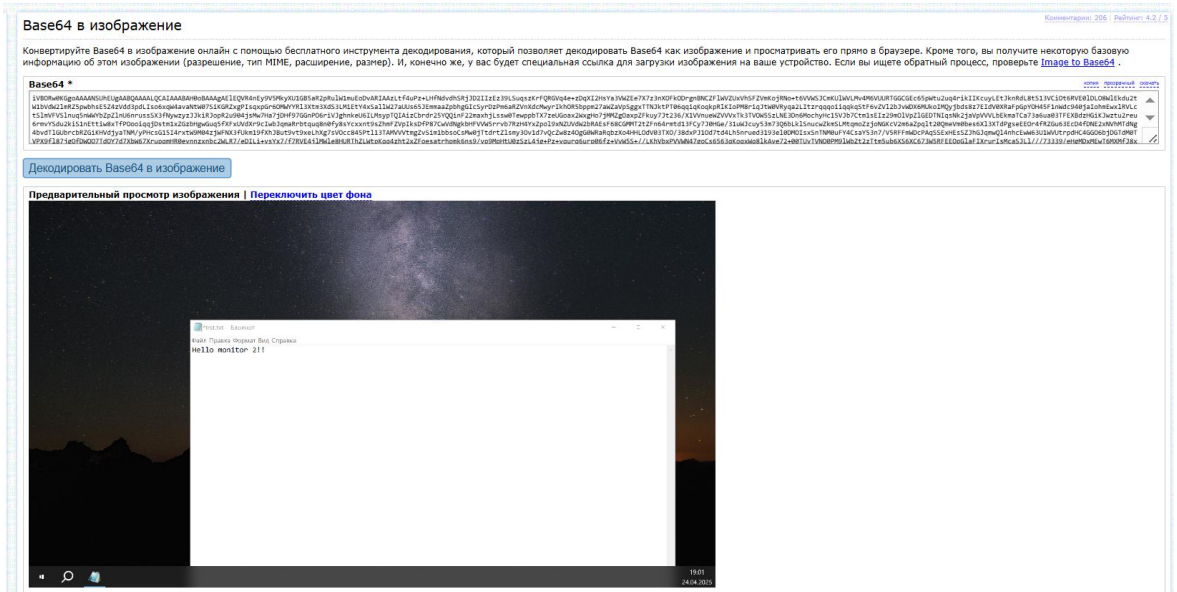


Рисунок Б.6 – Попередній перегляд скриншота другого монітора

Б.2 Приклади роботи за стосунку через віддалене підключення до комп’ютера

На рисунку Б.7 зображено підключення до монітора за допомогою дистанційного програмного забезпечення AnyDesk, щоб зробити перевірку роботи монітора.

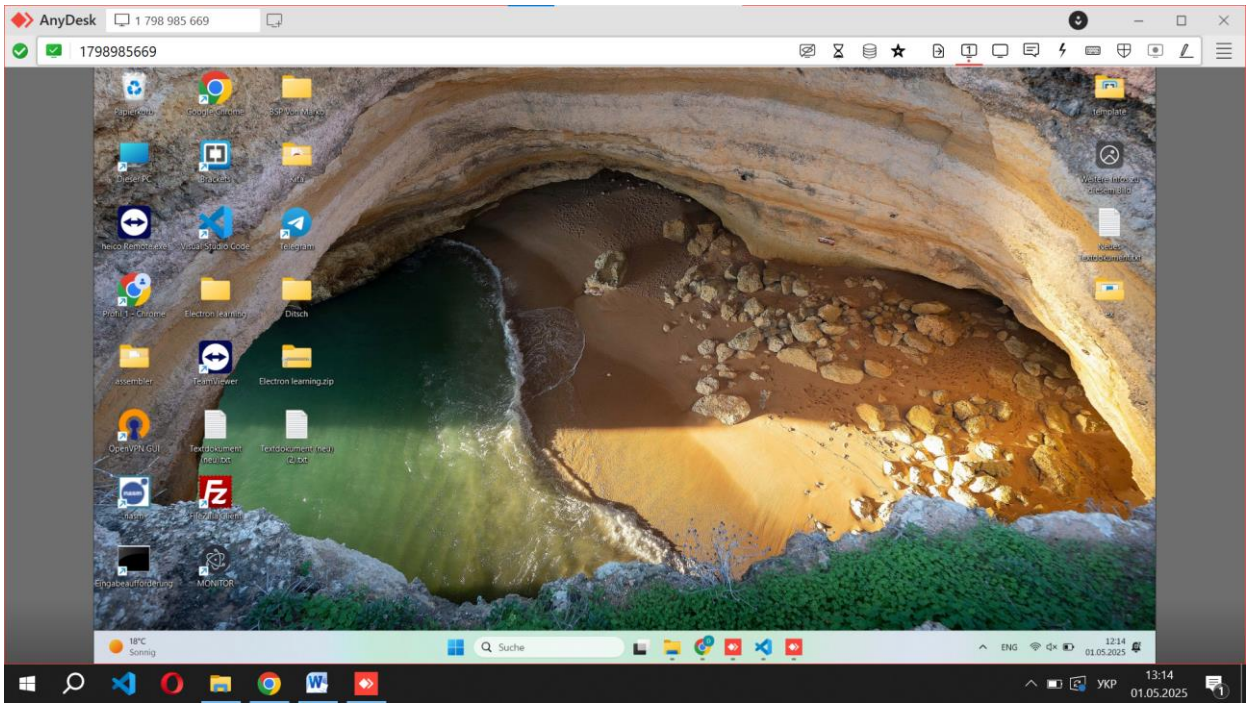


Рисунок Б.7 – Підключення за допомогою ПЗ AnyDesk

На рисунку Б.8 показано базову інформацію про комп’ютер: ім’я пристрою, виробник, модель, серійний номер, версії ОС, час, MAC-адреси, IP-адреси, стан Wi-Fi. Як ми бачимо на тестовому моніторі є проблеми з підключенням Інтернету, тобто машина не маже розпізнати Інтернет, його MAC-адресу та швидкість. Це означає, що в місті, де знаходиться монітор поганий Інтернет зв’язок і команді не потрібно виїзждати на місце, а лише передати інформацію клієнту, щоб він зв’язався з провайдером та вирішив проблему.

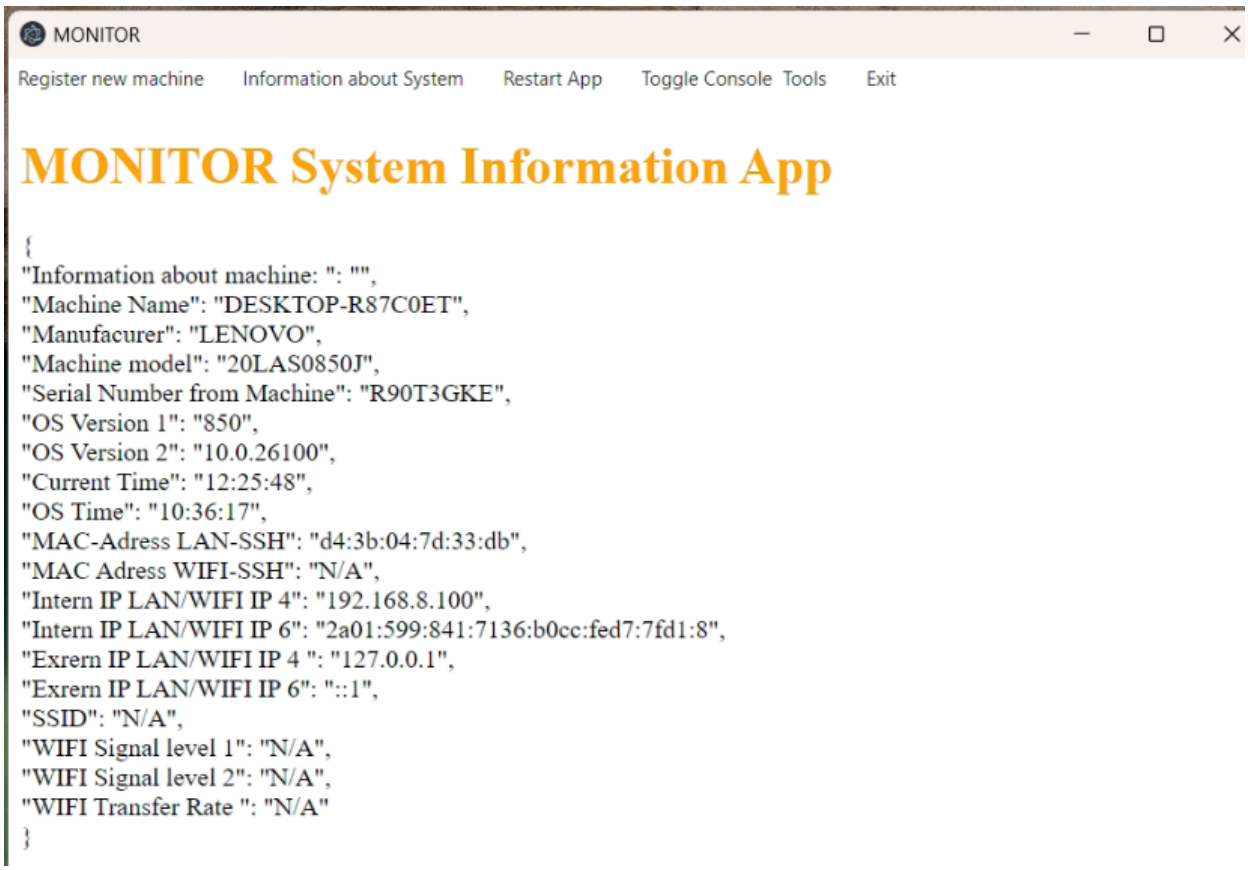


Рисунок Б.8 – Виведення системної інформації на головному вікні за стосунку

На рисунку Б.9 показані змінні в реальному часі: завантаження процесора, температура CPU та обсяг доступної оперативної пам’яті.



Рисунок Б.9 – Вивід динамічної інформації про систему

На рисунку Б.10 виводяться дані про монітор: роздільна здатність, тип підключення (INTERNAL або HDMI), серійний номер монітора (якщо є) і чи є він основним дисплеєм.

На рисунку Б.11 показано отриманий скриншот монітору у форматі Base64, що можна декодувати для перегляду зображення першого монітора.

```
{
"1 Monitor": [
"Resolution X 1920 px",
" Current Resolution X 1536 px",
" Resolution Y 1080 px",
" Current Resolution Y 864 px",
" Monitor connection INTERNAL",
" Monitor serial number undefined",
" Main monitor true"
]
}
```


Рисунок Б.10 – Інформація про підключені монітори


```
{
"1 Monitor Screenshot":
"iVBORw0KGgoAAAANSUhEUgAABgAAAANgCAIAAABlQDTBAAAgAElEQVR4nOzdd1gVV9oA8HOm3F6A
}
```


Рисунок Б.11 – Виведення скриншотів екрану

На рисунку Б.12 продемонстровано декодування Base64 у зображення, яке показує вміст монітора, де демонструється реклама клієнта у реальному часі.

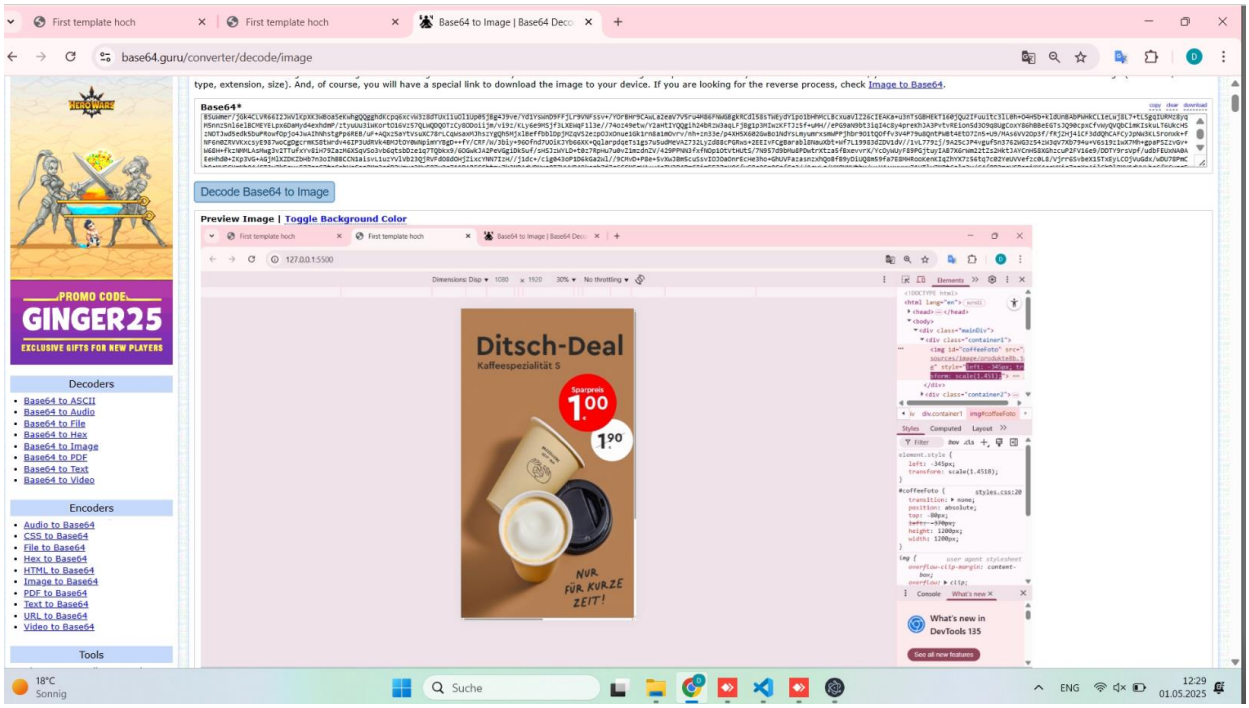


Рисунок Б.12 – Перегляд скриншота першого монітора

Таким чином, застосунок MONITOR не лише збирає загальну і технічну інформацію про систему, а й отримує актуальні дані про

навантаження системи, підключені монітори та дозволяє здійснювати захоплення екрана у реальному часі. Це відкриває можливість контролювати стан комп’ютера та підключених до нього моніторів за допомогою підключення HDMI-кабелем або у випадку віддаленого доступу, що особливо корисно для діагностики та швидкого реагування команди техніків на будь-які проблеми, які виникли.