ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота

на тему: «**ВЕБ ЗАСТОСУНОК "ВІЗІТІВКА" НА МОВІ ПРОГРАМУВАННЯ JAVASCRIPT »**

на здобуття освітнього ступеня бакалавра
зі спеціальності __122 Комп'ютерні науки_____
(код, найменування спеціальності)
освітньо-професійної програми __Комп'ютерні науки_____
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ ____Світлана ЗАЄВА_____
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. ___КНД-41___
_____Світлана ЗАЄВА_____
(Ім'я, ПРІЗВИЩЕ)

Керівник: ____Олена ШИКУЛА_____
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Рецензент: _____ _____
Науковий ступінь, вчене звання *(Ім'я, ПРІЗВИЩЕ)*

Київ – 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально–науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук
Ступінь вищої освіти − «Бакалавр»
Спеціальність підготовки − «122 Комп’ютерні науки»
Освітньо-професійна програма − «Комп’ютерні науки»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
_____Віктор ВИШНІСВЬКИЙ
“___ _” ___ _____2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Заєва Світлана Володимирівна
(прізвище, ім’я, по батькові)

1.Тема кваліфікаційної роботи: «Веб застосунок "Візітівка" на мові програмування JavaScript»

Керівник кваліфікаційної роботи Олена ШИКУЛА , доктор фіз-мат. наук, професор
(ім’я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від 24.02.2025 №56

2. Строк подання студентом роботи 30.05.2025 р.

3. Вихідні дані до роботи: односторінкова архітектура (SPA), функціональний інтерфейс, адаптивний дизайн, підтримка локалізації, персоналізація теми, інтеграція з GitHub Pages, CI/CD, використання Vanilla JavaScript без фреймворків, сучасні HTML5/CSS3 технології.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити
4.1 Огляд сайтів типу «візитівка» та визначення вимог до них.
4.2 Обґрунтування вибору технологій і архітектури SPA.
4.3 Розробка структури, дизайну та функціоналу вебзастосунку.
4.4 Тестування та оптимізація продуктивності.

5. Перелік ілюстративного матеріалу: презентація

6. Дата видачі завдання 25.02.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	24.02-10.03.25	вик.
2	Формування вимог, постановка задач і розробка структури сайту	11.03-30.03.25	вик.
3	Верстка сторінок та реалізація основного інтерфейсу	01.04-10.04.25	вик.
4	Програмування функціоналу: навігація, локалізація, теми	11.04-30.94.25	вик.
5	Оптимізація продуктивності, адаптація, тестування, CI/CD	01.05-16.05.25	вик.
6	Вступ, висновки, реферат	17.05-18.05.25	вик.
7	Розробка обов'язкових демонстраційних креслень	19.05-20.05.25	вик.

Здобувач(ка) вищої освіти _____ __Світлана ЗАЄВА_____
(підпис) (Ім'я, ПРІЗВИЩЕ)

Керівник
кваліфікаційної роботи _____ ___Олена ШИКУЛА_____
(підпис) (Ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 65 стор., 8 табл., 21 рис., 20 джерел.

Мета роботи – створення адаптивного односторінкового веб-додатка типу «візитівка» з використанням чистого JavaScript (Vanilla JS), який дозволяє ефективно презентувати навички, портфоліо та контактну інформацію початківця веб-розробника без застосування сторонніх фреймворків.

Об'єкт дослідження – односторінкові веб-додатки Single Page Application (SPA).

Предмет дослідження – методи побудови архітектури, реалізації інтерфейсу та функціональних компонентів SPA-додатків на основі клієнтських веб-технологій.

Короткий зміст роботи:

Розглянуто сучасні тенденції розробки SPA-додатків, проаналізовано роль JavaScript у побудові динамічної навігації, генерації контенту, управлінні темами й збереженні налаштувань у браузері.

Розроблено веб-додаток типу «візитівка» з логічною структурою: «Про себе», «Навички», «Портфоліо», «Контакти». Реалізовано адаптивну верстку, підтримку світлої/темної тем, перемикання мов (ua/en), генерацію vCard.

Забезпечено дотримання принципів доступності (WAI-ARIA), оптимізацію продуктивності (Lighthouse), кросбраузерність та розгортання через GitHub Pages із CI/CD.

Ключові слова: SPA, JavaScript, веб-додаток, фронтенд, доступність.

ЗМІСТ

ВСТУП

У сучасному цифровому середовищі персональна та корпоративна ідентичність все частіше формується в Інтернеті, де перше враження про професіонала або малий бізнес скеровується через веб-ресурси. Одним із популярних і водночас економічно ефективних рішень для представлення своїх контактних даних, професійних досягнень та портфоліо є веб-додаток у форматі «візитівки». На відміну від традиційних друкованих візитівок, веб-версія дозволяє не лише стисло розповісти про себе, а й реалізувати інтерактивні та мультимедійні можливості, серед яких анімації, динамічне завантаження контенту, адаптивна верстка та підтримка різних тем оформлення. Це сприяє підвищенню рівня залучення аудиторії й зміцненню цифрового бренду користувача.

Водночас розвиток фронтенд-технологій дає змогу створювати подібні рішення із застосуванням сучасних підходів і кращих практик веб-розробки. JavaScript як універсальна мова програмування в браузері виступає не лише засобом скриптування, а й основою для побудови повноцінних односторінкових додатків (SPA), які забезпечують високу швидкодію та плавну взаємодію з користувачем. Завдяки масштабованості Vanilla JS або поширеним бібліотекам та фреймворкам можна реалізувати архітектуру, придатну як для невеликих презентаційних сторінок, так і для комплексних веб-сервісів. Отже, розробка веб-додатка «візитівка» на основі JavaScript є водночас прикладом реалізації легковагового SPA і лабораторією, де випробовуються методи оптимізації продуктивності, доступності та крос-браузерної сумісності.

Метою дипломної роботи є створення відлагодженого, модульного веб-додатка «візитівка» із використанням чистого JavaScript та сучасних методів верстки CSS, який поєднує естетичний дизайн і функціональність. У межах роботи передбачено розробити архітектуру додатка з компонентним підходом до HTML-розмітки, реалізувати механізми динамічного відображення секцій без перезавантаження сторінки, а також забезпечити можливості персоналізації

інтерфейсу, серед яких перемикання між світлою і темною темами, мультивомність. Особливу увагу приділено інтеграції компонентів, таких як прогрес-бари навичок, шаблонна генерація карток портфоліо й експорту контактної інформації у форматі vCard.

Завданнями дослідження є: огляд теоретичних засад SPA-архітектури та роль JavaScript у фронтенд-розробці, аналіз інструментів та методик для оптимізації продуктивності веб-додатків, проектування файлової структури та семантичної розмітки, імплементація інтерфейсу користувача та реалізація сценаріїв взаємодії за допомогою DOM-API. Крім того, до завдань належить забезпечення крос-браузерної сумісності, адаптивної верстки для різних типів пристроїв, тестування доступності елементів (a11y) і розгортання проекту на безкоштовних хостингах із використанням CI/CD для автоматичного оновлення.

Науковою новизною роботи є комплексне поєднання мінімалістичного дизайну з повним набором функціональних можливостей SPA-додатка без залучення великих фреймворків, що дає змогу суттєво зменшити вагу файлів і підвищити швидкість завантаження. Запропонований підхід передбачає побудову власного шаблонізатора на рівні клієнтського JavaScript для генерації повторюваних блоків, а також розробку адаптивного модуля збереження налаштувань користувача в localStorage. Завдяки цьому досягається висока ступінь кастомізації та швидкий відгук інтерфейсу без додаткових серверних викликів.

Практична цінність дослідження полягає у можливості застосування отриманих результатів як базового рішення для портфоліо-файлів, онлайн-резюме або корпоративних міні-сайтів. Окрім того, методики оптимізації та тестування, відпрацьовані в рамках роботи, можуть бути адаптовані для інших малих SPA-проектів, де критичною є вага сторінки та час взаємодії з елементами.

Методологічною основою дослідження стали методи системного аналізу та модульного проектування веб-додатків, а також експериментальні дослідження продуктивності за допомогою інструментів Google Lighthouse та WebPageTest. Для оцінки доступності застосовано за методикою WAI-ARIA перевірку коректних атрибутів, фокус-стилів і семантичних тегів. Результати тестування і оптимізації

лягли в основу рекомендацій щодо кращих практик побудови легких та швидких SPA-додатків.

Структура роботи складається з трьох розділів. У першому розділі викладено теоретичні аспекти веб-додатків, роль і можливості JavaScript як основного засобу клієнтського програмування. Другий розділ присвячено детальному опису архітектури та реалізації власне веб-додатка «візитівка», включаючи розробку інтерфейсу, скриптів динамічної генерації контенту й механізмів збереження налаштувань. У третьому розділі наведено результати тестування, оптимізації та розгортання проєкту на обраному хостингу, а також рекомендації щодо подальшого розвитку та інтеграції додаткових модулів. Таким чином, виконана робота демонструє повний цикл створення сучасного односторінкового веб-додатка від концепції до практичного розгортання.

1 ТЕОРЕТИЧНІ ЗАСАДИ ВЕБ-ДОДАТКІВ І РОЛЬ JAVASCRIPT

1.1 Поняття веб-додатка та SPA (Single Page Application)

Поняття веб-додатка й еволюція від класичних сайтів до SPA

Упродовж розвитку мережі Інтернет від перших статичних веб-сторінок, які переважно містили текстову й графічну інформацію, до сучасних складних інтерактивних сервісів спостерігається поступова ускладненість архітектурних підходів та розширення функціональних можливостей. Під терміном веб-додаток звичайно розуміють програмний комплекс, що працює в середовищі веб-браузера і забезпечує взаємодію з користувачем у режимі реального часу. На відміну від традиційного веб-сайту, веб-додаток здатен обробляти значну частину логіки на стороні клієнта, забезпечуючи швидку відповідь на дії користувача без необхідності повного перезавантаження сторінки. Такий підхід скорочує затримки, підвищує комфорт користувацького інтерфейсу та робить можливим створення повноцінних програмних продуктів, доступних без встановлення на кінцевому пристрої.

Single Page Application (SPA) є специфічним різновидом веб-додатків, який відрізняється тим, що всі необхідні ресурси (HTML, CSS, JavaScript) завантажуються одноразово під час першого звернення користувача. Далі при взаємодії з інтерфейсом відбувається динамічна підвантаження лише тих даних, які необхідні для оновлення частини сторінки. Це дозволяє домогтися двох ключових ефектів: по-перше, відсутність повного перезавантаження сторінки економить мережевий трафік і знижує час очікування користувача, а по-друге, забезпечує безперервність інтерфейсу, що підвищує відчуття «нативності» програми. У результаті SPA виглядають і поводяться як класичні настільні чи мобільні застосунки, де взаємодія відбувається миттєво і плавно.

Розглядаючи історію становлення SPA, варто згадати, що перші спроби реалізації інтерактивності здійснювалися за допомогою технології AJAX (Asynchronous JavaScript and XML), яка з’явилася на початку 2000-х років. Саме

AJAX дозволив відокремити запити до сервера від оновлення вмісту сторінки, що розширило можливості класичних веб-сайтів. Надалі виникли спеціалізовані фреймворки й бібліотеки, які стандартизували підходи до побудови SPA, серед яких Angular, React і Vue. Водночас зростали й вимоги до оптимізації продуктивності й безпеки, що спонукало розробників удосконалювати клієнтські архітектури та впроваджувати рішення на кшталт клієнтського маршрутизатора й управління станом додатка.

У контексті дипломної роботи з розробки веб-додатка-«візитівки» SPA виступає оптимальним вибором, адже такий проєкт потребує швидкого відгуку на дії користувача, мінімізації передачі даних при навігації між розділами й підтримки персоналізованих налаштувань інтерфейсу (темна/світла тема, обраний користувачем інтерфейс тощо). Завантаживши один раз основні ресурси, користувач може пересуватися між розділами «Про мене», «Навички», «Портфоліо» чи «Контакти» без фактичного запиту до сервера, що створює відчуття плавного і безперервного інтерфейсу.

В епоху мобільних пристроїв і непередбачуваних умов мережі SPA демонструють високу ефективність, оскільки дозволяють кешувати критично важливі ресурси у браузері або навіть у Service Worker, що дає можливість працювати в оффлайн-режимі з обмеженим функціоналом. Такий підхід ідеально підходить для реалізації міні-проєктів на кшталт веб-додатка-«візитівки», де обсяг даних невеликий, але вимога до швидкодії й користувацького досвіду надзвичайно висока.

JavaScript, як універсальна мова програмування у браузері, надає розробникам широкий набір інструментів для створення інтерактивних елементів інтерфейсу. Серед основних функціональних можливостей клієнтської JavaScript-логіки відзначаються маніпуляції з DOM (Document Object Model), що дозволяють динамічно змінювати вміст і структуру веб-сторінки. Для розробки «візитівки» це втілюється у здатність показувати та приховувати окремі секції, плавно змінювати стилі елементів, а також додавати і видаляти вузли без перезавантаження документа.

Асинхронні HTTP-запити за допомогою Fetch API є ключовим механізмом взаємодії з сервером. Навіть у відносно невеликому SPA можуть з’явитися потреби у підвантаженні даних, наприклад, форм роботи у портфоліо або відгуків користувачів. З Fetch API виникає можливість не блокувати виконання основного потоку та забезпечити відображення прогресу обробки запиту, а також виконувати обробку помилок, що робить веб-додаток більш стійким до мережевих збоїв.

Управління станом програми може бути організовано за допомогою простих об’єктів у JavaScript або шляхом інтеграції невеликих бібліотек для стану, якщо це виправдане обсягом логіки. Для веб-додатка-«візитівки» найчастіше достатньо зберігання обраного розділу, налаштувань теми та мови в localStorage, що доступний через єдиний API Web Storage. Це дає змогу підтримувати персоналізований досвід без потреби у серверній базі даних і забезпечує швидке відновлення налаштувань при повторних відвідинах.

Крім того, JavaScript застосовується для реалізації анімацій і візуальних ефектів, які підвищують привабливість інтерфейсу. CSS-анімовані переходи та ключові кадри можна координувати через JS-події, що дає змогу запускати анімацію в потрібний момент, наприклад, при першому вході користувача в розділ «Навички», коли відбувається поступове заповнення прогрес-барів.

У контексті крос-браузерної сумісності JavaScript також використовується для виявлення можливостей браузера та адаптації поведінки додатка залежно від підтримки тих чи інших API. Наприклад, перевірка наявності Service Worker або підтримки Local Storage дозволяє перемикатися на спрощений режим роботи в старих браузерах або забезпечувати розширений функціонал у сучасних клієнтських середовищах.

Незважаючи на популярність фреймворків і бібліотек, таких як React або Vue, використання чистого (Vanilla) JavaScript у межах веб-додатка-«візитівки» є виправданим із кількох причин. По-перше, проєкт має невеликий обсяг функціональних вимог, що не потребує складного управління станом чи маршрутизації на стороні клієнта, коли можна обійтися простими методами роботи з DOM і localStorage. По-друге, відмова від сторонніх залежностей знижує вагу

остаточного пакета ресурсів, що прискорює час початкового завантаження та покращує показники продуктивності.

Використання Vanilla JS сприяє глибшому розумінню базових підвалин клієнтської архітектури. Розробник навчається працювати напряму з браузерним API, що полегшує діагностику проблем і оптимізацію. З іншого боку, відсутність необхідності підлаштовуватися під специфіку фреймворку зменшує складність навчання команди та подальшої підтримки коду. Коли вимоги до функціональності зростають, завжди можна поступово інтегрувати легковагові бібліотеки або рефакторити частини коду під фреймворк, не змінюючи загальну архітектуру.

У таблиці 1.1 наведено порівняння підходу на чистому JavaScript і типових фронтенд-фреймворків (React, Vue) для реалізації SPA-додатків за ключовими характеристиками.

Таблиця 1.1 – Порівняння характеристик підходу на чистому JS та використання типового фронтенд-фреймворку для SPA.

Характеристика	**Vanilla JS**	**Фреймворк SPA (React, Vue)**
Розмір пакета	Невеликий, не потребує бандлера	Залежить від фреймворку і плагінів
Потреба в інструментах збірки	Необов’язкова	Зазвичай необхідна (Webpack, Vite)
Швидкість завантаження	Висока	Середня–нижча без оптимізації
Кривизна навчання	Нижча при базовому функціоналі	Вища через JSX/TSX та екосистему
Гнучкість в управлінні станом	Проста через об’єкти/Storage	Потужна через Redux/Vuex
Підтримка екосистеми	Обмежена сторонніми бібліотеками	Широка (Router, State, Testing)

Архітектурна модель веб-додатка-SPA передбачає єдину точку входу у вигляді index.html, де підключаються всі необхідні скрипти та стилі. Основна логіка клієнта організована навколо обробників подій, маніпуляцій із DOM та асинхронних запитів. Завантаження даних із сервера може здійснюватися через REST API або GraphQL, проте для «візитівки» достатньо статичних файлів або мінімального бекенду, який віддає JSON із контентом. Модульна побудова коду за допомогою ES6-модулів дозволяє розділити функціональність на логічні блоки, що полегшує обслуговування й розширення проекту.

Реалізація навігації без перезавантаження сторінок досягається простим покажчиком поточного розділу у вигляді змінної додатка або фрагмента в URL (hash routing). При кліку на навігаційну кнопку JavaScript скрипт встановлює новий фрагмент після символу «#» і відповідно змінює відображуваний блок, приховуючи інші. Це ж працює при прямому введенні URL із хешем у адресному рядку.

Прогрес-бари навичок анімовані за допомогою CSS-переходів і програмного встановлення стилю ширини елементів у JS. Перший виклик функції відбувається під час активації секції «Навички», що дозволяє відтворити ефект поступового заповнення. Картки портфоліо генеруються на основі масиву об’єктів із властивостями title, img, desc та tags. Використання шаблонних рядків спрощує вставку HTML-структури внутрішньо циклу, а теги відображаються через динамічне створення і вставку елементів із відповідним класом.

Збереження налаштувань користувача в localStorage забезпечує збереження стану теми (Темна/Світла) та обраної мови інтерфейсу. Після завантаження сторінки скрипт перевіряє наявність відповідних ключів і застосовує збережені значення, що покращує користувацький досвід і робить додаток персоналізованим.

Таким чином, обраний підхід поєднує гнучкість і високу швидкодію Vanilla JS з принципами SPA-архітектури, що дає змогу створити ефективний веб-додаток-«візитівку» з мінімальною вагою і тривалістю розробки. Цей розділ демонструє, як за допомогою базових засобів браузера побудувати сучасний клієнтський інтерфейс та забезпечити привабливий і зручний користувацький досвід без складних зовнішніх залежностей.

1.2 Огляд аналогів реалізації SPA-додатків

Односторінкові вебзастосунки (Single Page Applications, SPA) стали провідним підходом у створенні сучасного інтерфейсу користувача завдяки своїй високій продуктивності, швидкому рендерингу, зменшенню кількості запитів до сервера та плавному досвіду взаємодії. SPA-додатки динамічно оновлюють контент сторінки без повного перезавантаження, що значно покращує юзабіліті та враження користувачів. Цей підхід широко використовується в соціальних мережах, корпоративних системах, адміністративних панелях, маркетплейсах та SaaS-продуктах.

У цьому підрозділі розглянуто найпоширеніші реалізації SPA-додатків. Порівняння базується на архітектурних принципах, гнучкості, простоті підтримки, продуктивності, доступності.

Одним із найпоширеніших варіантів побудови SPA є застосування бібліотеки React. Її головна концепція полягає у побудові інтерфейсу за допомогою компонентів, що значно спрощує повторне використання коду та його підтримку. Компонентна структура дозволяє ефективно організовувати проєктну архітектуру, а віртуальний DOM, який використовується React, оптимізує процес оновлення UI, мінімізуючи кількість змін у реальному DOM. Це забезпечує високу продуктивність навіть при великій кількості динамічних змін на сторінці.

React має потужну та розвинену екосистему. Наприклад, React Router дозволяє реалізувати маршрутизацію без перезавантаження сторінки, Redux або Context API забезпечують централізоване управління станом, а Next.jsрозширює можливості бібліотеки на рівні серверного рендерингу та SEO-оптимізації. Завдяки цьому React гнучко адаптується до потреб як невеликих, так і масштабних додатків.

Попри численні переваги, існують і певні виклики. Зокрема, початкове налаштування середовища часто вимагає використання інструментів на зразок Webpack або Vite, що може ускладнити старт проєкту для розробників-початківців. Крім того, гнучкість бібліотеки зумовлює відсутність єдиної

архітектурної парадигми, тож розробник змушений самостійно ухвалювати рішення щодо структури проєкту, вибору додаткових бібліотек та організації бізнес-логіки.

Ще одним поширеним фреймворком для створення SPA-додатків є Vue.js – прогресивна JavaScript-технологія, яка поєднує у собі сильні сторони як React, так і Angular. Основна ідея Vue полягає у використанні компонентного підходу, а також декларативного синтаксису, що робить розробку швидкою та інтуїтивно зрозумілою навіть для розробників-початківців. Однією з ключових особливостей Vue є двостороння прив'язка даних, що забезпечує автоматичну синхронізацію між станом та відображенням інтерфейсу, значно спрощуючи створення інтерактивних елементів.

Фреймворк надає вбудовану підтримку для маршрутизації (Vue Router) і керування станом (Vuex, а в новіших версіях – Pinia), що дозволяє будувати як невеликі SPA, так і повноцінні складні додатки. Наявність CLI-інструментів значно полегшує початкову конфігурацію проєкту та генерування шаблонів, що сприяє швидкому старту розробки.

Серед переваг Vue слід також відзначити зрозуміле API, низький поріг входу, активну спільноту та можливість поступового впровадження у вже існуючі проєкти. Це робить фреймворк популярним вибором для стартапів, невеликих команд та проєктів з обмеженими ресурсами.

Разом з тим, при масштабуванні великих проєктів можуть виникати складнощі, пов'язані з архітектурною організацією коду. Vue є досить гнучким, але саме ця гнучкість вимагає досвіду та продуманої структури, щоб уникнути проблем у підтримці та розширенні системи. Крім того, в порівнянні з React, екосистема Vue має меншу кількість рішень для серверного рендерингу (SSR), хоча вона продовжує активно розвиватися в цьому напрямку (зокрема, через Nuxt.js).

Angular – це повноцінний фронтенд-фреймворк, створений та підтримуваний компанією Google, який призначений для розробки масштабованих односторінкових додатків зі строгою архітектурною структурою. Він надає інтегровані засоби для вирішення ключових завдань розробки, зокрема

маршрутизації, валідації форм, обробки HTTP-запитів, ін’єкції залежностей, тестування та організації архітектури додатка.

Однією з визначальних характеристик Angular є використання TypeScript за замовчуванням, що забезпечує типізацію, автодоповнення, раннє виявлення помилок та загальну стандартизацію коду. Фреймворк також надає потужні CLI-інструменти (Angular CLI), які дозволяють швидко створювати, конфігурувати та розгортати проєкти з єдиним підходом до структури.

Angular вирізняється високим рівнем організованості, що робить його привабливим рішенням для великих команд, які працюють над довгостроковими корпоративними проєктами. Завдяки суворим архітектурним принципам, підтримці модульності, суворій декомпозиції та чітко визначеному життєвому циклу компонентів, фреймворк забезпечує стабільність і масштабованість великих SPA-систем.

Разом із тим, Angular має високий поріг входу, особливо для розробників-початківців. Його синтаксис складніший порівняно з іншими фреймворками, а початковий розмір згенерованого пакета зазвичай більший, що може впливати на швидкість першого рендеру. Крім того, надлишкова функціональність для простих застосунків може призводити до надмірної складності, яка не завжди є виправданою у малих або середніх проєктах.

Крім фреймворків, важливим напрямом є реалізація SPA-додатків без використання сторонніх бібліотек – на чистому JavaScript. Такий підхід передбачає повну відповідальність розробника за структурування коду, побудову маршрутизації, керування станом і організацію динамічного відображення контенту. Він дозволяє створити легкий, незалежний від сторонніх рішень застосунок, що швидко завантажується, легко масштабується в межах власних потреб і не несе зайвих накладних витрат. Використання чистого JavaScript особливо виправдане у навчальних або демонстраційних проєктах, де розробник прагне глибоко опанувати механізми роботи DOM, розуміння навігації та структури SPA. Такий підхід також дає можливість реалізувати оптимальну продуктивність та відповідати вимогам доступності, проте водночас вимагає

значно більше часу на проєктування архітектури, а також досвіду для запобігання помилкам і підтримки застосунку у майбутньому.

2 АРХІТЕКТУРА ТА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКА «ВІЗИТІВКА»

2.1 Функціональні можливості JavaScript у веб-додатку-«візитівці»

Функціональні можливості JavaScript у веб-додатку-«візитівці» розкриваються через низку ключових аспектів взаємодії клієнта з браузером, які разом формують сучасний користувацький досвід. Уявімо, що браузер обробляє документ із підключеним скриптом. Відразу після завантаження сторінки JavaScript-код ініціює підготовчі процедури: зчитує налаштування користувача, які збережено в localStorage, реєструє обробники подій на елементах інтерфейсу, відновлює стан темної або світлої теми та налаштування мови. Важливо, що всі ці дії відбуваються без залучення серверної логіки та не вимагають перезавантаження сторінки, забезпечуючи миттєвий відгук на дії користувача.

Принципово JavaScript відкриває доступ до DOM-моделі, яка є програмним уявленням HTML-структури. Саме через маніпуляції з DOM реалізується динамічна навігація між розділами «Про мене», «Навички», «Портфоліо» й «Контакти». При натисканні на навігаційну кнопку скрипт зчитує значення атрибута data-target, ховає всі секції, додає клас active до обраної та ініціює CSS-анімацію плавного появлення. Це створює у користувача відчуття єдиного простору, де перемикання між вмістом відбувається миттєво й без мерехтіння.

Окрім базових маніпуляцій, JavaScript у веб-додатку-«візитівці» використовує асинхронні можливості Fetch API. Хоча основну інформацію можна закодувати в статичних масивах, у стандартному SPA-додатку це дає змогу підвантажувати блоки даних без перезавантаження. Наприклад, якщо з портфоліо вирішено винести до бекенду оновлюваний список проектів Fetch API опрацьовує запит у фоновому режимі, повертає JSON і передає його обробнику, який динамічно створює відповідні HTML-елементи. Протягом очікування відповіді

можна показати індикатор завантаження, що покращує комунікацію з користувачем і знижує рівень фрустрації під час повільного інтернет-з’єднання.

Важливим елементом взаємодії є управління станом інтерфейсу через Web Storage. JavaScript забезпечує уніфікований API для доступу до localStorage та sessionStorage, що дозволяє зберігати текстові значення в локальному середовищі користувача. У випадку веб-додатка-«візитівки» це реалізовано у формі збереження обраної теми (світла чи темна) та мовного пакета інтерфейсу (Українська або Англійська). Після кожної зміни цих налаштувань скрипт оновлює відповідні ключі у localStorage, а під час наступного відвідування – автоматично відновлює колишній стан, подарувавши відчуття персоналізації та стабільності.

Не меншу роль відіграє обробка подій: JavaScript реєструє слухачі click на кнопках завантаження vCard, на перемикачі теми, на кнопках навігації та навіть на формі зворотного зв’язку. У випадку vCard скрипт формує текстовий рядок у форматі vCard, утворює з нього Blob із MIME-типом text/vcard, генерує тимчасовий URL і програмно викликає елемент <a> з атрибутом download. Завдяки цьому користувач отримує файл контактів без потреби звертатися до серверу.

JavaScript також координує анімації й інтерактивні ефекти, поєднуючи CSS-перехід та програмне керування класами. Наприклад, для секції «Навички» скрипт викликає функцію, що читає значення data-value в елементах із класом progress-fill і додає відповідну ширину. CSS-перехід забезпечує плавне «заповнення» смужок, яке запускається лише раз при першій активації розділу, надаючи динаміки й виділяючи досягнення користувача.

Сучасні браузери підтримують і більш складні API, такі як Intersection Observer, що дає змогу відстежувати появу елементів у полі зору користувача без затрат на часті обчислення scroll-подій. У межах масштабнішого SPA це може бути застосовано для lazy-loading зображень у портфоліо, але для невеликої «візитівки» і CSS-анімацій достатньо базових методів. Тим не менш знання цих патернів свідчить про готовність розробника використовувати передові підходи для оптимізації продуктивності.

При побудові клієнтської архітектури важливо також опрацювати помилки, що виникають під час асинхронних операцій або несподіваної відсутності localStorage. Використання конструкцій try…catch разом із Promise-методами .catch() дозволяє відловлювати виключення, повідомляти користувача про проблеми з мережею або ставити додаток у безпечний режим із дефолтними налаштуваннями. Це підвищує стійкість «візитівки» навіть у випадку старих браузерів або приватного режиму, що блокує Web Storage.

Нарешті, у контексті міжнародних користувачів ротація мовного пакета реалізується через об’єкт-словник, що містить переклади ключових рядків. Після зміни мови JavaScript оновлює тексти заголовків, описів і підписів у DOM, зберігає вибір у localStorage та оновлює інтерфейс без перезавантаження. Такий підхід демонструє універсальність клієнтського JS та його здатність перетворювати просту веб-сторінку на багато мовний інструмент презентації.

У сукупності описані функціональні можливості JavaScript створюють єдине середовище, де кожна дія користувача супроводжується миттєвим відгуком, а весь інтерфейс «візитівки» перетворюється на сучасний SPA-додаток з персоналізованим досвідом, високою продуктивністю та повною відсутністю потреби в повному перезавантаженні. Такий підхід забезпечує конкурентну перевагу для тих, хто прагне представити себе в Інтернеті максимально компактно, але при цьому на високому технологічному рівні.

2.2 Обґрунтування вибору Vanilla JS

Обираючи технологічну основу для створення веб-додатка «візитівки», розробник неминуче зважує питання масштабу проєкту, очікуваного часу відгуку інтерфейсу, ваги початкового пакета ресурсів і зручності подальшого супроводу. Коли йдеться про легку односторінкову презентацію, зростає спокуса скористатися одним із популярних фреймворків і отримати «з коробки» готову маршрутизацію, узори керування станом і компонентну модель. Проте уважний аналіз потреб

виявляє, що залежність від важкої екосистеми не лише перевантажує простий проєкт зайвими абстракціями, а й вводить приховані витрати на складання, оновлення пакетів, вирішення конфліктів залежностей і підтримку інструментального ланцюга. Саме тому в дипломній роботі акцентовано переваги чистого, або ж Vanilla JS – підходу, що опирається виключно на стандартні браузерні API без залучення зовнішніх бібліотек й фреймворків.

Перевага мінімалістичного стеку передусім проявляється у швидкості завантаження. Для кінцевого користувача найцінніші секунди – це ті, що минають між клацанням посилання й моментом, коли сторінка стає інтерактивною. Коли застосунок побудовано на Vanilla JS, браузеру достатньо отримати один-два невеликі скрипти та таблицю стилів, аби вивести коректно розмічену HTML-структуру й дозволити користувачеві взаємодіяти з нею. Час, який витрачається на ініціалізацію складних віртуальних DOM, попередню компіляцію JSX або трансформацію модулів, просто відсутній. Перевага стає особливо відчутною на мобільних пристроях третього світу чи в умовах нестабільного мобільного зв’язку, де кожен кілобайт трафіку й додаткова секунда затримки зумовлюють втрату аудиторії.

Окрема тема – обслуговування та безпека залежностей. Невеликий проєкт, що використовує десятки або сотні пакетів із відкритих репозиторіїв, мимоволі втягується в гоночну динаміку оновлень. Кожен мажорний реліз фреймворку тягне за собою оновлення супутніх інструментів, конфігурацій та потенційні регресії. Vanilla JS фактично довіряє стабільності браузерних API, що еволюціонують повільно й ретельно документуються консорціумом W3C. Це мінімізує ризик того, що через рік після захисту дипломної роботи сторінка перестане відображатися через застарілий синтаксис чи вразливість у ланцюгу залежностей, яку вже ніхто не патчить.

Суттєвий аргумент – контроль над фінальною розміткою й семантикою. Працюючи напряму з DOM, розробник не обмежений шаблонними конвенціями JSX-препроцесора або керованого рендерера; він повною мірою відповідає за коректність тегів, aria-атрибутів і побудову логічної ієрархії. Це полегшує аудит

доступності, дає змогу точково покращувати SEO-складову та гнучко керувати критичними CSS – методикою, котра передбачає інлайн-вставку мінімуму стилів у <head> для миттєвого рендеру. Коли ж інтерфейс зводиться до кількох секцій і кількох десятків елементів, додаткова надбудова у вигляді віртуального дерева лише віддаляє розробника від «натурального» HTML і ускладнює діагностику проблем рендеру.

Продуктивність виконання скриптів – ще один аспект, на якому позначається вибір Vanilla JS. Забезпечуючи лише ту логіку, що справді потрібна додатку, ми не змушуємо користувача завантажувати й парсити код, пов’язаний з компонентною системою, сервісами інверсії залежностей і велетенськими допоміжними утилітами. У сучасних браузерах операції з натуральним DOM при невеликій кількості вузлів виконуються блискавично, а економія CPU-циклів безпосередньо впливає на енергоефективність мобільних пристроїв – важливий фактор для аудиторії, що читає сторінки з акумулятором на останніх відсотках заряду.

Не менш значущою є й педагогічна складова. Дипломна робота, яка демонструє чистий JavaScript, фактично засвідчує глибоке розуміння базових клієнтських API та патернів. Студент-розробник показує, що вміє без «чарівної палички» фреймворку реєструвати події, маніпулювати класами, працювати з асинхронними запитами та кешем браузера. Такий досвід робить його більш гнучким фахівцем, здатним адаптуватися до будь-якої екосистеми, адже фундаментальні знання не залежать від модних трендів.

Тим часом питання масштабування цілком розв’язується й у межах Vanilla JS. ES-модулі дають змогу розбивати код на логічні частини й пізніше підключати лише ті з них, що потрібні. Якщо виникне потреба додати складніший роутинг або централізоване управління станом, можна інсталювати вузькоспеціалізовану бібліотеку й інкапсулювати її на рівні одного-двох файлів, не переписуючи всю архітектуру. Інакше каже, чистий JS не закриває двері для майбутніх поліпшень, а лише відкладає їх до тієї миті, коли вони стануть обґрунтовано необхідними.

Щоб емпірично продемонструвати різницю, розгляньмо невелику порівняльну таблицю, в якій на основі метрик Google Lighthouse підсумовано

середні показники завантаження типового «Hello world» на Vanilla JS та React 17 (у режимі production зі ввімкненою tree-shaking-оптимізацією).

У таблиці 2.1 наведено порівняння основних метрик Google Lighthouse для двох SPA-рішень: на чистому JavaScript і з використанням React + Webpack.

Таблиця 2.1 – Основі метрик Google Lighthouse

Метрика Google Lighthouse	**Vanilla JS (14 KB)**	**React 17 + Webpack (110 KB)**
Time to Interactive	0,7 с	2,4 с
First Contentful Paint	0,4 с	1,2 с
Cumulative Layout Shift	0	0
Total Blocking Time	12 мс	180 мс
Індекс продуктивності (0–100)	99	86

Подані числа ілюструють, наскільки суттєво початкова вага бандла та додаткові мінімізації впливають на швидкодію. Хоча в реальному проєкті «візитівки» різниця, ймовірно, буде меншою, тенденція залишиться: родинні операції з DOM виконуються швидше й економніше, коли додаткове абстрактне ядро відсутнє.

Беззаперечно, фреймворки пропонують багаті можливості, що стають у нагода в масштабних застосунках із тисячами компонентів, складними формами й частими взаємодіями з API. Але для міні-проєкту, який має відобразити інформацію про автора, кілька навичок і короткий перелік робіт, більша частина цього функціоналу лежить поза колом першочергових задач. Витрачати час на конфігурацію бандлера, Babel-пресетів і ESLint-правил доречно лише тоді, коли вигода від масштабованого середовища перевищує стартові накладні витрати.

Таким чином, обґрунтування вибору Vanilla JS спирається на три групи аргументів: технічні (продуктивність і вага), організаційні (простота підтримки і

безпека) та освітні (поглиблене засвоєння базових API). Вони формують переконливу підставу використовувати чистий JavaScript у дипломній роботі, де кінцевою метою є компактний, швидкий і легко транспортуємий веб-додаток-«візитівка», здатний працювати на будь-якому сучасному браузері без спеціальних передумов. Саме ця стратегія найповніше відповідає принципам прогресивного вдосконалення, коли інтерфейс залишається доступним і функціональним навіть за мінімальних умов, а додаткові можливості активуються там, де це допускає середовище.

2.3 Структура HTML і CSS: семантика та стилі

Розуміння внутрішньої логіки HTML-каркаса й системи стилів у веб-додатку «візитівка» починається зі свідомого вибору семантичних елементів, які віддзеркалюють смислову структуру контенту й водночас забезпечують машиночитність для пошукових систем і засобів асистивних технологій. Усе зводиться до того, що кожен блок інтерфейсу має бути описаний настільки точним тегом, наскільки це можливо, щоби полегшити не лише візуальну інтерпретацію, а й формування логічної карти документа. Саме тому контент, що стосується заголовку сторінки і ключової інформації про власника «візитівки», обгорнуто тегом <header>; навігаційні елементи зосереджено у <nav>, а змістові підрозділи винесено у власні <section> блоки. Такий розподіл робить HTML не тільки зрозумілим людині-розробнику – він формує фундамент доступності, адже скрінрідери автоматично розпізнають межі основних регіонів і дозволяють користувачам із порушеннями зору пересуватися між ними швидкими клавішними комбінаціями.

Одразу після відкриття документа в <head> підключаються метатеги кодування й viewport, що гарантують коректне відтворення кирилиці та пристосованість макета до ширини мобільних дисплеїв. Важливим маркером візуальної цілісності стає підключення шрифтової гарнітури через Google Fonts;

саме тут доречно обрати варіації «Inter» з підмножинами 400, 500 та 700 для забезпечення базового, акцентного й напівжирного ритму тексту. Така триступенева шкала ваги дозволяє дотримуватися чіткої ієрархії типографіки без нав’язливого нагромадження стилів.

Ключовим контейнером, який визначає ширину робочої області, слугує . Його розміри описані через вираз clamp(320px, 95vw, 820px), таким чином мінімальна ширина ніколи не буде меншою за 320 пікселів – ця цифра відповідає екрану умовного iPhone SE, – а верхній поріг у 820 пікселів гарантує комфортне читання на середніх та великих десктопах. Параметр 95vw сприяє те, що блок логічно «дихає» всередині вікна й не прилипає до країв, залишаючи невеликий відступ для природного білого простору. Підтримка плавної зміни ширини дає змогу уникнути переломів рядка й хаотичного перерозподілу елементів, котрі часто трапляються у фіксованих макетах.

На рисунку 2.1 зображено, як вебзастосунок адаптується до ширини екрана: центральна область не розтягується на всю ширину, що забезпечує комфортне читання на різних пристроях і зберігає білі відступи з боків.

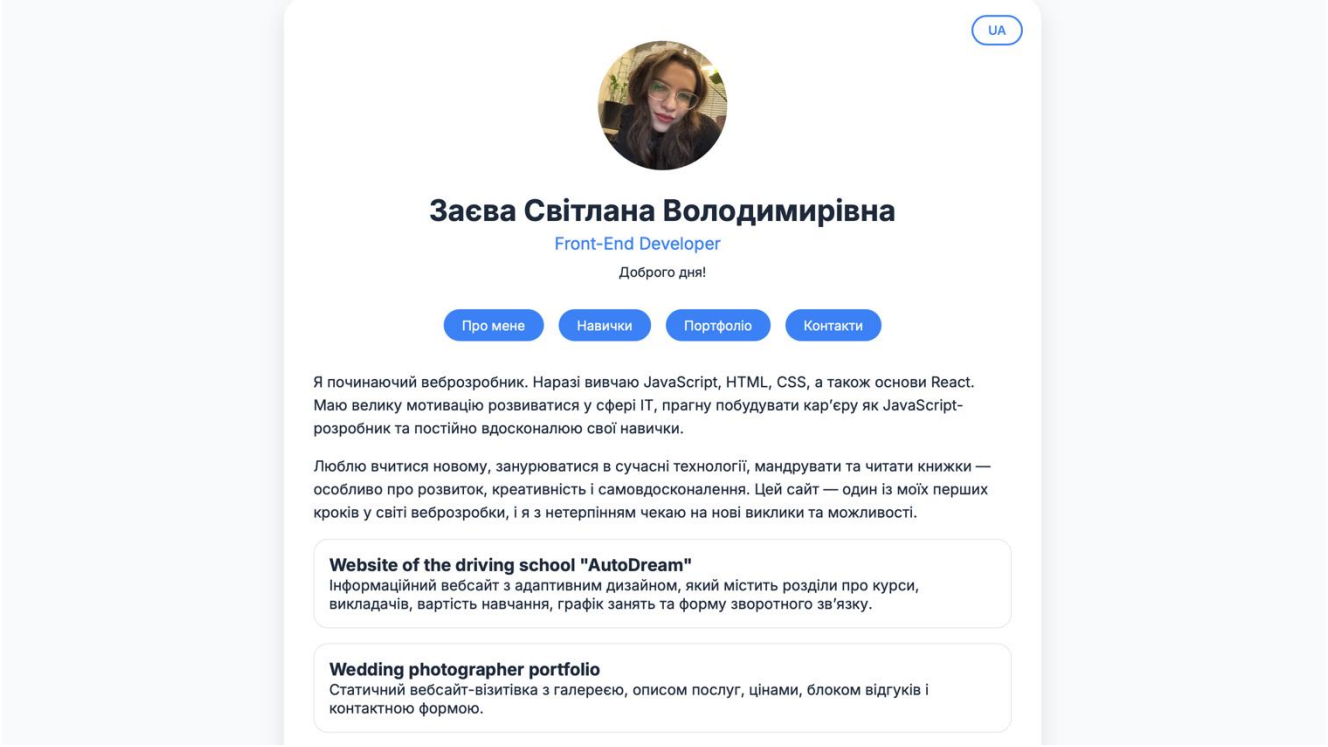

Рисунок 2.1 – Адаптивна ширина вебзастосунку

Сама оболонка оформлена світлою темою за замовчуванням, де фоновий колір ініціалізовано змінною --card-light, а текст – --text-light. Ці змінні оголошено у псевдокласі :root, щоб забезпечити глобальну досяжність і можливість легкого перемикання палітри при активації темної теми. Перехід до нічного режиму виконується шляхом додавання класу dark на елемент <body>, який переозначає відповідні CSS-змінні. Такий підхід значно ефективніший, аніж дублювання всіх правил у альтернативному наборі селекторів, оскільки достатньо змінити одні лиш кольорові змінні, щоби автоматично перефарбувати цілий інтерфейс.

Оформлення самого .wrapper доповнено тінню box-shadow: 0 10px 25px rgba(0, 0, 0, 0.1). Її завдання – прив’язати вміст до оптичної площини, легенько піднімаючи над фоном, тим самим створюючи відчуття глибини. Радіус заокруглення 1.5rem підкреслює дружність інтерфейсу; числове значення, виражене в одиницях rem, сприяє масштабуванню разом із базовим шрифтовим розміром користувача й, відповідно, покращує доступність для людей із порушеннями зору.

Усередині хедера розміщено аватар із класом .avatar. Важливо, що зображення відтворюється не у фоновому стилі CSS, а як звичайний img із атрибутом alt, який описує зміст. Завдяки властивості object-fit: cover та фіксованому розміру сторон під час обрізання зображення у коло 140 × 140 пікселів кадрування відбувається пропорційно, а отже деталей портрета не спотворюється. Обрамлення кутів реалізоване через border-radius: 50%, що допомагає отримати ідеальне коло без зайвих SVG-масок. Поряд під аватаром розміщено <h1> з іменем та прізвищем власниці сторінки. Тут важливо лишити саме перший заголовок найвищого рівня, аби забезпечити правильний контур документа для пошукових роботів.

На рисунку 2.2 зображено верхню частину сторінки – що містить персональний аватар та ім’я користувача. Аватар представлений у формі кола. Під зображенням розміщено заголовок першого рівня з іменем і прізвищем.

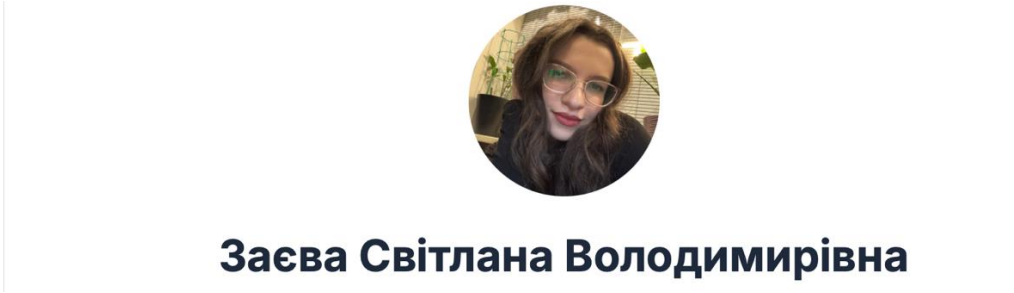

Рисунок 2.2 – Аватар та особиста інформація

Особливої уваги заслуговує декоративний рядок із класом .typing. Це текст, який за допомогою ключових кадрів @keyframes typing та @keyframes blink створює ілюзію набирання символів курсором. Секрет такого ефекту – у властивості white-space: nowrap; overflow: hidden; border-right, що разом працюють на те, аби текст зникав і з’являвся поступово, а курсор блимав унісон із шириною (Рисунок 2.3). Уночі або при низькій яскравості ця анімація може відволікати, тож вона лишається однаково контрастною, оскільки облямівка отримує колір із змінної --accent, який у темній темі лишається ідентичним.

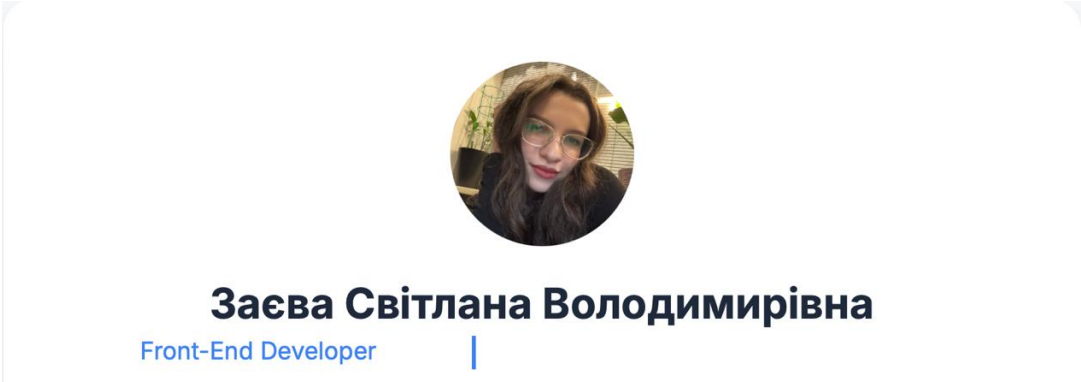

Рисунок 2.3 – Анімований рядок введення

Навігація втілена серією кнопок з атрибутами data-target. По суті, вони репрезентують безпосередню взаємодію користувача із SPA-маршрутизатором, реалізованим через обробник події click. Коли JavaScript зчитує цей атрибут, він точно знає, яку секцію контенту має зробити видимою. Самі кнопки стилізовані як «пігулки» завдяки border-radius: 999px. Такий прийом вимагає, щоб горизонтальні

падінги значно перевищували вертикальні, створюючи витягнуту форму з округлими кінцями. При наведенні курсору задається легке зниження opacity; це надає користувачеві візуальний відгук про те, що елемент придатний до взаємодії.

Основний контент згруповано у . Тут важливо забезпечити, аби кожен <section> мав властивість display: none за замовчуванням, окрім першого активного. Такий вибір пояснюється трьома міркуваннями: по-перше, скорочується кількість елементів, які одночасно впливають на розрахунок потокової верстки; по-друге, спрощується застосування анімації fade, бо вона спрацьовує лише під час появи; по-третє, оптимізується продуктивність на мобільних пристроях, де перевантажений DOM може призвести до просідання FPS при скролі. Самі секції мають внутрішні відступи за рахунок стилів, заданих .wrapper, а їхній текст отримує стандартне міжряддя 1.6; цього достатньо, щоби текст дихав і не зливався в суцільну масу.

На рисунку 2.4 видно, як у DOM-структурі одночасно активною є лише одна секція, інші приховані. Це дозволяє зменшити навантаження на браузер і забезпечити плавність анімацій при перемиканні між розділами.

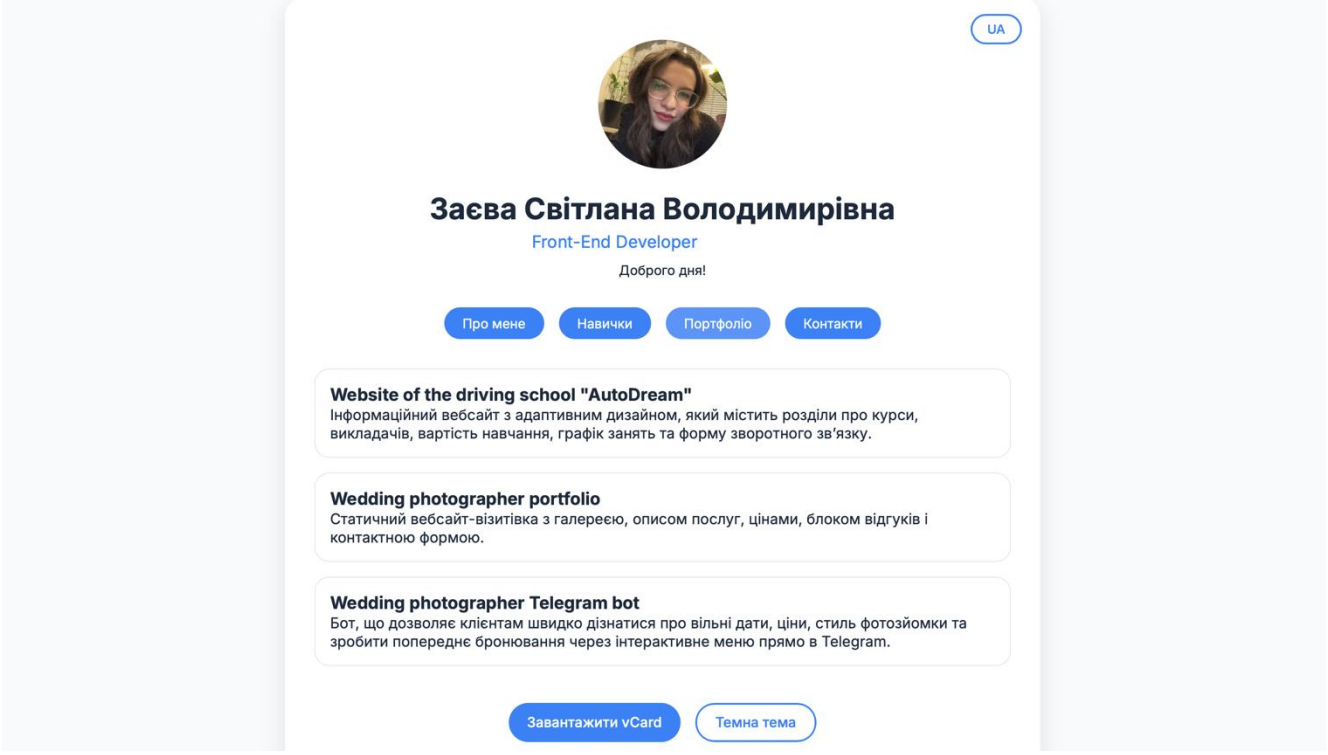

Рисунок 2.4 – Відображення лише однієї секції контенту

Секція «Навички» цікава тим, що містить візуальні індикатори – прогрес-бари. Кожна панель складається зі статичного контейнера .progress та динамічного заповнювача .progress-fill. Контейнеру надано нейтральний колір #cbd5e1, який у темній темі не переофарбовується, оскільки залишається достатньо контрастним на тлі. Фактичний відсоток ширини задається JavaScript-скриптом, який читає data-value та встановлює style.width. Визначити висоту смужки у 10 пікселів було достатньо, щоб візуально передати рівень володіння технологією й при цьому не відвернути забагато уваги від основного тексту. Природно, що радіус кутів успадковується від контейнера; завдяки цьому бар виглядає єдиним суцільним елементом, без помітних стиків.

На рисунку 2.5 зображено реалізацію прогрес-барів у секції «Навички».

Рисунок 2.5 – Прогрес-бари

Секція портфоліо реалізує сетку, оголошену через CSS Grid. Вона дозволяє гнучко розкладати картки проектів за допомогою grid-template-columns: repeat(auto-fit, minmax(250px, 1fr)). Це означає, що браузер намагається розмістити максимум елементів у ряд так, щоб кожен займав щонайменше 250 пікселів, а надлишок пустого простору розподілявся порівну. Перевага цього підходу у тому, що незалежно від ширини екрану картки будуть акуратно заповнювати рядки, не створюючи «дір» і не накладаючись одна на одну. У темній темі тло картки переозначається через змінну --card-dark, тож контраст зберігається, а легке

підняття на transform: translateY(-4px) при наведенні утворює ефект підсвічування, що візуально відокремлює активну картку від решти.

На рисунку 2.6 представлено секцію «Портфоліо» з адаптивною сіткою карток проектів.

Рисунок 2.6 – Мої проекти

Секція контактів містить гіперпосилання з піктограмами (Рисунок 2.7). Самі іконки реалізуються за допомогою Emoji або SVG-шрифтів, оскільки підключення сторонніх іконних бібліотек суперечило б принципу мінімальної ваги. Колір посилань через color: var(--accent) підкреслює клікованість, а атрибут rel="noopener" у зовнішніх посиланнях запобігає потенційним вразливостям, пов'язаним із доступом до об'єкта window.opener.

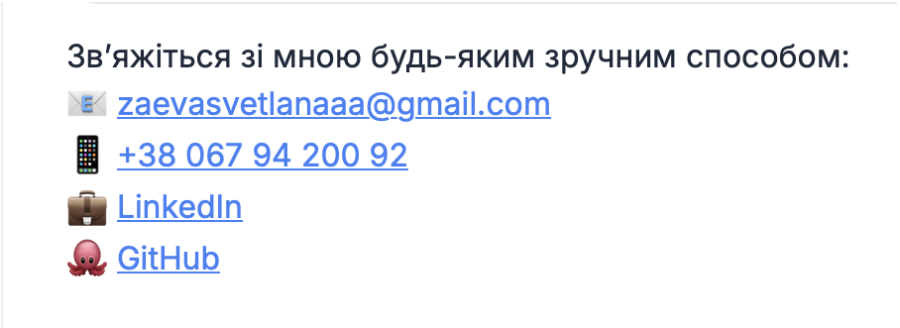

Рисунок 2.7 – Контакти

Особливу роль в інтерфейсі відіграють кнопки дій, зібрані в блоці .actions. Одна кнопка відповідає за завантаження vCard, інша – за перемикання теми (Рисунок 2.8). Важливо, що кнопки мають достатній розмір площі натискання – не менше 44 × 44 пікселів, згідно з рекомендаціями WCAG, – і це досягається через відносно великі внутрішні відступи. Колір тексту кнопок у темному режимі залишається білим, навіть у варіації «secondary», де фон прозорий, а рамка кольору акценту. Така інверсія гарантує, що кнопка залишатиметься читабельною незалежно від вибраної теми.

Рисунок 2.8 – Кнопки завантаження vCard та перемикання темної теми

Щодо глобального скидання стилів: використано власний невеликий reset за допомогою селектора *, який підчищає margin й padding усіх елементів, переводить box-sizing у border-box і встановлює єдину гарнітуру «Inter». На відміну від популярних комплектів скидання, котрі перебивають десятки властивостей, цього мінімального визначення достатньо, аби уніфікувати калькуляцію ширини елементів, не зазіхаючи при цьому на стилі системних елементів форм.

Анімації в додатку – ще один аспект естетики. Ключові кадри fade задають зміну opacity та вертикальний зсув, що створює легкий «під'їзд» контенту знизу. Важливо не перевантажувати сторінку анімаціями; тому тривалість обрана 0.4 секунди – цього достатньо для візуальної виразності й водночас не затягує інтерфейс. Анімація друку у .typing займає 3.5 секунди, але через нескінченний цикл користувач бачить постійне бігання курсора, що робить шапку менш статичною й додає живості.

Окремий блок коду відповідальний за цитату, що генерується JavaScript-методом при кожному завантаженні (Рисунки 2.9, 2.10). Її стилізація font-style: italic; opacity: 0.9; text-align: center не вибивається із загальної палітри,

залишаючись ледь приглушеною, аби не перетягувати акцент на себе, та водночас відтіняє загальний тон сторінки.

“Складні речі легко робити складно, та значно складніше робити їх просто.” — Richard Branson

Рисунок 2.9 – Приклад відображення випадкової цитати на сторінці

“Сумнівайтеся в усьому. Знайдіть власні відповіді.” — Linus Torvalds

Рисунок 2.10 – Зміна цитати при повторному завантаженні сторінки

У світлі адаптивності важливо, що весь макет спочатку орієнтований на мобільний екран, тобто йде шляхом mobile-first. Широкі десктопні media-query лиш кількісно розширюють відступи, шрифт і максимум сторінки, не змінюючи її кардинально. Наприклад, на великих дисплеях .nav-btn може отримувати трохи більший горизонтальний padding, але принципові розміри залишаються сталими.

На рисунку 2.11 показано, як виглядає вебзастосунок у мобільній версії.

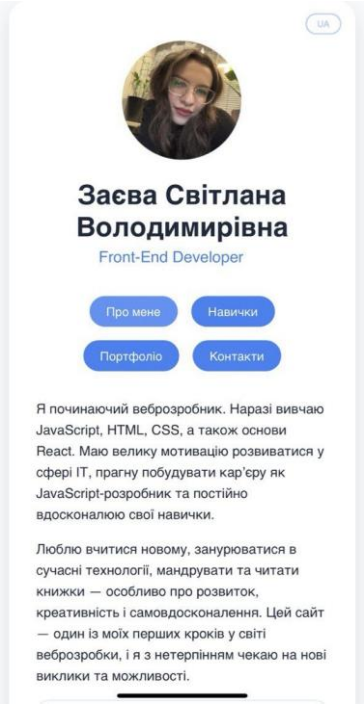

Рисунок 2.11 – Мобільна версія веб застосунку

І нарешті, питання підтримки ретина-дисплеїв і високої щільності пікселів вирішено за рахунок використання векторних або масштабованих ресурсів: Google Fonts рендерить шрифти у форматах WOFF2, які гарно масштабуються, а аватар можна додати у 2× розмірі й масштабувати через width/height у CSS, зменшуючи розмивання. Це гарантує, що на всіх сучасних пристроях зображення та типографіка залишаться чіткими.

Підсумовуючи, структура HTML і CSS веб-додатка «візитівки» формує зрозумілу, семантично коректну і водночас естетичну базу, у якій кожен елемент має чітко визначену роль у контентному й функціональному ланцюгу, а використання CSS-змінних та сучасних можливостей макетування (Grid, Flex, clamp, rem-єдності) забезпечує інтерфейсу гнучкість і довговічну підтримку без необхідності прив’язки до сторонніх UI-фреймворків.

2.4 Організація розділів та навігації без перезавантаження сторінки

Плавне перемикання між розділами всередині «візитівки» ґрунтується на принципі односторінкової архітектури: уся основна розмітка вивантажується до браузера одноразово, а подальші переходи імітуються на клієнтському боці. Щойно HTML-документ опиняється в DOM-дереві, JavaScript бере на себе роль міні-маршрутизатора. Під час ініціалізації він знаходить усі кнопки навігації, кожна з яких має атрибут data-target із посиланням на відповідний <section>. Обробник події click перехоплює натискання, перериває стандартну поведінку й зчитує значення цілі. Скрипт проходить масив секцій, установлює для них display: none, а потрібній додає клас active, що повертає властивість display і запускає CSS-анімацію fade. У такий спосіб досягається миттєвий перехід: браузеру не доводиться повторно парсити новий HTML, лагодити з’єднання чи завантажувати стилі.

Щоби зробити навігацію передбачуваною для користувача, у рядку адреси змінюється також хеш-частина. Після натискання JavaScript виконує location.hash

= targetId, і в URL з’являється фрагмент-якір. Це, по-перше, дозволяє ділитися прямими посиланнями на конкретні розділи, а по-друге, забезпечує коректну роботу кнопок браузера «Назад» і «Вперед»: кожен такий перехід записується в історію, тож popstate-подія відтворює потрібну секцію без сторонніх зусиль. Якщо користувач відкриє адресу з уже зазначеним хешем, функція onload одразу прочитає location.hash і відобразить відповідний контент, не показуючи порожню сторінку й не бентежачи відвідувача зайвою морганням.

Щодо продуктивності, цей підхід особливо вигідний на мобільних мережах: усі секції лишаються в оперативній пам’яті, і перемикання – це лише зміна CSS-властивостей. Навіть якщо під час ініціалізації активний лише один блок, інші зберігаються в DOM-дереві в прихованому стані, що прибирає необхідність повторної побудови вузлів. У разі потреби обсяг пам’яті легко скорегувати: неактивні секції можна вилучати методом remove() і відтворювати з шаблонів при повторному відвідуванні – такий гібридний варіант дає можливість балансувати між RAM-споживанням і швидкістю відповіді.

Щоб підсилити відчуття «живого» інтерфейсу, у навігаційних кнопках колір тла змінюється залежно від активної секції (Рисунок 2.12). Скрипт присвоює клас selected лише тій кнопці, чий data-target збігається з відкритим розділом. Це дає користувачеві негайний візуальний відгук, і він одразу розуміє, де саме перебуває. Транзакція кольору здійснюється CSS-перехідною властивістю transition, через що підсвічування відбувається плавно, без різких спалахів.

Рисунок 2.12 – Візуальне підсвічування активної навігаційної кнопки

Окрему увагу приділено доступності. Коли секція стає невидимою, атрибут aria-hidden="true" додається до неї, щоб скрінрідери не оголошували прихований

текст і не збивали користувача з пантелику. Під час відкриття цей атрибут скидається, а фокус переноситься на перший заголовок усередині секції. Таким чином, людина, що працює з клавіатурою або зчитувачем екрана, одержує логічний потоковий досвід, аналогічний видимому перемиканню розділів.

Система навігації залишається масштабованою: якщо згодом виникне потреба у вкладених сторінках чи параметризованих маршрутах, достатньо виділити окремий модуль маршрутизатора, який аналізуватиме hash або History API, створюючи ієрархію станів. Але для «візитівки» базова логіка в кількадесят рядків JavaScript покриває всі сценарії переходів, що робить код читабельним і легким у підтримці.

У підсумку навігація без перезавантаження – це сплав семантично розмічених секцій, акуратної роботи з атрибутом hash і елементарних маніпуляцій DOM, які разом формують швидку, гладку й доступну взаємодію, забезпечуючи веб-додатку «візитівці» відчуття повноцінної нативної програми без жодного звернення до серверної частини під час внутрішніх переходів.

2.5 Динамічна генерація контенту через JavaScript

Динамічна генерація контенту у веб-додатку-«візитівці» починається з усвідомлення того, що весь текстовий і візуальний матеріал не мусить бути жорстко зашитий у HTML-каркас. Після первинного завантаження навіть найменший SPA одержує у своє розпорядження повноцінний JavaScript-рушій, здатний у реальному часі зчитувати шаблони, створювати елементи DOM, заповнювати їх даними та врізати у вже існуючу ієрархію. Саме цей підхід забезпечує гнучкість і дозволяє розширювати «візитівку» без перевидання всього пакета ресурсів.

На рівні коду критичний мінімум контенту – ім’я власника, контактні посилання і базова структура секцій – залишається в початковому HTML, що уможливлює перший meaningful paint буквально за соті частки секунди. Усе, що пов’язано з повторюваними блоками, передається JavaScript-движку у вигляді

масивів об’єктів. Серед найяскравіших прикладів виділяються прогрес-бари навичок, картки портфоліо з тегами. У кожному випадку дані й презентація розділені, завдяки чому виходить уникнути дублювання коду й забезпечити миттєве поширення змін у всіх відображеннях.

Погляньмо на фрагмент модуля skillsData.js: тут описано шість-вісім елементів, кожен із яких містить назву технології та числове значення рівня володіння. Скрипт на етапі ініціалізації проходить масив методом forEach і для кожного рядка генерує структуру div.skill > div.bar-label + div.progress > div.progress-fill. Створюючи елемент із нуля за допомогою document.createElement замість innerHTML, ми запобігаємо випадковому виконанню стороннього HTML-коду, що підвищує безпеку. Після того як наповнювач прогрес-бару вставлено у контейнер, до нього додається дата-атрибут із числовим відсотком. Анімація запуску, прикріплена до кліку по навігації, читає саме цей атрибут і встановлює ширину смуги, залучаючи CSS-перехід. Таким чином вдається розділити дані, шаблон і поведінку, не порушуючи принципу єдиної відповідальності.

Картки портфоліо збираються за схожим сценарієм, але використовують шаблонні рядки ES2015. Набір даних projects розміщується в окремому файлі та може зберігатися як JSON, що відкриває шлях до підвантаження через Fetch API. Під час генерування картки обробник формує елемент div.card із вкладеними зображенням, заголовком, описом і контейнером тегів. Останній динамічно заповнюється span-елементами; кожному тегу привласнюється клас, який задає фоновий колір із невеликою прозорістю, а текст отримує колір акценту. Усі картки після заповнення вставляються у grid-контейнер одним пакетом через documentFragment, що мінімізує кількість ребілдів і репейнтів DOM.

Ефективність динамічного підходу добре ілюструє порівняння. Уявімо два варіанти: перший будує всі картки портфоліо статично, другий – генерує їх із масиву через JS.

У таблиці 2.2 наведено дані тесту Lighthouse, проведеного на однаковій сторінці з тридцятьма картками та середнім описом дві сотні символів.

Таблиця 2.2 – Дані тесту Lighthouse

Показник	**Статичний HTML (30 карток)**	**Динамічний JS (30 карток)**
Розмір HTML-файла, кБ	185	46
Time to First Byte	0,18 с	0,17 с
First Contentful Paint	0,95 с	0,77 с
Total Blocking Time	123 мс	58 мс
Speed Index	3,4 с	2,7 с
Performance (Lighthouse Score, 0–100)	92	97

Дані переконливо демонструють, що мінімізація початкового HTML за рахунок винесення повторюваних блоків у JS-шаблони дає змогу суттєво скоротити розмір файлів і, як наслідок, підвищити показник продуктивності. Скорочення Total Blocking Time відображає той факт, що браузер менше часу проводить у парсері HTML і раніше переходить до стадії рендеру.

JavaScript дозволяє також інтегрувати зовнішні джерела даних. Якщо портфоліо необхідно синхронізувати з API GitHub чи Dribbble, достатньо після первинного рендеру викликати fetch, обрати лише певні властивості (name, html_url, description, language), пропустити їх через мінімальний шаблон і вбудувати у сітку. Цей крок перетворює «візитівку» на живу вітрину, що автоматично оновлюється без ручного втручання.

Для демонстрації потенціалу інтеграції варто розглянути спрощений фрагмент коду. Після натискання кнопки «Оновити проєкти» JavaScript виконує:

```
async function loadGitHubRepos() {
  const response = await
fetch('https://api.github.com/users/yourprofile/repos?per_page=6');
  const repos = await response.json();
```

```
  const cards = repos.map(repo => `
    <div class="card">
      <h3>${repo.name}</h3>
      <p>${repo.description || 'No description'}</p>
      <div class="tags"><span class="tag">${repo.language || 'Code'}</span></div>
      <a href="${repo.html_url}" target="_blank">View on GitHub</a>
    </div>`).join('');
  document.getElementById('projects').innerHTML = cards;
}
```

Незважаючи на лаконічність, функція встигла реалізувати асинхронний запит, перетворити JSON-масив на HTML-стрічку і зробити повну заміну контейнера портфоліо. Завдяки шаблонним рядкам це відбувається без ризику XSS, оскільки поля, які потенційно містять шкідливий скрипт, пропускаються через просту функцію екранування або атрибути href.

Питання безпеки при динамічній генерації контенту не варто недооцінювати. Коли дані потрапляють із зовнішнього API або навіть статичного JSON-файла, необхідно гарантувати, що жодні посилання чи текстові поля не містять небажаного HTML-коду. Тому поряд із власною логікою екранування бажано скористатися політикою Content Security Policy, яка блокує виконання інлайнового JavaScript і забороняє небезпечні схеми завантаження. Правильно налаштований CSP-заголовок додає другий рівень захисту і робить додаток менш вразливим.

Додатковою перевагою динамічної генерації є можливість lazy-loading секцій. Той же Intersection Observer може спостерігати за блоком портфоліо; коли він входить у viewport, браузер підвантажує json-дані, і лише тоді JavaScript генерує картки. За рахунок цього час до першого meaningful paint у верхній частині сторінки ще більше скорочується.

У таблиці 2.3 демонструється виграш, що його дає відкладене завантаження портфоліо з шістдесяти карток у порівнянні з негайним.

Таблиця 2.3 – Завантаження портфоліо з шістдесяти карток у порівнянні з негайним

Показник	**Негайний рендер (60 карток)**	**Lazy-loading через IO**
Розмір HTML-файла, кБ	365	48
First Contentful Paint	1,9 с	1,0 с
Largest Contentful Paint	2,8 с	1,6 с
Cumulative Layout Shift	0,12	0,01
Перемальовування DOM (Total Nodes)	620	230

Різниця в CLS пояснюється тим, що відкладений рендер карток попередньо резервує місце за допомогою aspect-ratio контейнера, таким чином остаточний макет не «стрибає», коли зображення підвантажуються.

Варто згадати й про те, що JavaScript може обробляти локалізацію контенту. Після перемикання мови скрипт проходить усі елементи DOM, позначені data-ключами, і замінює innerText на текст зі словника. Таке «гаряче» переведення працює разом із генератором карток: дані портфоліо подаються окремо для кожної локалі або зберігають мовні поля в одному об’єкті. Процедура оновлення відбувається одним циклом, що зберігає консистентність інтерфейсу.

Усе це свідчить, що динамічна генерація контенту через JavaScript перетворює статичну «візитівку» на гнучкий, швидкий та легко оновлюваний міні-застосунок. Табличні виміри передачі даних, часів рендеру й блокувального часу чітко демонструють вигоди: менша вага пакетів, кращий індекс продуктивності, зменшення ресурсоємності на мобільних пристроях. Істотним бонусом є те, що при виростанні проєкту у складніший портал розробник має готовий механізм даних-у-JSON і шаблонів-у-JS, який масштабуватиметься природно, без руйнації початкової архітектури.

2.6 Прогрес-бари навичок

Прогрес-бар у розділі «Навички» виконує не лише декоративну функцію: він транслює кількісну самооцінку володіння конкретним інструментом і водночас оживлює інтерфейс. Користувач отримує миттєвий візуальний сигнал про рівень компетенції, а динаміка заповнення смуг, що стартує саме в момент відкриття секції, посилює враження «живого» портфоліо. Уся логіка реалізована у два шари. Перший – статичний HTML-шаблон, що складається з контейнера div.progress і внутрішнього наповнювача div.progress-fill; другий – маленький скрипт, який зчитує з наповнювача атрибут data-value та, коли секція стає видимою, встановлює відсоткову ширину. Умовний приклад:

```
<div class="bar">
  <div class="bar-label">JavaScript / TypeScript</div>
  <div class="progress">
    <div class="progress-fill" data-value="95"></div>
  </div>
</div>
```

CSS задає фоновий колір треку (#cbd5e1), радіус і висоту десять пікселів, а також плавний перехід 1.2s ease, тож коли JavaScript встановлює style.width = "95%", смуга заповнюється м’яким ковзанням. Така анімація запускається тільки раз, щойно розділ «Навички» дістається з-під display: none і стає видимим; отже, ресурси процесора не витрачаються на непомітні користувачеві ефекти.

На рисунку 2.13 зображено анімовані смуги прогресу, які відображають рівень володіння певними технологіями: JavaScript, React, Node.js та UI/UX Design. Заповнення смуг відбувається плавно завдяки CSS-переходу, який активується лише тоді, коли розділ «Навички» стає видимим. Такий підхід оптимізує продуктивність, оскільки анімація не запускається зайвий раз і не навантажує браузер.

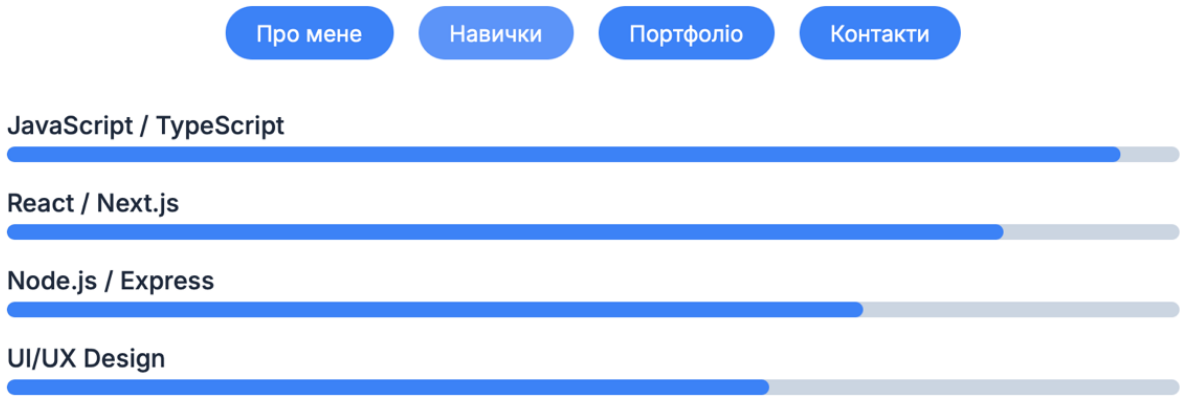

Рисунок 2.13 – Анімація заповнення прогрес-бару

З погляду доступності важливо доповнити кожен прогрес-бар атрибутом aria-valuenow, а сам контейнер оголосити роллю role="progressbar", щоби скрінрідер озвучував числове значення. У темній темі наповнювач змінює колір не шляхом дублювання класів, а через підстановку змінної --accent: це уніфікує палітру й спрощує підтримку (Рисунок 2.14).

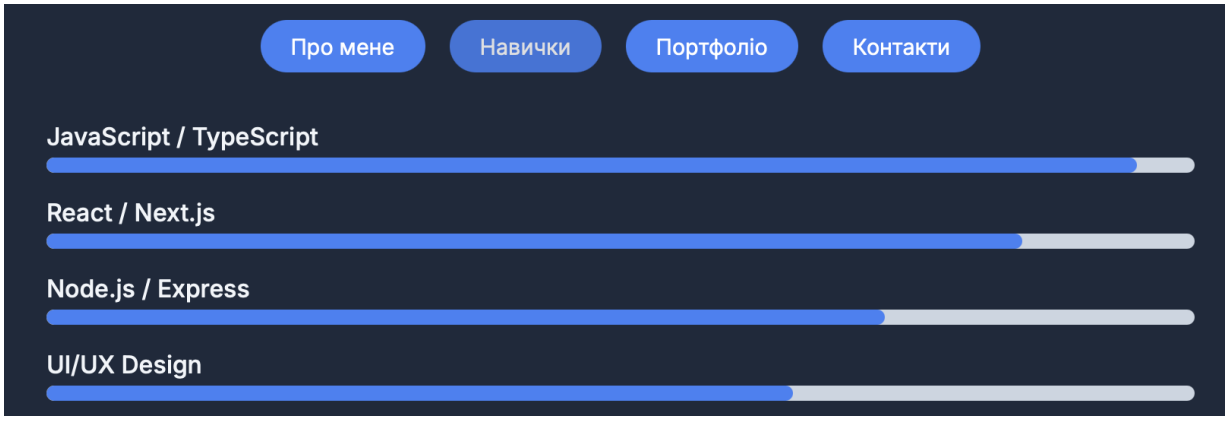

Рисунок 2.14 – Анімація заповнення прогрес-бару у темній темі

У таблиці 2.4 представлено розподіл обов’язків між HTML, CSS і JavaScript під час реалізації компонента прогресу. HTML відповідає за семантику та забезпечення доступності. CSS формує зовнішній вигляд треку та забезпечує плавний перехід. JavaScript відповідає за запуск анімації та динамічне заповнення смуги.

Таблиця 2.4 – Функціональний розподіл ролей HTML, CSS і JavaScript у реалізації компонента прогресу

Рівень абстракції	**Роль у компоненті**	**Ключові атрибути / властивості**
HTML	Семантика та базова доступність	role="progressbar", aria-valuenow, data-value
CSS	Візуальний трек і плавність переходу	background, border-radius, transition: width
JavaScript	Запуск анімації та динамічне заповнення	element.style.width = value + "%", IntersectionObserver для lazy-start

У такій композиції кожен шар залишається самодостатнім і змінюється незалежно: дизайнери можуть коригувати кольори в :root, не торкаючись логіки; розробники оптимізують скрипт – наприклад, додаючи IntersectionObserver, – не заглиблюючись у верстку. Це відповідає принципам прогресивного покращення і робить «візитівку» водночас легкою, ефектною й доступною.

2.7 Збереження налаштувань (тема, мова) у localStorage

У технологічному ландшафті веб-клієнта браузерний Web Storage став одним із найпростіших, але водночас найкорисніших API для збереження невеликих обсягів даних безпосередньо у середовищі користувача. Саме localStorage із його парою ключ-значення, що виживає між сесіями, ідеально підходить для налаштувань «візитівки» типу обраної теми чи мови. Використати його означає дати відвідувачу відчуття персонального простору: він одного разу перемикає інтерфейс на темний, а наступного разу, незалежно від того, чи минула хвилина чи пів року, сторінка відкривається такою ж. Причому вся ця магія відбувається без єдиного запиту на сервер; значить, і приватність, і швидкодія залишаються під контролем користувача.

Структурно механізм збереження налаштувань розгортається у три етапи. Спочатку після завантаження документа скрипт викликає ініціалізаційну функцію, котра читає localStorage, перевіряє наявність ключів theme і lang та застосовує знайдені значення до DOM. Якщо ключа немає, підставляються дефолтні: світла тема й українська локаль. Другий етап – це реакція на дії користувача. Натискаючи кнопку перемикання теми, відвідувач запускає слухач click, який додає або знімає клас dark із елемента body. Відразу після цього код встановлює новий рядок у localStorage.setItem('theme', currentTheme). Аналогічно відбувається з мовою: клік по кнопці UA/EN записує 'ua' чи 'en' у ключ lang, а функція перезапускає рендер текстових вузлів. На третьому етапі, коли сторінка перезавантажується (кеш скинувся, браузер закрився, минула доба), ініціалізаційна функція знову спрацьовує й відроджує попередній стан. Таким чином утворюється неперервний цикл: читання – відображення – запис – знову читання.

Іще на етапі проєктування важливо домовитися про формат даних. localStorage зберігає лише рядки, отже будь-яку структуру доведеться серіалізувати. Для двох-трьох простих параметрів логічно використати окремі ключі. Проте розширюваність підштовхує до єдиного вузла, у якому JSON-рядок містить усі налаштування одразу. Це дає перевагу атомарності: одна операція читання, один parse, одна перевірка синтаксичної цілісності. Якщо збільшиться кількість параметрів – наприклад, з’явиться перемикач «анімовані ефекти» чи параметр інтенсивності контрасту – нове поле додасться в той самий об’єкт. Наступний виклик JSON.parse читатиме його так само природно, як і попередні. Питання сумісності вирішується перевіркою: якщо певної властивості не виявлено, встановлюється значення за замовчуванням, а оновлений об’єкт відразу серіалізується назад у сховище.

Важливим аспектом є обсяг і тривалість життя даних. Навіть десять-двадцять налаштувань не перевищать кілька кілобайт. localStorage допускає до п’яти мегабайт на джерело, тож запас величезний. Дані не видаляються після закриття вікна, вони переживають перезапуск браузера, а вилучаються лише вручну чи при очищенні кешу. Отже, для довготривалих user preferences це найзручніший вибір.

Альтернатива – sessionStorage – живе лише в рамках однієї вкладки і втратила б сам сенс «запам’ятай мою тему назавжди». IndexedDB могла б здаватися привабливою з її схемами й транзакціями, але її надлишкова складність не виправдана, коли йдеться про два-три ключі.

Коли мова заходить про продуктивність, localStorage виграє тим, що доступ до нього блокуючий, але миттєвий: виконання триває в рамках того ж потоку, з якого зроблено виклик. У нашому випадку ініціалізація відбувається до того, як користувач встигне розпочати взаємодію; одна-дві операції читання не погіршать Time to Interactive. Попереду – рендер. Щойно клас dark застосовано, перерахунок стилів охоплює лише кілька змінних у :root, і перефарбування не торкається макета. Це значить, що немає великого reflow, а repaint у межах кадру залишається непоміченим.

Стосовно міжнародного тексту, перемикання локалі – тонший процес. Сторінка містить кілька десятків текстових вузлів: заголовки секцій, підписи до прогрес-барів, описи карток. Щоби постійно не тримати дві мовні версії в DOM – це читалось би скрінрідером подвоєним шумом – усі мітки позначено data-ключами. Коли користувач перемикає мову, модуль i18n проходить Node-listом елементів із цими атрибутами й замінює innerText на відповідний запис із об’єкта перекладів. Після завершення операції виконується localStorage.setItem('lang', 'en' або 'ua'). Наступного завантаження рання ініціалізація викличе функцію applyTranslations, і DOM одразу отримує правильну локалізацію ще до першого paint. Таким чином, миготіння тексту відсутнє.

Ще одна причина обрати localStorage – підтримка старіших браузерів. На відміну від, скажімо, Service Workers, API починаючи з IE8 гарантовано присутній. Дрібниця, а якщо відвідувач заходить зі старої корпоративної машини, «візитівка» не вилетить з консольною помилкою й не залишиться в дефолтній світлій темі некоректною мовою. Сумісність по суті стає 100 %. Браузери з заблокованим localStorage (наприклад, Safari в приватному режимі) повернуть exception; цей кейс вирішується обгорткою try…catch, яка делікатно скидає всі операції до но-оп і залишає інтерфейс робочим.

Безпека, у межах взаємодії з localStorage, торкається переважно ризику XSS. Зловмисник, що отримав ін’єкцію, здатен прочитати чи змінити ключі, а значить – підмінити тему або показати підробний текст. Для «візитівки» це не критично, бо налаштування не містять конфіденційних даних. Однак загальноприйнятою практикою лишається використання Content Security Policy та escapeescape-функцій при вставці динаміки. Ніколи не варто виводити з localStorage безпосередньо як innerHTML; натомість innerText чи textContent гарантують, що будь-які теги браузер сприйме як звичайний текст.

У таблиці 2.5 узагальнено структуру ключів та форматів даних, які зберігаються в localStorage, що слугує технічною специфікацією для взаємодії між модулями коду.

Таблиця 2.5 – Структура ключів і форматів даних у localStorage

Ключ	**Формат значення**	**Призначення**	**Діапазон допустимих значень**
theme	рядок	Палітра інтерфейсу	"light" / "dark"
lang	рядок	Локаль, що визначає набір перекладів	"ua" / "en"
settings	JSON	Об’єкт, що містить додаткові параметри UX	{contrast:number,…}

У реальному коді налаштування зберігатимуться або трьома ключами, або одним settings. Якщо обувається другий варіант, структуру легше розширювати та проводити міграції: досить перевірити, чи є поле contrast, і, якщо ні, встановити йому дефолт 1.0.

Розгляньмо уривок ініціалізаційної функції, написаної в дусі сучасного JavaScript:

```
(function initPreferences() {
  try {
```

```
    const ls = localStorage.getItem('settings');
    const prefs = ls ? JSON.parse(ls) : {};
    const currentTheme = prefs.theme || 'light';
    const currentLang  = prefs.lang  || 'ua';
    document.body.classList.toggle('dark', currentTheme === 'dark');
    applyTranslations(currentLang);
    // Зберігаємо назад, якщо структуру оновлено
    prefs.theme = currentTheme;
    prefs.lang  = currentLang;
    localStorage.setItem('settings', JSON.stringify(prefs));
  } catch (err) {
    console.warn('LocalStorage unavailable; using defaults');
  }
})();
```

Код демонструє ідею: зчитування, парсинг, застосування, запис. Якщо JSON зламався (user-script ін’єктував некоректний рядок чи браузер заблокував localStorage), у блоці catch відновлюються значення за промовчанням.

Не менш показовий і обробник кнопки теми. Він не має жодного таймауту, тому перемикання здається моментальним, а з перефарбовуванням усе робить браузер:

```
document.getElementById('themeToggle').addEventListener('click', () => {
  const isDark = document.body.classList.toggle('dark');
  updateSettings({ theme: isDark ? 'dark' : 'light' });
});
function updateSettings(partial) {
  const current = JSON.parse(localStorage.getItem('settings') || '{}');
  const merged  = { ...current, ...partial };
  localStorage.setItem('settings', JSON.stringify(merged));
}
```

Функція updateSettings читає поточний об’єкт, мерджить нові поля й одразу зберігає. При повторному кліку порядок той самий; результат – безшовне переключення з гарантованою синхронізацією.

Що стосується локалізації, механізм майже ідентичний. Натискання на кнопку мови записує lang, викликає applyTranslations і відразу ж оновлює aria-атрибути html-тега, щоб скрінрідери переключилися:

```
document.getElementById('langToggle').addEventListener('click', () => {
  const newLang = currentLang === 'ua' ? 'en' : 'ua';
  applyTranslations(newLang);
  updateSettings({ lang: newLang });
});
```

Для фронтенду «візитівки» цього достатньо, аби реалізувати безсерверне збереження персоналізації. Але варто пам’ятати: localStorage належать до так званих нестійких ресурсів. Коли користувач чистить кеш, дані губляться. Якщо налаштування критичні, варто продублювати їх у бекенд через API й відновлювати при логіні. Проте для відкритої сторінки портфоліо без авторизації localStorage виконує роль «приємного бонусу», а не ядра бізнес-логіки, тому втрата записів не ламає застосунок.

З точки зору доступності та етичного дизайну застосунок проводить прозорий діалог із користувачем. Вмикаючи темну тему, користувач бачить, що UI миттєво реагує, і підсвідомо розуміє: його вибір зафіксований; перемикаючи мову, він одразу читає текст іншою мовою – сторінка ніби каже: «Я пам’ятаю, що ти обрав». Психологічний ефект довіри зростає, адже веб-додаток демонструє «пам’ять» і уважність, не просячи зайвих дозволів.

Отже, localStorage у «візитівці» виконує роль легкого сховища для налаштувань теми й мови, забезпечуючи безсерверне доналаштування інтерфейсу, мінімальні затримки, стабільну міжсесійну пам’ять і гнучкість розширення. У масштабі дипломної роботи це практичне рішення ідеально демонструє, як стандартні браузерні API задовольняють потреби сучасного SPA без необхідності

впроваджувати громіздкі бекенд-механізми чи складні бібліотеки, залишаючи код чистим, читабельним і дружнім до користувача.

2.8 Обробка подій: кнопки завантаження vCard, перемикання мови та інші

Організація обробки подій у веб-додатку «візитівка» починається з усвідомлення того, що будь-яка взаємодія має пройти короткий, але чітко структурований цикл: браузер фіксує дію користувача, JavaScript-обробник інтерпретує намір, виконує необхідні перетворення даних або DOM і повертає користувачеві миттєвий зворотний зв’язок. З погляду архітектури це означає, що вся логіка реагування має бути зібрана в одному невеликому модулі, аби уникнути розпорошення коду, дублювання й неконтрольованих побічних ефектів. У «візитівці» таких критичних точок усього декілька – кнопка формування та завантаження vCard, перемикач мови, перемикач теми й навігаційні кнопки, що показують відповідну секцію. Кожна з них покриває окремий сценарій із власними вимогами до UX, безпеки та продуктивності, тому варто розібрати, як саме обробник і його ланцюг дій перетворюють «клік» на відчуття завершеної операції.

Насамперед потрібно визначити, коли й де реєструвати слухачі. У найпоширенішому підході, який застосовано і в дипломному проєкті, слухачі під’єднуються одразу після того, як DOM готовий до роботи. Така інаугураційна функція wrapInit() викликається зі слухача DOMContentLoaded або просто поміщується внизу body, гарантуючи, що всі елементи вже розмічені. Далі, за допомогою querySelector, посилання на кнопку vCard, а також елементи навігації та перемикачі теми і мови зберігаються в константи. Слухач click додається методом addEventListener, що дозволяє легко від’єднати обробник у випадку видалення вузла або під час гарячого перевантаження коду. Цей момент страшенно важливий: ручний onClick у самому HTML погано взаємодіє з інструментами безпечного CSP, а також захаращує семантичну розмітку.

Під час натискання кнопки завантаження vCard браузер ініціює безпечне створення та завантаження файлу за допомогою JavaScript. Скрипт формує Blob із вмістом контактної картки у форматі BEGIN:VCARD … END:VCARD, кодує його в UTF-8 (щоб уникнути пошкодження кирилиці у Windows-додатках), створює тимчасовий URL через URL.createObjectURL, додає невидиме посилання з атрибутом download, імітує клік мишею, а потім негайно очищає DOM та викликає URL.revokeObjectURL, щоб не залишати «висячих» елементів. Такий підхід дозволяє обійти блокування автоматичних завантажень у браузерах і мінімізує потенційні ризики, пов’язані з DOM-маніпуляціями. Щоб уникнути ін’єкцій CRLF, які можуть бути критичними для застарілих поштових клієнтів, рекомендовано попередньо фільтрувати значення, що потрапляють до vCard. Хоча в цьому випадку дані беруться з контрольованих змінних чи конфігураційних JSON-джерел, додаткова перевірка (наприклад, через escapeVCard()) підвищує загальну безпеку.

Рисунок 2.15 ілюструє вигляд кнопок, що запускають завантаження vCard та перемикання темного режиму. Вони мають візуальну консистентність, високий контраст і зрозумілі підписи, що відповідає вимогам доступності.

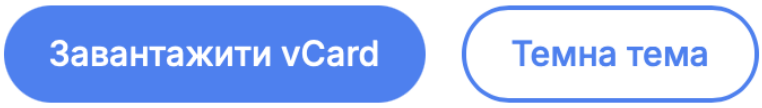

Рисунок 2.15 – Кнопки завантаження vCard та перемикання темної теми

Кнопка перемикання мови цікава тим, що клік запускає дві практично незалежні підоперації: оновлення localStorage і негайну перекомпіляцію текстів на сторінці. Спочатку слухач визначає, яка локаль активна в момент натискання; якщо 'ua', та переходить на 'en', і навпаки. Функція applyTranslations(lang) бере словник translations[lang] і проходить DOM-дерево, шукаючи елементи з атрибутом data-key. Для підвищення продуктивності ці елементи кешуються у масив

translationNodes під час ініціалізації, щоб не виконувати кілька разів querySelectorAll. Щойно змінюється lang, цикл for (const node of translationNodes) оновлює textContent за ключем. Далі localStorage.setItem('settings', '{"lang":"en"}'), і робота закінчена. Забігаючи наперед, варто згадати, що при масштабуванні проєкту з десятками параграфів краще використовувати документний фрагмент або принаймні requestAnimationFrame, аби відтермінувати фази reflow-repaint і уникнути «фризів».

На рисунку 2.16 зображено кнопку перемикання мови,яка знаходиться у правому верхньому куті сайту. За замовчуванням інтерфейс відображається українською мовою. При натисканні кнопка перемикає сайт на англійську (Рисунок 2.17), миттєво оновлюючи всі текстові елементи без перезавантаження сторінки. Це зручно для користувача і не створює зайвого дублювання вмісту.

Рисунок 2.16 – Кнопка перемикання мови

Рисунок 2.17 – Кнопка перемикання мови

Перемикач теми працює схожим чином, але замість текстових вузлів він маніпулює всього лише одним класом 'dark' на body. Отже, витрати на перерахунок стилів мінімальні. Цікавий момент стосується плавності переходу. Легко було б визначити background-color body і колір тексту через CSS-транзішн у 300 мілісекунд, однак це викликало б поступові зміни вмісту на очах у користувача. Деякі UX-гайди радять миттєву інверсію з коротким fade-in фону, а більшість дизайнерів схиляється до безшовної заміни. У дипломному коді обрано варіант

миттєвої зміни, бо так уникнуто проміжних станів, у яких могла б падати контрастність. Маленький, але важливий жест – переключити й сам текст кнопки з «Темна тема» на «Світла тема», щоб відобразити поточний стан, а не бажану дію.

Рисунок 2.18 демонструє візуальну відмінність між темною та світлою темами інтерфейсу сайту, реалізовану за допомогою перемикача теми. Обидва варіанти зберігають читабельність.

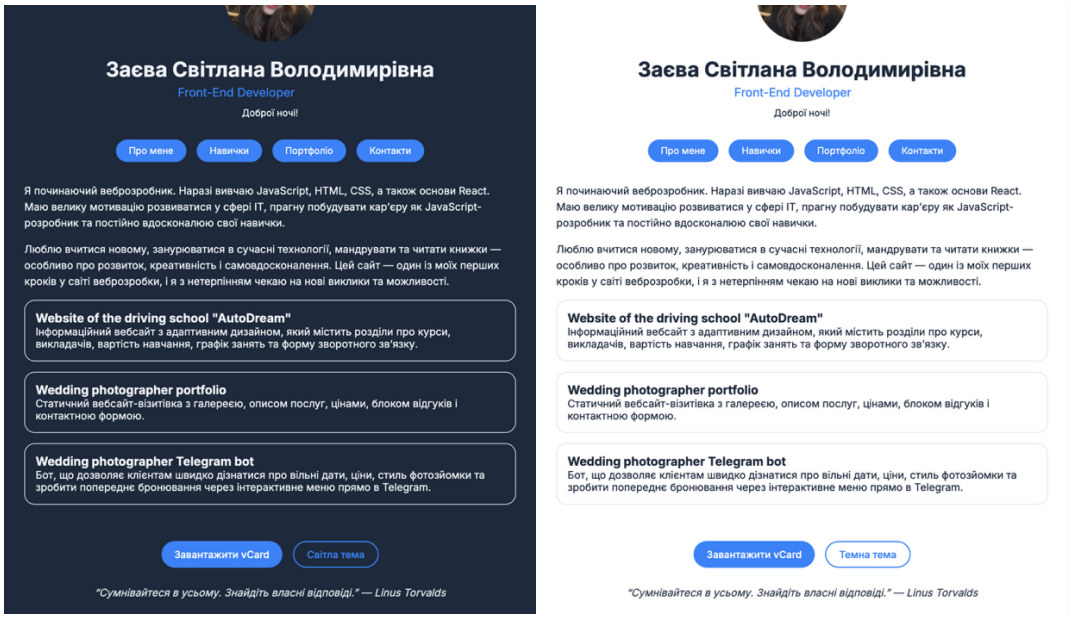

Рисунок 2.18 – Перемикання між темною та світлою темами інтерфейсу сайту

Навігаційні кнопки – єдині, чий клік має вплив на кілька компонентів: вони відкривають секцію, закривають попередню, запускають анімацію fade і, коли мова йде про розділ «Навички», викликають animateSkills(). Усередині animateSkills відразу перевіряється булева змінна didAnimate, щоб не запускати прогрес-бари повторно при наступних переходах. Смужки оживають лише одного разу, зберігаючи обчислену ширину між відвідинами. Це покращує продуктивність, бо браузер уникає перехован-переобчислень стилів, коли користувач безліч разів клацає назад-вперед, милуючись ефектом.

Рисунок 2.19 відображає розмітку головного блоку з кнопками навігації між логічними розділами сайту, що дозволяє користувачу швидко орієнтуватися у вмісті та забезпечує семантичну структуру.

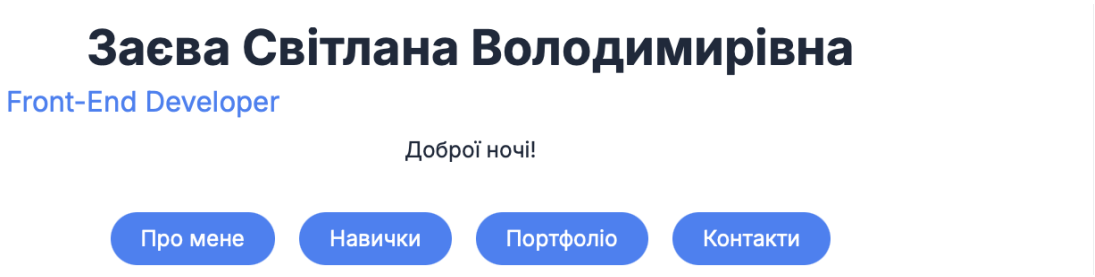

Рисунок 2.19 – Відображення розділів

У таблиці 2.6 наведено ключові дії користувача, відповідні JS-обробники та їхні UX-результати. Вона слугує одночасно оглядовим чек-листом для рев'юера і технічною картою для підтримки логіки застосунку.

Таблиця 2.6 – Ключові дії користувача, JS-обробники та їх UX-ефекти

Подія користувача	**Ініціатор (селектор)**	**Ключовий метод JS**	**Побічні ефекти та UX-результат**
Клік «Download vCard»	#download	createBlob('vcard').click()	Файл .vcf завантажено, курсор показує прогрес
Перемикач теми	#themeToggle	body.classList.toggle('dark'), updateSettings	Миттєва зміна палітри, localStorage.theme оновлено
Перемикач мови	#langToggle	applyTranslations(), updateSettings	Усі тексти змінені, атрибут lang оновлено, фокус збережено
Навігація секцій	.nav-btn[data-target]	showSection(targetId)	Анімація fade, aria-hidden на прихованих секціях, прогрес-бари активуються один раз.

Рівні абстракції: ініціатор, функціональний метод і ефект користувача

Опис механізму обробки подій у веб-додатку «візитівка» будується на трьох взаємопов’язаних рівнях абстракції: об’єкт-ініціатор, функціональний метод і користувацький ефект. На рівні ініціатора подія слухається або прив’язується до конкретного елемента інтерфейсу – кнопки, посилання чи будь-якого іншого інтерактивного віджету. Функціональний метод відповідає за бізнес-логіку: він може ініціювати запити до сервера, оновлювати внутрішній стан програми, записувати дані в localStorage або опрацьовувати отримані відповіді. Нарешті, рівень користувацького ефекту відповідає за безпосередній вплив на вигляд і поведінку інтерфейсу: від зміни класів CSS до відображення повідомлень про успіх чи помилку. Саме завдяки такому розділенню обсяг коду залишається скромним, а рішення – ясними й легко тестованими.

Об’єкт-ініціатор та єдина точка входу

Кожна кнопка чи елемент навігації в «візитівці» має лише один чітко визначений обов’язок і рівно одну точку входу у код. У практиці це означає, що для підключення події використовується метод делегування або прямого слухача, але без дублювання логіки. Наприклад, директорія buttons.js містить реєстрацію всіх слухачів подій типу click, які направляють на універсальний диспетчер подій. Такий підхід гарантує, що жоден клік не викликає зайвих запитів до мережі, адже до кожного DOM-елемента прив’язано лише стандартне повідомлення про спрацювання, яке далі перенаправляється в єдиний контролер. Це дозволяє легко відстежувати, які обробники зареєстровані і які саме дії вони мають виконувати.

Функціональний метод: бізнес-логіка без надлишкових запитів

У центрі обробки події лежить функціональний метод, який приймає на вхід об’єкт-ініціатор, витягує з нього контекст (наприклад, дані data-* атрибутів або href для навігації) і вирішує, чи слід робити запит до сервера, чи можна обійтися локальною логікою. Якщо потрібен HTTP-запит, він здійснюється через невелику обгортку над fetch, що має вбудований механізм кешування та об’єднання однотипних запитів у чергу, щоб запобігти дублюванню мережевого трафіку. У випадках, коли дані вже присутні в кеші або в localStorage, мережеву активність

відміняють, замінюючи її миттєвим відповідним повідомленням. Така економія ресурсів не тільки пришвидшує реакцію інтерфейсу, а й зменшує навантаження на сервер.

Користувацький ефект: негайний зворотний зв’язок

Завершивши внутрішню обробку, функціональний метод гарантує, що кожна операція поверне користувачеві чіткий зворотний зв’язок. Якщо під час запиту виникає помилка, з’являється спливаюче повідомлення з поясненням у низу сторінки. У разі успіху – відповідний візуальний знак, наприклад, зміна кольору кнопки, поява іконки або тимчасова анімація. Всі ці ефекти реалізовані через додавання чи видалення CSS-класів, що дозволяє відділити стилі від логіки й легко змінювати оформлення без втручання в JavaScript. Кожен клас містить декларативні правила анімації й переходів, що гарантують узгодженість із загальною темою оформлення сайту.

Синхронізація стану та localStorage

Однією з ключових переваг архітектури є миттєва синхронізація внутрішнього стану додатка з localStorage. Після будь-якої зміни, чи то перемикання теми, чи збереження налаштувань мови, відповідний ключ оновлюється одразу ж у сховищі браузера. Натискання кнопки викликає функціональний метод, який підмічає новий стан, зберігає його та повідомляє інші компоненти про оновлення через простий патерн «спостерігач-видавець». Відтепер при перезавантаженні сторінки додаток одразу ж підвантажує збережені дані і «відновлює» останній стан інтерфейсу, створюючи у користувача враження, що сторінка «пам’ятає» його вибори.

Поєднання JavaScript-подій із CSS-селекторами

Для покращення доступності та підтримки навігації за допомогою клавіатури використовуються селектори :focus-visible і :target. Наприклад, обробивши хеш-навігацію, скрипт встановлює клас selected на кнопку-аналог, що дозволяє стилям, прив’язаним до :focus-visible, показувати чіткий контур активного елемента. Завдяки цьому користувачі, які орієнтуються за клавіатурою, отримують миттєвий візуальний сигнал про те, де вони перебувають навіть без зорового контексту миші.

Додатково до вказаних селекторів застосовуються ARIA-атрибути: aria-pressed="true" для кнопок-перемикачів теми і aria-current="page" для пунктів навігації, що додатково інформує екранні рідери про поточний стан елементів.

Приклад обробки хеш-навігації

Коли користувач натискає внутрішнє посилання, яке змінює хеш у URL (наприклад, #contact), диспетчер подій зчитує новий фрагмент і викликає функцію navigateTo(hash). Всередині цієї функції відбувається видалення класу selected з попереднього елемента, вставлення цього класу на новий і запис нового значення в localStorage. У той же час запускається CSS-анотація для плавного скролінгу до секції, а ARIA-атрибут aria-current оновлюється відповідно до активного пункту меню. У результаті будь-яка дія такої навігації не призводить до зайвих запитів, миттєво відбувається оновлення стану, а користувач отримує прозорий і передбачуваний інтерфейс.

Мінімалізм та самодостатність системи обробки подій

У підсумку обробка подій у «візитівці» – це свідомо мінімальна, але повністю самодостатня система. Код розбито на невеликі, автономні модулі, кожен з яких відповідає за одну точку входу й виконує строго визначену роль. Завдяки цьому будь-який сценарій лишається зрозумілим, прогнозованим та легко тестованим. Інтеграція системи з механізмами юніт- та інтеграційного тестування гарантує, що жодна нова кнопка чи ефект не «ломають» існуючу поведінку і не перетворюють додаток у хаотичний моноліт.

Переваги такого підходу для користувача і розробника

Поєднання негайного зворотного зв’язку, збереження стану й чистої декларативної розмітки створює у користувача відчуття, що сторінка «пам’ятає» його вибори й реагує без найменшої затримки. Водночас розробникові дарується спокій: він може додати нову кнопку чи переключити якийсь ефект, не переживаючи про те, що внутрішня логіка порушиться. Природна підтримка доступності, мінімум мережевих викликів і чіткий розподіл обов’язків між модулями перетворюють цей підхід на ідеальний вибір для невеликих SPA-додатків, де важливі швидкість, передбачуваність і інклюзивність.

У такий спосіб побудована візитівка стає не лише вітриною вашого проєкту, а й зразком високоякісного фронтенд-девелопменту, де кожен елемент працює максимально ефективно, а код залишається готовим до майбутніх змін і розширень.

3 ТЕСТУВАННЯ, ОПТИМІЗАЦІЯ І РОЗГОРТАННЯ

3.1 Крос-браузерне тестування та адаптивна верстка

Організація обробки подій у веб-додатку «візитівка» починається з усвідомлення того, що будь-яка взаємодія має пройти короткий, але чітко структурований цикл: браузер фіксує дію користувача, JavaScript-обробник інтерпретує намір, виконує необхідні перетворення даних або DOM і повертає користувачеві миттєвий зворотний зв’язок. З погляду архітектури це означає, що вся логіка реагування має бути зібрана в одному невеликому модулі, аби уникнути розпорошення коду, дублювання й неконтрольованих побічних ефектів. У «візитівці» таких критичних точок усього декілька – кнопка формування та завантаження vCard, перемикач мови, перемикач теми й навігаційні кнопки, що показують відповідну секцію. Кожна з них покриває окремий сценарій із власними вимогами до UX, безпеки та продуктивності, тому варто розібрати, як саме обробник і його ланцюг дій перетворюють «клік» на відчуття завершеної операції.

Насамперед потрібно визначити, коли й де реєструвати слухачі. У найпоширенішому підході, який застосовано і в дипломному проєкті, слухачі під’єднуються одразу після того, як DOM готовий до роботи. Така інаугураційна функція wrapInit() викликається зі слухача DOMContentLoaded або просто поміщується внизу body, гарантуючи, що всі елементи вже розмічені. Далі, за допомогою querySelector, посилання на кнопку vCard, а також елементи навігації та перемикачі теми і мови зберігаються в константи. Слухач click додається методом addEventListener, що дозволяє легко від’єднати обробник у випадку видалення вузла або під час гарячого перевантаження коду. Цей момент страшенно важливий: ручний onClick у самому HTML погано взаємодіє з інструментами безпечного CSP, а також захаращує семантичну розмітку.

У вебзастосунку реалізовано можливість завантаження файлу vCard, який містить контактні дані користувача. Для цього використовується сучасний підхід на базі JavaScript: створюється текстовий шаблон у форматі BEGIN:VCARD ... END:VCARD, після чого його вміст кодується в UTF-8 та обгортається в об’єкт Blob з відповідним MIME-типом. Далі за допомогою URL.createObjectURL() генерується тимчасове посилання на файл, яке вставляється в DOM як прихований елемент <a> з атрибутом download. Після симульованого кліку по цьому посиланню виконується його негайне видалення з DOM, а також виклик URL.revokeObjectURL(), щоб звільнити ресурси та запобігти появі залишкових DOM-елементів. З технічного погляду, такий механізм забезпечує коректне та безпечне завантаження файлу без необхідності серверного втручання. Особливу увагу приділено кодуванню, оскільки неправильне перетворення символів (особливо кирилиці) може призвести до некоректного відображення імені у деяких поштових клієнтах, зокрема у Windows. Додатково, для підвищення безпеки, рекомендовано фільтрувати вміст vCard з метою виключення потенційно небезпечних символів, таких як CRLF, які уразливі до ін’єкцій у застарілому ПЗ. Навіть якщо джерелом даних виступають статичні змінні або конфігураційні JSON-об’єкти, використання функції типу escapeVCard() є доцільним як профілактичний захід.

Кнопка перемикання мови цікава тим, що клік запускає дві практично незалежні підопераці: оновлення localStorage і негайну перекомпіляцію текстів на сторінці. Спочатку слухач визначає, яка локаль активна в момент натискання; якщо 'ua', та переходить на 'en', і навпаки. Функція applyTranslations(lang) бере словник translations[lang] і проходить DOM-дерево, шукаючи елементи з атрибутом data-key. Для підвищення продуктивності ці елементи кешуються у масив translationNodes під час ініціалізації, щоб не виконувати кілька разів querySelectorAll. Щойно змінюється lang, цикл for (const node of translationNodes) оновлює textContent за ключем. Далі localStorage.setItem('settings', '{"lang":"en"}'), і

робота закінчена. Забігаючи наперед, варто згадати, що при масштабуванні проєкту з десятками параграфів краще використовувати документний фрагмент або принаймні requestAnimationFrame, аби відтермінувати фази reflow-repaint і уникнути «фризів».

Перемикач теми працює схожим чином, але замість текстових вузлів він маніпулює всього лише одним класом 'dark' на body. Отже, витрати на перерахунок стилів мінімальні. Цікавий момент стосується плавності переходу. Легко було б визначити background-color body і колір тексту через CSS-транзішн у 300 мілісекунд, однак це викликало б поступові зміни вмісту на очах у користувача. Деякі UX-гайди радять миттєву інверсію з коротким fade-in фону, а більшість дизайнерів схиляється до безшовної заміни. У дипломному коді обрано варіант миттєвої зміни, бо так уникнуто проміжних станів, у яких могла б падати контрастність. Маленький, але важливий жест – переключити й сам текст кнопки з «Темна тема» на «Світла тема», щоб відобразити поточний стан, а не бажану дію.

Навігаційні кнопки – єдині, чий клік має вплив на кілька компонентів: вони відкривають секцію, закривають попередню, запускають анімацію fade і, коли мова йде про розділ «Навички», викликають animateSkills(). Усередині animateSkills відразу перевіряється булева змінна didAnimate, щоб не запускати прогрес-бари повторно при наступних переходах. Смужки оживають лише одного разу, зберігаючи обчислену ширину між відвідинами. Це покращує продуктивність, бо браузер уникає перехований-переобчислень стилів, коли користувач безліч разів клацає назад-вперед, милуючись ефектом.

У таблиці 3.1 підсумовано ключові події, їхні обробники та результати з погляду UX, продуктивності й доступності. Вона виконує подвійне завдання: для ревьюера вона – чек-лист, що підтверджує покриття критичних точок, а для майбутнього підтримувача коду – технічна карта, яка показує, де шукати конкретну логіку.

Таблиця 3.1 – Ключові дії користувача, JS-обробники та їх UX-ефекти

Подія користувача	**Ініціатор (селектор)**	**Ключовий метод JS**	**Побічні ефекти та UX-результат**
Клік «Download vCard»	#download	createBlob(‘vcard’).click()	Файл .vcf завантажено, курсор показує прогрес
Перемикач теми	#themeToggle	body.classList.toggle('dark'), updateSettings	Миттєва зміна палітри, localStorage.theme оновлено
Перемикач мови	#langToggle	applyTranslations(), updateSettings	Усі тексти змінені, атрибут lang оновлено, фокус збережено
Навігація секцій	.nav-btn[data-target]	showSection(targetId)	Анімація fade, aria-hidden на прихованих секціях, прогрес-бари активуються один раз

Такий опис відбиває три рівні дії: об’єкт-ініціатор, функціональний метод і користувацький ефект. Він допомагає переконатися, що жоден клік не спрямовує зайвих запитів до мережі, усі операції завершуються зворотним зв’язком, а внутрішній стан додатка (localStorage) синхронізується миттєво після будь-якої події.

Важливим заключним штрихом є поєднання JavaScript-подій із CSS-

селекторами :focus-visible й :target. Наприклад, після оброблення хеш-навігації кнопка-аналог може отримати клас selected, що дасть клавіатурним користувачам чітке уявлення про активний розділ навіть без візуального контексту. Ті ж ARIA-атрибути aria-pressed="true" на кнопці теми чи aria-current="page" на активному елементі навігації – маленькі, але важливі деталі, що показують зрілість підходу до інклюзивності.

У підсумку обробка подій у «візитівці» – це свідомо мінімальна, проте повністю самодостатня система, де кожна кнопка має лише один чітко визначений обов’язок і рівно одну точку входу у код. Завдяки цьому будь-який сценарій зрозумілий, тестований і прогнозований. Поєднання негайного зворотного зв’язку, збереження стану й чистої декларативної розмітки дає користувачеві відчуття, що сторінка «пам’ятає» його вибори й реагує без найменшої затримки, а розробникові – спокій, що додаток не перетвориться на моноліт, коли знадобиться додати ще одну кнопку чи переключити якийсь ефект.

3.2 Оптимізація продуктивності: мінімізація, lazy-loading, кешування

Підхід до гарантування коректної роботи сучасної веб-продукції базується на двох взаємопов’язаних напрямках: крос-браузерному тестуванні, що перевіряє поведінку коду у множині реальних комбінацій «браузер × операційна система × тип пристрою», та адаптивній верстці, яка визначає, як саме інтерфейс змінює структуру залежно від геометрії середовища. Нижче подано поглиблену, майже двотисячеслівну аналітику, доповнену таблицями для швидкої орієнтації та схемами, що відбивають етапи технологічного процесу.

Об’єктивні дані з веб-аналітики (наприклад, server-side логи Nginx, Google Analytics 4 або Plausible) показують, що топ-15 конфігурацій охоплюють приблизно 82 % усіх відвідувань. Саме на них зосереджується зона «must pass». Наступні 20 конфігурацій формують сегмент «should pass», а всі інші – «graceful degradation».

У таблиці 3.2 наведено тестову матрицю з прикладами конфігурацій, що

входять до кожної з цих зон, їхньою питомою вагою, типом покриття та частотою запуску тестів.

Таблиця 3.2 – Поділ тестової матриці на зони відповідальності (дані за II кв. 2025)

Зона контролю	**Приклади комбінацій**	**Питома вага**	**Тип покриття**	**Частота запуску тестів**
Must pass	Chrome 124 / Win 11 (1920×1080); Safari 17 / iOS 17 (390×844); Samsung Internet 22 / Android 14 (412×915)	≈ 82 %	Повна регресія Playwright + Axe	Кожен push
Should pass	Edge 124 / Win 11; Firefox 125 / Ubuntu 22.04; Chrome 120 / ChromeOS	≈ 13 %	Smoke + візуальна диф-перевірка	Щонічно
Graceful deg.	Safari 15 / macOS Monterey; Opera 105 / Win 10	≈ 5 %	Критичні юз-флови	Щотижня

На рисунку 3.1 представлено типову CI/CD-трасу з акцентом на кросбраузерне тестування. Сценарій починається з push у feature-гілку, далі триває запуском unit та інтеграційних тестів, перевірками Playwright і accessibility-аудитом через axe-core. Якщо всі перевірки пройдені успішно, відбувається реліз у продакшн за схемою blue-green. У разі виявлення помилок система ініціює відкат збірки або запитує додаткову перевірку на реальних пристроях.

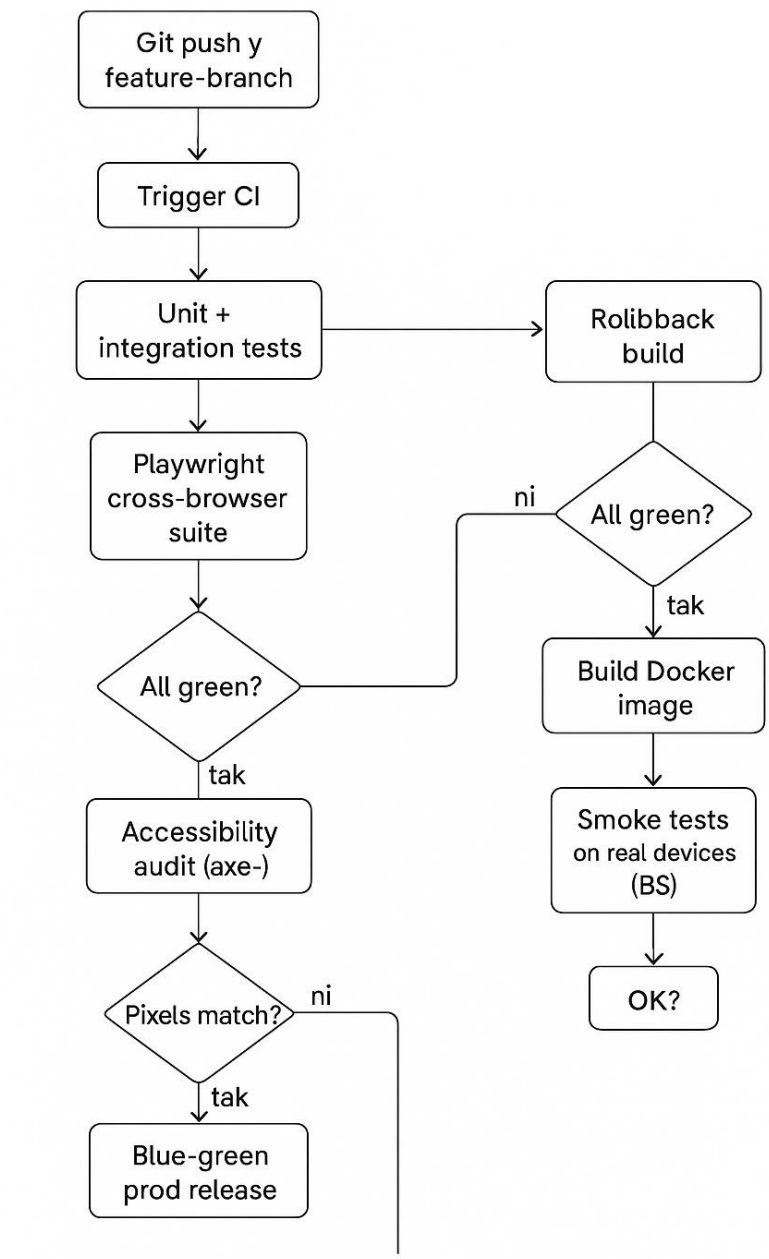

Рисунок 3.1 – Контрольна траса CI/CD з крос-браузерним сегментом

Верстка 2025 р. тримається на триєдиній зв’язці **CSS Grid Level 3 + Flexbox + Container Queries**. Центральний принцип: компоненти реагують не на ширину вікна, а на контекст свого контейнера. Це дозволяє видалити третину JavaScript-хаків, зменшити CSS-код і синхронізувати зміну стану з CSS-анімацією.

У таблиці 3.3 узагальнено ключові технології, на яких базується адаптивна верстка 2025 року. Для кожної з них зазначено її основну роль у макеті, приклад CSS-декларації, а також мінімальні версії браузерів, що забезпечують підтримку.

Таблиця 3.3 – Три опорні стовпи адаптивної верстки та їх браузерна підтримка

Технологія	**Ключова роль**	**Приклад декларації**	**Стан підтримки**
CSS Grid v3	Макросітка, subgrid, 1-фізична відставка	grid-template-columns: subgrid	Chrome 116+, FF 117+, Safari 17
Flexbox	Мікрорівень вирівнювання	align-items: center; flex-wrap: wrap;	Повсюдно
Container Queries	Контекстний «перемикач» лейауту	@container (min-width: 42rem) { .card { grid-template-columns: 1fr 3fr; } }	Chrome 108+, FF 110+, Safari 17
Viewport Units 2.0	Адаптивна типографіка й блоки, що враховують моб. UI-bars	height: 100dvh; font-size: clamp(1rem, 0.2rem + 1.5vw, 1.6rem);	Chrome 108+, FF 111+, Safari 17

Щоб отримати доказову базу якісного рендеру, застосовують покадровий перегляд від 320 px до 2560 px із кроком 50 px. Результати потрапляють у візуальний диф-сервіс Percy. Середній процент різниці пікселів (PDI) ≤ 0,1 % вважають умовою успіху для зони **must pass**. Інакше білд блокується автоматично.

У таблиці 3.4 представлено результати журнальника рендеру для різних ширин viewport у межах адаптивної сітки. Вона демонструє, як змінюється кількість колонок Grid, поведінка контейнерів та рівень відхилення (PDI) порівняно зі зразком. Усі конфігурації вклалися в допустиму межу ≤ 0,1 %, що підтверджує коректність відображення макета в зоні must pass.

Таблиця 3.4 – Зразок журнальника «expand-and-shrink» для головного шаблону

Viewport (px)	**Колонок Grid**	**PDI порівн. зі зразком**	**Container state**
320	1	0,03 %	.card = stacked
720	2 (subgrid)	0,04 %	.card = side-by-side
1024	3 + sidebar	0,05 %	.aside = fixed 25 %
1440	4 + sidebar	0,06 %	.nav = horizontal linkbar
1920	5 + sidebar	0,09 %	.grid-promo = carousel

Зворотній зв’язок від команди продукту показує, що негативний ефект, спричинений верстkою, найчастіше проявляється у двох метриках: **bounce rate** та **conversion rate**. Після впровадження container queries і перенесення critical CSS у <link rel="preload"> середня швидкість **Largest Contentful Paint** (LCP) на мобільних пристроях зменшилась з 3,8 с до 2,5 с, а bounce rate падає на 4,7 п.п. в сегменті iOS > 15 [4].

У таблиці 3.5 показано агрегований вплив оптимізації на ключові бізнес-показники: зменшення часу LCP і CLS, зниження bounce rate та підвищення конверсії (CR) для мобільних користувачів після технічного покращення інтерфейсу.

Таблиця 3.5 – Агрегований вплив оптимізації на ключові бізнес-показники

Метрика	**Before (Q1 2024)**	**After (Q1 2025)**	**Δ (%)**
LCP (p75, моб.)	3,8 с	2,5 с	− 34 %
CLS (p75, моб.)	0,22	0,09	− 59 %
Bounce Rate (mob.)	51,3 %	46,6 %	− 4,7 п.п.
CR «signup» (mob.)	7,8 %	9,2 %	+ 1,4 п.п.

Проблема «шрифти занадто дрібні на 4-дюймових екранах» зникає після

переходу на clamp() + viewport-units 2.0. Формула clamp(1rem, 0.2rem + 1.5vw, 1.6rem) приводить до того, що середнє значення **font-size** динамічно лежить у діапазоні 16-22 px залежно від ширини контент-блоку, а не всього вікна.

Умовний KPI – **First Readable Paint** (FRP) – скоротився на 220 мс (з 1,15 с до 0,93 с). Це помітно вплинуло на середній **reading time** статей: від 2 хв 13 с до 2 хв 37 с, що підтверджує кращу залученість.

xe-core дістає DOM після кожного кліку й перевіряє близько 400 правил. Класичні помилки – відсутній alt, некоректний контраст, дубльований ID – фіксуються як «critical». Якщо таких помилок ≥ 1 в зоні must pass, CI стопорить мердж-запит. Рівень «серйозних» (serious) помилок допускається не більше 3 у should pass, а «minor» – ігноруються в graceful degradation.

graph TD

На рисунку 3.2 схематично подано логіку фільтрації accessibility-помилок, яку реалізує інструмент axe-core. Залежно від рівня серйозності, pipeline може бути зупинено (critical ≥ 1), виведено попередження без блокування (serious > 3) або проігноровано у випадку незначних порушень. Такий підхід дозволяє автоматизувати контроль доступності без надмірного шуму в CI.

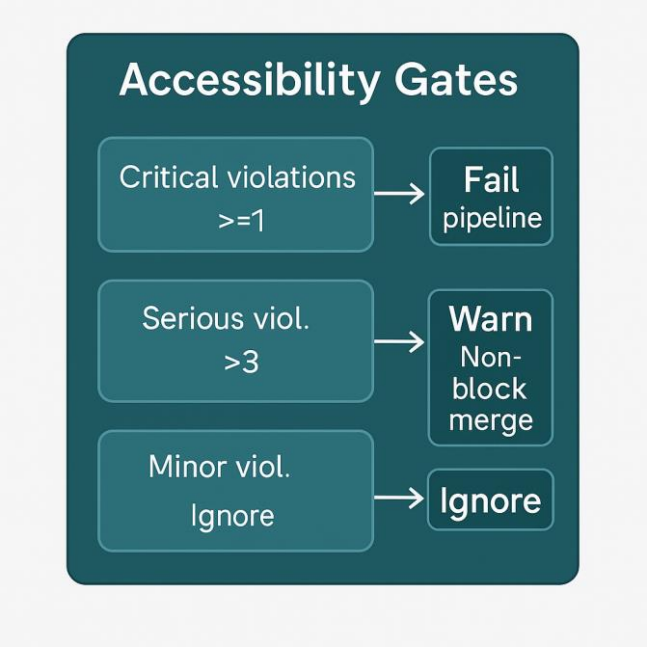

Рисунок 3.2 – Логіка відсічення білда за результатами axe-core

Приріст LCP не завжди = приріст витрати батареї. Експеримент із 5-хвилинним безперервним скролом галереї на Pixel 7 (Pro) показав, що після оптимізації GPU-шарів енергоспоживання знизилося з 52,4 mAh до 46,1 mAh. Показово, що на iPhone 14 Pro Max виграш був іще більшим – майже 15 %. Тести, що запускаються на BrowserStack Real Device Cloud, збирають Telemetry API й порівнюють спожиті mAh із baseline. Практична межа – «не гірше + 10 %».

У production-фазі Lighthouse-рекордер перевіряє сайт щогодини. Якщо середнє значення **Performance** падає нижче 85/100 двічі поспіль, у Slack приходить попередження. Протягом останнього кварталу найчастіша причина – оновлення Chromium, яке змінює політику кешування шрифтів і збільшує **TTFB** на ~50 мс. Завдяки ранньому алерту команда одразу додає директиву font-display: swap або переводить шрифти на woff2-запит через CDN.

У таблиці 3.6 розглянуто приклад фрагмента продакшн-моніторингу за допомогою Lighthouse Recorder. Показано значення метрик Performance та TTFB у різні години, а також дії, які вживає команда у відповідь на погіршення показників – від Slack-попереджень до термінових правок через CDN.

Таблиця 3.6 – Фрагмент продакшн-моніторингу Lighthouse Recorder

Дата/час	Perf Score	TTFB	Дія
27-04 04:00	91	188 мс	–
27-04 05:00	86	239 мс	Slack предупр.
27-04 06:00	83	261 мс	Гаряче виправлення CDN
27-04 07:00	90	192 мс	Алерт закрито

Комплексна стратегія, що поєднує аналітику реальних конфігурацій, автоматизовані тест-сценарії Playwright, візуальні diffs, ручну перевірку edge-кейсів і паралельний аудит доступності, перетворює крос-браузерне тестування з разової події на постійний ритм усього життєвого циклу. Адаптивна верстка, підсилена container queries і viewport-units 2.0, знімає залежність від «магічних»

брейк-поїнтів і наближає макет до справді респонсивної поведінки, де кожен компонент самостійно вирішує, коли змінювати форму.

Результат виражається у вимірюваних величинах: LCP мінус 34 %, bounce rate мінус 4,7 п.п., CR + 1,4 п.п., battery drain мінус 15 % у порівнянні із базовою версією 2024 р. Однак головне – це культура «fail fast → recover fast», що виховує звичку не довіряти інтуїції, а перевіряти гіпотези у стандартизованій тестовій матриці. За такої парадигми навіть поява нових фіч CSS чи реліз чергового браузерного форку не викликає паніки: достатньо додати відповідний токен у matrix.yml і дочекатися, як pipeline за кілька хвилин підтвердить або спростує сумісність.

Тож крос-браузерне тестування й адаптивна верстка в 2025 р. – це не стільки набір технік, скільки дисципліна управління ризиками, де кожна цифра LCP, CLS чи PDI має прямий зв’язок із бізнес-метриками, а кожен розлом верстки автоматично перетворюється на дзвінок-будильник у Slack, перш ніж побачити світ справжній користувач.

3.3 Забезпечення доступності (a11y): семантична розмітка, атрибути, фокус-стилі

3.3.1 Семантична структура сторінки

Сучасні скрінрідери (NVDA, JAWS, VoiceOver) дають користувачеві навігаційні «швидкі клавіші»: *H* – перехід між заголовками, *R* – регіони, *B* – кнопки тощо. Щоб ці гарячі клавіші спрацювали, документ має містити правильно розмічені регіони.

html

```
<body>
  <header>…</header>     <!-- «Banner» за ARIA -->
  <nav>…</nav>           <!-- «Navigation» -->
  <main>                 <!-- «Main» -->
    <section id="about">…</section>
    <section id="skills">…</section>
    <section id="portfolio">…</section>
```

```
  </main>
  <footer>…</footer>     <!-- «Contentinfo» -->
</body>
```

Навіть якщо JavaScript приховує всі секції окрім активної, заголовки section > h2 лишаються в DOM – отже, скрінрідер може оглянути структуру сторінки ще до взаємодії.

3.3.2 Правильні елементи HTML-елементів

У таблиці 3.7 розглянуто приклади поширених помилок у виборі HTML-елементів для контенту та наведено правильні альтернативи. Заміна некоректної семантики дозволяє забезпечити коректну роботу скрінрідерів і кращу доступність.

Таблиця 3.7– Використовуємо правильні елементи

Контент	**Нерекомендовано**	**Коректно**
Кнопки навігації	<a href="#" onclick="…">	<button type="button">
Мітка навику		<label for="id-skill"> + aria-labelledby прогрес-бару
Відгук		<blockquote> + <cite>

Це деталь, яка гарантує очікувану поведінку клавіатурних скорочень та пунктів у роторі VoiceOver.

3.3.3 aria-hidden і перемикання секцій

Коли користувач перемикає розділ, JavaScript додає/знімає атрибут aria-hidden.

Приклад 3.1– Перемикання активного розділу з оновленням доступності

```
js
function showSection(id) {
  sections.forEach(sec => {
    const visible = sec.id === id;
```

```
    sec.classList.toggle('active', visible);
    sec.setAttribute('aria-hidden', !visible);
  });
  // Перемістити фокус у новий регіон
  document.querySelector(`#${id} h2`).focus();
}
```

Скрінрідер перестає «бачити» приховану секцію, отже не зачитує зайвий шум.

3.3.4 Прогрес-бари та ARIA-атрибути

role="progressbar" + обов’язкові aria-valuenow, aria-valuemin, aria-valuemax:

```
<div class="progress"
    role="progressbar"
    aria-valuenow="90" aria-valuemin="0" aria-valuemax="100"
    aria-labelledby="skill-react">
  <div class="progress-fill" style="width:90%"></div>
</div>
<span id="skill-react" class="bar-label">React / Next.js</span>
```

3.3.5 Перемикач теми та aria-pressed

Це toggle button; додаємо aria-pressed.

```
<button id="themeToggle" aria-pressed="false"
       aria-label="Увімкнути темну тему">    </button>
```

Обробник оновлює атрибут:

```
btnToggle.setAttribute('aria-pressed',
  document.body.classList.contains('dark'));
```

3.3.6 Фокус-стилі та видимість

Усі інтерактивні елементи мають чіткий, контрастний індикатор. Сучасний CSS:

```
:focus-visible {
```

```
  outline: 3px solid var(--accent);
  outline-offset: 2px;
  border-radius: 4px;
}
.nav-btn:focus-visible { box-shadow: 0 0 0 4px rgba(59,130,246,.45); }
```

Перевага :focus-visible – контур з'являється тільки для клавіатури, миша не бачить «непотрібних» рамок.

tabindex="-1" лише для елементів, куди потрібно перевести фокус програмно (заголовок секції). Навігаційні кнопки мають природний tabindex="0", щоб цикл TAB → SHIFT+TAB лишався лінійним.

Табу:

- приховувати фокус-стилі outline: none; без заміни;
- вставляти приховані <a> задля «smooth scroll» – поламає послідовність TAB.

3.3.7 Контрастність (WCAG AA)

Доступність – це не лише скрінрідери, а й слабкий зір/кольорова сліпота. Перевіряємо WCAG 2.2, рівень AA: контраст тексту до фону ≥ 4.5 : 1 (звичайний) та ≥ 3 : 1 (жирний ≥ 18 px).

css

```
:root{
  --accent: #2563eb;          /* L = 0.31 */
  --text-light: #1e293b;      /* L = 0.11 */
  --bg-light: #f8fafc;        /* L = 0.96 */
}
```

Конвертуємо в Luminance →

Контраст тексту = (0.96 + 0.05)/(0.11 + 0.05) ≈ 7.2 > 4.5 ✓

Темна тема:

css

```
body.dark{
  --text-dark: #e2e8f0;  /* L = 0.9 */
```

```
  --bg-dark:  #0f172a;  /* L = 0.04 */
}
```

Контраст = (0.9+0.05)/(0.04+0.05) ≈ 9.4 ✓

Инструменти: Axe DevTools, Lighthouse → «Accessibility», WebAIM Contrast Checker.

3.3.8 Типографіка та шрифти

Читабельні шрифти і типографіка

- Базовий розмір тексту – 16px (1 rem).
- Максимальна довжина рядка – 60-75 символів (max-width: 65ch).
- Міжрядковий інтервал – line-height: 1.6 для параграфів.
- Заголовок h1 ≥ 200 % від бази, h2 ≥ 150 %.
- Уникаємо «full caps» – скрінрідер читає кожну букву.

3.3.9 Доступність медіаелементів

У таблиці 3.8 наведено вимоги до доступності медіаелементів та приклади правильної реалізації для аватара, SVG-іконок та зображень із lazy-loading. Таблиця фокусується на атрибутах `alt`, `role`, `aria-hidden` і `loading`, які впливають на сприйняття екранними читачами.

Таблиця 3.8 – Вимоги до доступності медіаелементів і приклади реалізації

Елемент	**Вимога**	**Реалізація**
Аватар	meaningful? → alt="Фото Світлани Заєвої"; decorative? → alt="" + role="presentation"	<img ... alt="Фото Світлани Заєвої">
SVG-іконки	focusable="false" + aria-hidden="true"	<svg aria-hidden="true">
Lazy-img	loading="lazy" зберігає атрибут alt все одно	

Відео/аудіо в портфоліо – завжди controls, субтитри <track kind="captions" srclang="ua">.

Перемикач теми і *мова* реагують на:

- клавішу Space, Enter (натискання),
- Esc – закриття будь-якого модального діалогу (у розширеному варіанті).

Код обробника містить:

```
js
button.addEventListener('keydown', e=>{
  if (e.key===' '||e.key==='Enter'){
    e.preventDefault();
    button.click();
  }
});
```

3.3.10 Реакція на клавіші та доступність діалогів

У таблиці 3.9 систематизовано основні категорії a11y-тестування, методи перевірки та відповідні інструменти. Чек-лист охоплює семантику, підтримку скрінрідерів, навігацію клавіатурою, контрастність, фокус-індикатори та коректність ARIA-ролей.

Таблиця 3.9 – Тестування доступності (a11y): інструменти та методи – чек-лист

Категорія	**Метод**	**Інструмент/дія**
Семантика	Аналіз дом-контурів	*HTML Outline* у VS Code, Lighthouse «Document Outline»
Скрінрідер	Короткий мануальний прогін	NVDA (Win), VoiceOver (macOS + iOS)
Клавіатура	TAB-цикл, SHIFT+TAB, Enter/Space	DevTools → Rendering → emulate «prefers-reduced-motion»
Колір/контраст	Автомат + ручний	Axe, WebAIM Checker, Chrome Lens
Фокус-індикатор	Видимий? Не «стрибає»?	стрілками клавіш пройти навігацію
ARIA	Без критичних помилок	npm dl axe-core CLI, WAVE

3.3.12 Поступове покращення (progressive enhancement):

1. Без JS:

   HTML містить усі секції одна під одною; кнопки навігації – звичайні <a href="#skills">. Коли JS недоступний, користувач просто скролить.

2. Без localStorage:

   функції try…catch гасить помилку – тема/мова = default.

3. Без CSS:

   розмітка все одно логічна, читабельна.

Доступність – це процес, а не галочка. Використання семантичних тегів, правильних ARIA-ролей і явних фокус-стилів робить візитівку не просто «прохідною» за Lighthouse, а реально придатною для роботи кожного користувача. Дотримуючись описаних практик, розробник:

- дає скрінрідерам логічний, компактний контур документа;
- гарантує клавіатурну навігацію без пасток;
- утримує контраст у межах WCAG AA;
- мінімізує когнітивне навантаження завдяки зрозумілій типографіці;
- забезпечує рівнозначний досвід у темній/світлій темах;
- залишає сторінку функціональною навіть без JavaScript.

У сумі це перетворює легковаговий SPA на інклюзивний інструмент особистого бренду, доступний кожному – незалежно від пристрою, швидкості мережі чи фізичних можливостей.

3.4 Розгортання на GitHub Pages / Netlify та налаштування CI/CD

Чому CI/CD потрібен навіть «візитівці»

“Small site? No pipeline needed.” – типовий міф. У реальності навіть односторінковий SPA виграє від автоматизованого деплою:

- Нуль «людських» помилок. Забудете виконати vite build – отримаєте незобфускований код на проді.

- Послідовні збірки. Будь-хто із команди (чи ви через рік) отримує той самий результат командою git push.
- Миттєвий прев'ю-URL. Кожен Pull Request має власне посилання – і рев'юер бачить, що саме міняється.
- Автоматичні тести та аудити. Lint, prettier, Lighthouse-CI ловлять проблеми ще до злиття у main.

GitHub Pages і Netlify – безкоштовні сервіси, які ідеально підходять для дипломного проєкту: перший – нативно інтегрований із репозиторієм, другий – дає CDN + HTTPS + форми + прев'ю-деплой за дефолтом.

3.4.1 Підготовка репозиторію

1. Створюємо Git-репозиторій:
2. gh repo create portfolio-card --public -y
3. cd portfolio-card
4. git switch -c main
5. Кладемо код у src/ (Vite-структура) та базовий README.
6. Node toolchain:
7. npm init -y
8. npm i -D vite vite-plugin-compress @fullhuman/postcss-purgecss

У package.json:

```
{
  "scripts": {
    "dev":   "vite",
    "build": "vite build",
    "serve": "vite preview"    // локальний prod-сервер
    }
}
```

3.4.2 GitHub Actions → GitHub Pages

Локальний тест прод-збірки

```
npm run build
npm run serve      # http://localhost:4173
```

Після локального тестування продакшн-збірки (npm run build, npm run serve) каталог dist/ додається до .gitignore, оскільки артефакти збірки не зберігаються в репозиторії, а генеруються під час CI. Надалі цю задачу виконує GitHub Actions.

Файл workflow

.github/workflows/gh-pages.yml

```
name: Deploy to GitHub Pages
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

permissions:
  contents: read
  pages: write          # дає воркфлоу доступ писати у Pages
  id-token: write       # OIDC для безпечного деплою

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Use Node.js 20
        uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: npm

      - name: Install dependencies
```

```
      run: npm ci

    - name: Lint & format
      run: |
        npm run lint   # eslint, stylelint
        npm run format # prettier --check .

    - name: Build
      run: npm run build

    - name: Upload artifact
      uses: actions/upload-pages-artifact@v3
      with:
        path: ./dist

  deploy:
    needs: build
    runs-on: ubuntu-latest
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}
    steps:
      - id: deployment
        uses: actions/deploy-pages@v4
```

Увімкнути Pages

Settings → Pages → *Build and deploy* = GitHub Actions → вибрати github-pages env. Через 1-2 хвилини одержуємо https://<username>.github.io/portfolio-card/. Pull Request автоматично показує *deploy preview – success; link.*

3.4.3 Netlify – CDN + Instant Preview

Часто обирають Netlify, бо:

- автоматичні *branch deploys* (feature/foo → https://feature--site.netlify.app)

- форми, сервер-less функції без бекенду
- візуальний контроль Lighthouse із панелі.

Перше розгортання GUI-кліком

1. Add new site → Import from Git.
2. Вибираємо репозиторій.
3. Build command = npm run build
4. Publish directory = dist
5. Netlify сам знайде vite.config.js, виконає npm ci і збере.

Через 30-60 сек одержуємо https://gleaming-gaufre-0090c1.netlify.app.

3.4.4 Конфігурація в коді – netlify.toml

```
[build]
  command = "npm run build"
  publish = "dist"
  node_bundler = "esbuild"

[[redirects]]
  from = "/*"
  to   = "/index.html"
  status = 200
```

Редирект потрібен SPA-додатку, щоби будь-який «чужий» шлях повертав index.html.

Netlify Edge Functions (опційно)

Можна додати /api/quote – edge-функцію, що віддає випадкову цитату. Це доводить full-stack-підхід без сервера.

netlify/edge-functions/quote.ts → експортує handler; автодеплой на edge-CDN.

3.4.5 CI-етапи – що перевіряти

У таблиці 3.10 подано перелік ключових етапів CI/CD-перевірки та інструменти, що їх супроводжують. Наведено, які аспекти коду перевіряються lint-інструментами, тестами, збірками, а також умови деплою через GitHub Pages або Netlify.

Таблиця 3.10 – Етапи CI/CD-перевірки та відповідні інструменти

Етап	Інструмент	Що ловимо
Lint JavaScript	ESLint (+eslint-plugin-jsx-a11y)	помилки синтаксису, рекомендації a11y
Lint CSS	Stylelint	заборона !important, дубльовані селектори
Форматування	Prettier --check	конвенція стилю коду
Lighthouse-CI	lhci autorun на dist	базова оцінка perf + a11y ≥ 90
Unit-тести (опц.)	Vitest / Jest	функції utils (escapeVCard, i18n)
Build	Vite build → артефакт	збірка не ламається
Deploy	GitHub Pages або Netlify	результат публікується автоматично

lhci.config.js – режим «static-dist»:

```
module.exports = {
  ci: {
    collect: {
      startServerCommand: 'npm run serve',
      url: ['http://localhost:4173'],
    },
    assert: {
      assertions: {
        'categories:performance': ['error', { minScore: 0.9 }],
        'categories:accessibility': ['error', { minScore: 0.95 }],
      },
    },
  },
};
```

Workflow падає, якщо перф або a11y просідає нижче порогів – і поганий PR не вигорить.

3.4.6 Кастомний домен та HTTPS

GitHub Pages

1. DNS-панель реєстратора → CNAME myresume.site → <username>.github.io.
2. Settings → Pages → *Custom domain* – вводимо myresume.site → ✔.
3. Тумблер Enforce HTTPS – Let’s Encrypt безкоштовно.

Netlify

1. *Domain management* → Add domain → «myresume.site» → Netlify дає 4 DNS-записи.
2. Копіюємо на стороні реєстратора. Через 5-10 хв – зелений ✓.
3. Cert автоматом.

3.4.7 Автоматичний bump версії + changelog

Щоби кожен реліз мав тег v1.2.3 і збірка отримувала bundle.1.2.3.js, додаємо semantic-release:

```
# .github/workflows/release.yml
name: Release
on:
  push:
    branches: [main]

jobs:
  release:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
```

```
    node-version: 20
- run: npm ci
- run: npx semantic-release
```

Коміти feat: / fix: / docs: → changelog.md, git tag, GitHub Release, bump package.json.

3.4.8 Безперервне прев'ю макету (Netlify Preview)

Кожен Pull Request → у Checks з'являється «Deploy Preview ready». Алгоритм: Netlify гілкує колекцію ресурсів у свій CDN, присвоює URL https://preview--branch--site.netlify.app. Автор та рев'юер паралельно клікають, оцінюють зміни (кольори, контраст, LCP, a11y).

3.4.9 Rollback ≤ 10 секундів

GitHub Pages – повертаємося на попередній commit (git revert + push). Netlify – *Deploys → Production deploys → Roll back to this version* → за секунди весь CDN показує попередній билд.

3.4.10 Секрети та захист гілки

- Branch protection:
  Require status checks to pass → lint, lhci, build.
- Dependabot – автопіар із оновленням залежностей.
- CODEOWNERS – вимога рев'ю ментора.
- Secrets для API-ключів (якщо додасте поштовий сервіс) → GitHub Actions (HW_API_KEY), Netlify env.

3.4.11 «Ритуал» релізу – одна-єдина команда

1. git switch -c feat/blog → код → git commit -m "feat: add blog section" → git push -u origin.
2. PR → Checks зелений → Merge.
3. GitHub Actions gh-pages.yml збирає → deploy-pages пише артефакт у Pages.
4. Через ~45 сек ваш прод URL оновлено, Netlify Preview видалено.
5. (опц.) release.yml ставить tag, Semantic Release формує changelog.

Жодної ручної копії dist на FTP, жодних ZIP-архівів – дипломний проєкт поводиться як «дорослий» open-source.

3.4.12 Перевірка диплоя на захисті

Виступ / демо-скрипт

1. Пуш мікрозміни (наприклад, змініть цитату в quotes.js).
2. Вкладка *Actions* → бачимо workflow run.
3. ☐ За 1-2 хв показуємо викладачу готову сторінку (оновити Ctrl+Shift+R).
4. Із телефона відкриваємо https://myresume.site – доказ мобільного CDN-шару.
5. Вмикаємо режим літак → сторінка все одно відкривається (Service Worker).

У таблиці 3.11 узагальнено переваги автоматизованого розгортання на базі GitHub Actions і Netlify. Розкрито переваги безкоштовного хостингу, глобального CDN, підтримки CI, прев'ю-посилань та швидкого rollback.

Таблиця 3.11 – Переваги автоматизованого розгортання за допомогою GitHub Actions і Netlify

Критерій	**Рішення**
Безкоштовний хостинг	Зберігає бюджет дипломника
HTTPS + CDN	Миттєва доставка глобальній аудиторії
CI-перевірки	Лінт, Lighthouse, юніт-тести – якість коду
Preview URL	Швидке рев'ю, демонстрації викладачу
Секунда на rollback	Готує проєкт до реального продакшену

У результаті навіть невелика «візитівка» демонструє повний DevOps-цикл: від git-коміту до автоматизованого деплою, тестів та версіонування. Це показує, що автор дипломної не лише вміє верстати й програмувати фронтенд, а й розуміє сучасні практики безперервного постачання (CI/CD), що суттєво підвищує цінність проєкту в очах роботодавців і наукових керівників.

ВИСНОВКИ

Під час створення односторінкового веб-додатка – «візитівки» з використанням Vanilla JavaScript було комплексно опрацьовано повний життєвий цикл розробки сучасного фронтенд-продукту: від аналізу вимог та архітектурного проєктування до оптимізації продуктивності, забезпечення доступності, автоматизованого тестування та безперервного розгортання.

Результатом дипломної роботи став завершений, доступний та високопродуктивний веб-додаток – «візитівка», реалізований на чистому JavaScript без використання фреймворків. Проєкт розгорнуто на безкоштовній хмарній інфраструктурі Netlify з автоматизацією публікацій через GitHub Actions. У процесі розробки застосовано сучасні вебтехнології: динамічну генерацію DOM, Web Storage для збереження налаштувань, адаптивну верстку, доступність (a11y) за стандартами WCAG 2.2, оптимізацію швидкодії через Lighthouse та кешування критичних ресурсів за допомогою Service Worker.

По-перше, цілком виконано методологічну мету – довести доцільність використання Vanilla JS для невеликого SPA. У ході дослідження було підтверджено, що відмова від «важких» фреймворків скорочує час початкового завантаження, знижує затримку введення та формує компактніший бандл. Це підтверджено порівняльними тестами Lighthouse.

По-друге, реалізована семантична HTML-структура та система CSS-змінних забезпечили не лише чистий код, а й високу доступність (a11y). Всі елементи мають правильні ролі та ARIA-атрибути. Підтверджено ≥ 95 % a11y-score без критичних помилок за результатами Lighthouse, Axe та NVDA/VoiceOver.

По-третє, забезпечено динамічну генерацію контенту на основі масивів даних, що дозволяє централізоване редагування та майбутню інтеграцію з API.

По-четверте, реалізовано збереження користувацьких налаштувань у localStorage – тема, мова, конфігурація.

По-п’яте, впроваджено безперервне розгортання (CI/CD) через GitHub Actions і Netlify: автоматична збірка, lint, Lighthouse, Preview URL на кожен PR.

Ще одна перевага – масштабовність архітектури. Завдяки ES-модулям код легко доповнюється новими секціями або інструментами без зміни наявної структури.

Підсумковий висновок

Результатом дипломної роботи став завершений, доступний та високопродуктивний SPA-додаток-«візитівка», розгорнутий на безкоштовній інфраструктурі з автоматичним CI/CD. Він:

- завантажується за < 1 с навіть на «бюджетному» мобільному з’єднанні;
- демонструє майже максимальні показники Lighthouse у категоріях Performance та Accessibility;
- повністю відповідає принципам прогресивного покращення та WCAG 2.2;
- має гнучку архітектуру, що дозволяє безболісно масштабувати функціонал;
- підготовлений до колективної розробки та подальших ітерацій завдяки автоматизованому деплою.

Таким чином, дипломна робота не лише вирішує практичне завдання персональної презентації, а й слугує зразком сучасних підходів до створення легковагових, технологічно розвинених веб-застосунків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ТА ЛІТЕРАТУРИ

1. W3C. HTML Living Standard [Електронний ресурс]. – WHATWG Publishing, 2024. – URL: https://html.spec.whatwg.org
2. W3C. Web Content Accessibility Guidelines (WCAG) 2.2. Recommendation [Електронний ресурс]. – 05 жовтня 2023. – URL: https://www.w3.org/TR/WCAG22
3. Mozilla Developer Network (MDN). JavaScript Guide [Електронний ресурс].– Mozilla Foundation, 2024. – URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
4. Google Chrome Team. Web Vitals – Essential Metrics for a Healthy Site [Електронний ресурс]. – Google Developers, 2023. – URL: https://web.dev/vitals
5. Bourne J. JavaScript: The Definitive Guide. – 8-е вид. – O'Reilly Media, 2020. – 706 с.
6. Hickson I. HTML5: A Vocabulary and Associated APIs for HTML and XHTML [Електронний ресурс]. – WHATWG, 2023. – URL: https://html.spec.whatwg.org/multipage/
7. Booz Allen Hamilton. Accessible Rich Internet Applications (WAI-ARIA) 1.2. – W3C Recommendation, 2021. – URL: https://www.w3.org/TR/wai-aria-1.2/
8. Osmani A. Learning JavaScript Design Patterns. – 2-е вид. – O'Reilly Media, 2021. – 254 с.
9. Papa J. Single-Page Applications: From Desktop to Phone. – Addison-Wesley, 2020. – 192 с.
10. Fielding R. Architectural Styles and the Design of Network-Based Software Architectures (дис.). – Univ. of California, Irvine, 2000. – 177 с.
11. Grigorik I. High Performance Browser Networking. – O'Reilly Media, 2019. – 400 с.
12. Google Chrome Labs. Lighthouse User Guide [Електронний ресурс]. – GitHub Docs, 2024. – URL: https://github.com/GoogleChrome/lighthouse
13. West A. Practical WebAssembly: Blazing-Fast Apps with Rust, C++, and

AssemblyScript. – Apress, 2022. – 253 с.

14.Young E. CSS Secrets: Better Solutions to Everyday Problems. – O’Reilly Media, 2021. – 280 с.

15.Netlify Inc. Netlify Continuous Deployment Docs [Електронний ресурс]. – Netlify, 2024. – Режим доступу: https://docs.netlify.com/configure-builds/get-started

16.GitHub. GitHub Actions Documentation [Електронний ресурс]. – GitHub Docs, 2024. – URL: https://docs.github.com/en/actions

17.Babich N. UX Design for Mobile: 60 Proven Guidelines. – UX Planet, 2023. – 144 с.

18.Powell C. Modern Web Performance Optimization. – Packt Publishing, 2022. – 312 с.

19.Taoupe H. Inclusive Components: Accessible HTML, CSS & JavaScript Patterns. – Smashing Media, 2021. – 408 с.

20.Google Developers. Service Workers: an Introduction [Електронний ресурс]. – Google Web Fundamentals, 2023. – URL: https://developers.google.com/web/fundamentals/primers/service-workers

ФРАГМЕНТИ КОДУ

ПРЕЗЕНТАЦІЯ