НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

УДК 004.02

«ПОГОДЖЕНО»	**«ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ»**
Декан факультету інформаційних технологій	Завідувач кафедри комп'ютерних наук
Глазунова О.Г., д.п.н., професор	Голуб Б.Л., к.т.н., доцент
_____	_____
____ _____ **2023 р.**	____ _____ **2023 р.**

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему Методи розроблення бібліотеки компонентів інтерфейсу користувача для веб-додатків

Спеціальність Інженерія програмного забезпечення

(код і назва)

Освітня програма Програмне забезпечення інформаційних систем

(назва)

Орієнтація освітньої програми _____
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

_____ _____ _____
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник магістерської кваліфікаційної роботи

кандидат ф.-м. наук, доцент _____ Кириченко В.В.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Погребняк Денис Андрійович
(підпис) (ПІБ студента)

КИЇВ-2023

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) _____

ЗАТВЕРДЖУЮ
Завідувач кафедри_____
_____
_____ _____ _____
(науковий ступінь, вчене звання) (підпис) (ПІБ)
"____" _____20_____року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Погребняк Денис Андрійович
(прізвище, ім'я, по батькові)

Спеціальність 121 Інженерія програмного забезпечення
(код і назва)

Освітня програма Інформаційні управляючі системи та технології
(назва)

Орієнтація освітньої програми_____
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи Методи розроблення бібліотеки компонентів інтерфейсу користувача для веб-додатків

затверджена наказом ректора НУБіП України від "____"_____20___р. №_____

Термін подання завершеної роботи на кафедру "5" листопада 2023
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи _____

Перелік питань, що підлягають дослідженню:
1. _____
2. _____
3. _____

Перелік графічного матеріалу (за потреби) _____

Дата видачі завдання "_____"_____20___ р.

Керівник магістерської кваліфікаційної роботи _____ Кириченко В.В.
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ Погребняк Д.А
(підпис) (прізвище та ініціали студента)

ЗМІСТ

ВСТУП

У сучасному світі веб-застосунки стали вагомим атрибутом нашого повсякденного життя, адже вони використовуються в багатьох сфер, від бізнесу до освіти та розваг. Розмаїття цих застосунків суттєво збільшило попит на швидку та ефективну розробку їхніх інтерфейсів користувача. Якість і зручність веб-інтерфейсу сильно впливає на користувацький досвід. Проте розробка та підтримка інтерфейсів користувача можуть бути трудомісткими та часозатратними завданнями.

Для полегшення цих проблем створили бібліотеки компонентів, котрі стали гарним та зручним інструментом розробників для створення інтерфейсів користувачів. Вони стандартизують та прискорюють процес створення інтерфейсів, дозволяючи використовувати готові елементи замість написання коду з нуля. Це спрощує розробку, робить її більш ефективною та зберігає час та ресурси.

Проте, розробка власних бібліотек компонентів може бути складною через різноманіття можливостей їх реалізації та потреб конкретних застосунків. Дана магістерська робота спрямована на дослідження відомих методів розробки бібліотек компонентів для веб-застосунків та розробку загального методу для їх створення. Цей метод повинен спростити роботу розробників, забезпечуючи легкий та зрозумілий підхід до створення нових бібліотек.

Метою моєї роботи є розроблення нового методу створення бібліотек веб-компонентів на основі аналізу існуючих методів та їх порівняння. Для досягнення цієї мети ставляться наступні завдання:

- Проаналізувати сучасні бібліотеки компонентів та інструменти для їх розробки.
- Визначити основні принципи проектування та структури таких бібліотек.
- Розробити методику для створення бібліотек компонентів.
- Створити програмний застосунок для реалізації розробленої методики.

Для досягнення цих завдань в роботі використовуються наступні методи дослідження:

- Аналіз існуючих веб-застосунків та бібліотек компонентів для визначення їх особливостей та недоліків.
- Аналіз програмних інструментів та технологій для створення бібліотек компонентів.
- Моделювання загальної архітектури бібліотеки компонентів.
- Моделювання програмного застосунку для реалізації розробленого методу створення бібліотек компонентів.

Результати цієї роботи можуть бути корисні як для індивідуальних розробників, так і для комерційних компаній, які прагнуть покращити процес розробки веб-застосунків. Основні висновки дослідження можуть служити основою для подальших досліджень у галузі веб-розробки та розробки бібліотек компонентів.

У наступних розділах роботи будуть розглянуті і проаналізовані сучасні бібліотеки компонентів, розроблені методики створення нових бібліотек та реалізовані програмні рішення на основі розробленого методу.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Історія розвитку користувацьких інтерфейсів у веб-розробці.

Історію розвитку користувацьких інтерфейсів у веб-розробці можна поділити на декілька етапів [1]:

- Створення HTML: Спочатку веб-сторінки мали прості інтерфейси, складені переважно з тексту та зображень, і надавали обмежені можливості взаємодії.
- Поява JavaScript: Поява JavaScript в 90-х роках дозволила додавати динаміку до веб-сторінок, що покращило інтерактивність інтерфейсу. Але розробка складних компонентів була новою та доволі часоємкою.
- Зародження бібліотек компонентів: З появою jQuery та інших бібліотек JavaScript, розробники отримали можливість створювати та використовувати готові компоненти, такі як плагіни для календарів, галереї тощо.
- Розповсюдження односторінкових застосунків: З популярністю односторінкових застосунків (SPA) з'явилися бібліотеки та фреймворки, такі як Angular, React та Vue.js, які дозволили створювати складніші інтерфейси на веб-сторінках.
- Зростання популярності компонентів: Сучасні бібліотеки компонентів, такі як Material-UI для React або Vuetify для Vue.js, надають широкий вибір готових компонентів та спрощують створення стильних та функціональних інтерфейсів користувача.

Всі ці етапи відображають поступовий розвиток веб-інтерфейсів та значення бібліотек компонентів у покращенні користувацького досвіду та ефективності веб-розробки.

1.2 З чого складаються бібліотеки користувацьких інтерфейсів

Сучасні бібліотеки користувацьких інтерфейсів базуються на певних веб-фреймворках [1]. Веб-фреймворк - це набір заздалегідь написаних, готових до використання компонентів, бібліотек, правил та інструментів, які спрощують процес розробки веб-застосунків. Це програмне забезпечення, яке надає загальну структуру для побудови веб-застосунків і полегшує взаємодію з веб-серверами та базами даних.

Основні компоненти веб-фреймворка можуть включати:

- Маршрутизацію (Routing): Дозволяє визначати, які частини додатку пов'язані з конкретними URL-адресами.
- Шаблонізацію (Template Engine): Забезпечує можливість створення шаблонів для відображення веб-сторінок.
- Базу даних (Database Abstraction): Надає спрощений доступ до баз даних, дозволяючи взаємодіяти з ними без прямого написання SQL-запитів.
- Аутентифікацію та Авторизацію (Authentication and Authorization): Забезпечує механізми перевірки ідентифікації користувачів та надання їм доступу до різних частин додатку.
- Управління Сесіями (Session Management): Дозволяє зберігати стан користувача між різними запитами до веб-сервера.
- Обробку Помилок та Логування (Error Handling and Logging): Надає можливість ефективно вирішувати помилки та здійснювати запис подій для моніторингу додатку.
- Тестування (Testing): Забезпечує інструменти для автоматизації тестування функцій додатку.

В залежності від поставлених задач розробники вибирають відповідний тип. Розглянемо деякі типи фреймворків у веб-програмуванні. Загалом за призначенням [2] їх можна поділити на:

- Фронтенд (англ. “frontend”) фреймворки - це набір інструментів, бібліотек і структур, які допомагають розробникам створювати користувацький інтерфейс веб-застосунків. Вони спрощують роботу з HTML, CSS і JavaScript, роблять розробку інтерактивних та естетичних веб-сайтів більш швидкою та ефективною. Прикладом таких фреймовків є: React.js, Vue.Js, Angular.
- Бекенд (англ. “backend”) фреймворки - це набір інструментів для розробки серверної частини веб-застосунків. Вони забезпечують зв'язок із базами даних, обробку запитів, безпеку, аутентифікацію та інші серверні операції. Приклад бекенд фреймворку: Django, Express, Spring.
- Фулл стек (англ. “fullstack”) фреймворки - це фреймворки, які охоплюють як фронтенд, так і бекенд розробку, забезпечуючи повний спектр інструментів для створення веб-застосунків від початку до кінця. До такого типу фреймворків можна віднести Ruby On Rails, Laravel, Flask, ASP.NET.

На рівні управління застосунком веб-фреймворки можна поділити на два класи: веб-фреймворків на основі дій (англ. “Action-based web frameworks”) та компонентно-орієнтовані веб-фреймворки (англ. “Component-based web frameworks”) [2].

У випадку веб-фреймворків на основі дій послідовність HTTP-запиту-відповіді чітко відображена, в компонентно-орієнтованих веб-фреймворках - вона є більш абстрактною.

Коли йдеться про фреймворки на основі дій, контролер служить центральним компонентом, який приймає запити клієнтів, перевіряє їх і запускає

відповідну дію. Для кожної можливої дії розробник застосунку повинен спочатку створити програмний об'єкт, який містить відповідну логіку. Це, як правило, можна отримати з абстрактних класів. Якщо дія виконана, контролер оновлює модель даних і переадресовує результат до представлення, яке, своєю чергою, створює відповідь та надсилає її клієнту.

Веб-фреймворки на основі дій схильні до шаблону MVC (“Model-View-Controller”) і також відомі як запит-відповідь, через жорстку реалізацію цієї послідовності запиту-відповіді. Типовим представником такого класу фреймворків є: Ruby On Rails, Django, ASP.NET.

На відміну від підходу, керованого діями - компонентно-орієнтовані веб-фреймворки абстрагують HTTP-запит-відповідь, розглядаючи користувацький інтерфейс веб-застосунку як колекцію компонентів. Для кожного з цих компонентів, які пов'язані з програмними об'єктами на серверному боці, визначаються конкретні реакції під час розробки веб-застосунку. Ці реакції відбуваються при подіях, спричинених взаємодією користувача з компонентом. Тому їх називають подійно-керованими веб-фреймворками. До цього типу фреймворків відносяться: React, Vue.js, Angular.

Основна ідея за компонентно-орієнтованим підходом полягає в групуванні пов'язаних дій разом. Отже, один програмний об'єкт може бути відповідальний за кілька дій. Компонентно-орієнтовані веб-фреймворки зазвичай надають великий вибір повторно використовуваних компонентів, які приховують деталі базового шаблону запит-відповідь від розробника застосунку.

Відповідно до поставлених задач розробники вибирають конкретний вид фреймворків, проте потрібно зазначити, що при використанні веб-фреймворків на основі дій, можна використовувати компонентно-орієнтовані фреймворки для фронт енд частини застосунку.

Бібліотеки компонентів веб-інтерфейсу - це набір готових до використання елементів інтерфейсу, які розробники можуть використовувати при створенні веб-застосунків та веб-сайтів. Ці бібліотеки містять різноманітні компоненти, такі як кнопки, текстові поля, меню, панелі, календарі, графіки та багато інших, які можна легко інтегрувати в веб-сторінки або застосунки.

Самі компоненти в веб-інтерфейсі являють собою сукупність html-розмітки, що формує зовнішню структуру компонента; css-стилів, що відповідають за зовнішній вигляд; та javascript-коду, який реалізує інтерактивність компонента. Також кожен компонент має власні властивості для передачі даних (“props”) та події (“events”), для реагування на дії користувача.

1.3 Роль і значення бібліотек компонентів у веб-розробці

Часто під різні проєкт потрібно формувати новий унікальний дизайн інтерфейсу й відповідно, або редагувати готові бібліотеки, або створювати нову власну бібліотеку, або на основі вже готової бібліотеки створити новий набір (бібліотеку) яка міститиме один спільний дизайн. Для компаній котрі мають свій власну дизайн-систему особливо важливо забезпечити узгодженість дизайну інтерфейсу поміж своїх програмних продуктів [3]. Якщо ж такої узгодженості не буде, то дизайн програмного продукту буде мати хаотичний не зрозумілий для користувача стиль (див. рис. 1.1).

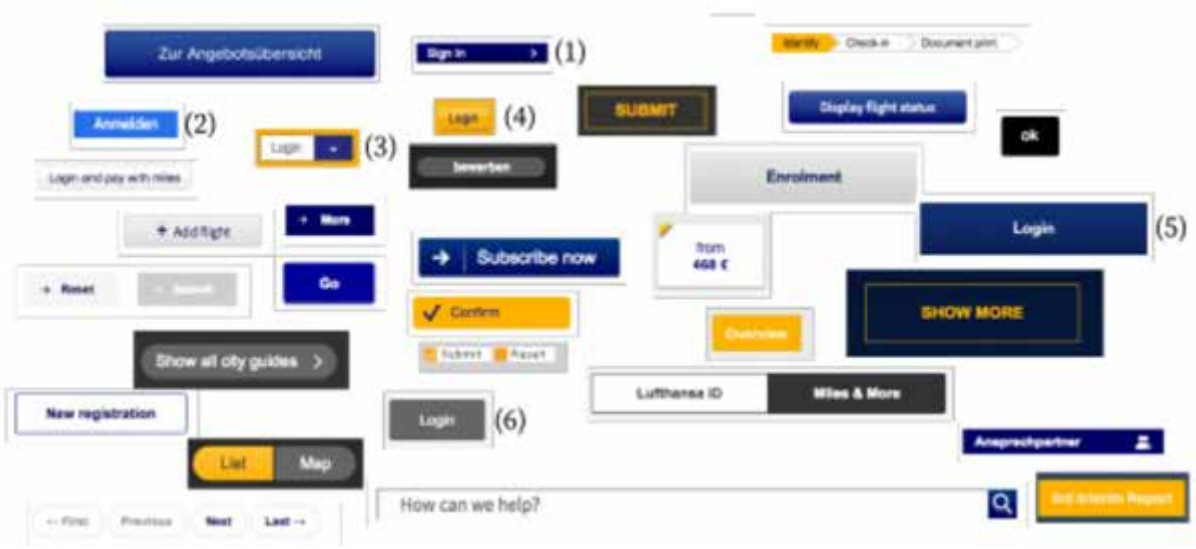

Рисунок 1.1 - Колекція різних кнопок із веб-застосунків компанії Lufthansa AG

Загалом бібліотеки компонентів використовують через ряд переваг котрі вони надають розробникам, зокрема серед переваг можна виділити наступні:

1. **Прискорення розробки:** Бібліотеки компонентів містять готові до використання елементи інтерфейсу, такі як кнопки, меню, форми тощо. Розробники можуть використовувати ці компоненти замість створення своїх компонентів з нуля. Треба зазначити, що створення та налагодження деяких таких компонентів може займати доволі багато часу, тому використання готових компонентів може значно пришвидшити процес розробки і зосередитися конкретно на розробці функціоналу самого застосунку.
2. **Стандартизація інтерфейсу:** Бібліотеки компонентів дозволяють створювати стандартизований та консистентний дизайн інтерфейсу користувача. Це важливо для підтримки бренду та зручності користувачів. Також бібліотеки компонентів зазвичай надають можливість

налаштовувати та кастомізувати компоненти відповідно до потреб проекту. Розробники можуть змінювати стиль, розмір, колір і інші параметри, щоб адаптувати компоненти під конкретні вимоги.

3. **Модульність:**Модульність компонентів веб-інтерфейсу - це ключовий аспект їхньої конструкції, який дозволяє легко розширювати, налаштовувати та використовувати їх в різних частинах веб-застосунків або веб-сайтів. Це досягається завдяки розділенню компонентів на менші підкомпоненти, які можуть бути незалежними та самодостатніми. Підкомпоненти можуть приймати параметри та властивості для налаштування їхньої поведінки, інкапсулювати свій внутрішній стан та логіку, а також бути легко заміненими чи розширюваними без впливу на інші частини системи. Модульність полегшує тестування компонентів та сприяє забезпеченню високої якості та надійності інтерфейсу користувача.
4. **Зменшення помилок:** Готові компоненти проходять тестування та валідацію, тому їх використання може допомогти уникнути типових помилок, що звичайно виникають при розробці інтерфейсу. Також готові компоненти зазвичай підтримуються командою розробників бібліотеки. Це означає, що вони оновлюються, наявні помилки виправляються та додаються нові можливості. Розробники можуть легко оновити компоненти у своєму проекті, забезпечуючи актуальність інтерфейсу користувача.

Але також потрібно зауважити, що використання бібліотек компонентів мають й певні недоліки, зокрема:

1. **Розмір:** Великі бібліотеки компонентів можуть бути великими за розміром або містити надлишковий функціонал - це призводить до додаткового завантаження на стороні клієнта та збільшення часу завантаження веб-

сторінок. Це особливо критично для мобільних пристроїв і повільних інтернет з'єднань.

2. **Залежність від сторонніх джерел:** Використання зовнішніх бібліотек може внести залежність від третіх сторін та зробити застосунок вразливим до змін або відмови в обслуговуванні цих сторін. Якщо стороння бібліотека припиняє підтримку або виникають конфлікти версій, це може призвести до проблем у застосунку.
3. **Несумісність:** Іноді бібліотеки компонентів можуть бути несумісними з іншими бібліотеками чи фреймворками, які використовуються у проекті. Це може призвести до конфліктів версій та виникненню нових помилок в проекті.
4. **Збільшення складності коду:** Використання багатьох компонентів може призвести до збільшення складності коду, особливо якщо потрібно налаштовувати кожен компонент окремо. Це може зробити код менш читабельним та важким у підтримці.
5. **Обмежена налаштованість:** Деякі бібліотеки можуть бути обмежені у плані налаштованості. Якщо вам потрібно здійснити значні зміни в зовнішньому вигляді чи функціональності компонента, це може бути складним або неможливим завданням.

Правильний вибір та використання бібліотеки компонентів може частково або повністю нівелювати ці недоліки. Тому особливо важливо обміркувати, які компоненти та бібліотеки варто використовувати у конкретному проєкт з урахуванням його особливостей та вимог.

1.4 Проблеми побудови та оптимізації бібліотек

Як було описано вище, існує ряд проблем пов’язаний з тим, що бібліотека може використовуватися для побудови найрізноманітніших

інтерфейсів веб-застосунків, а отже й повинна бути оптимізована під різні платформи.

Розглянемо головні проблеми оптимізації, а саме оптимізації під: конкретні способи генерації сторінок та технології treeshaking.

Існує декілька способів генерації веб-сторінок, а також може відрізнятися підтримка цих способів зі сторони бібліотек інтерфейсів.

Основними способами генерації сторінок є (в додатку 1 продемонстровані схеми роботи цих способів)[4]:

- Client-Side Rendering (CSR) - рендеринг сторінки на стороні клієнта;
- Server-Side Rendering (SSR) - рендеринг сторінки на стороні сервера;
- Static Site Generation (SSG) - генерація статичних сайтів, інформація на яких рідко змінюється;
- Incremental Static Regeneration (ISR) - це об’єднання методів SSR та SSG.

Проблемою є те, що у випадку SSR та SSG (й відповідно ISR) компоненти спочатку рендериться на сервері, де немає API браузера, тому під час написання коду оптимізованого під SSR потрібно бути уважним, щоб компоненти використовували API браузерів тільки коли код виконується на клієнті.

Tree Shaking - це технологія оптимізації коду у контексті веб-розробки, яка дозволяє вилучати частини коду, що не використовуються (наприклад, функції, стилі або змінні) під час компіляції, щоб зменшити розмір кінцевого файлу JavaScript [5]. Це особливо корисно у великих проєктах, де багато бібліотек або модулів можуть включати багато функціонала, який може не використовуватися повністю в кінцевому додатку.

Загалом процес Tree Shaking зазвичай наступним чином:

1. Аналіз коду: компілятор або інші інструменти аналізують вихідний код JavaScript та визначають, які функції, класи, методи, змінні, стилі та інші ресурси використовуються в коді.
2. Вилучення коду, що не використовується: код, який не використовується, вилучається з кінцевої збірки (англ. “bundle”) JavaScript. Це означає, що не використовувані функції або класи не потрапляють до кінцевого коду, зменшуючи таким чином його розмір.

Tree Shaking спеціально використовують для покращення ефективності у сучасних системах модульної розробки, таких як модулі ES6 у JavaScript. Багато сучасних інструментів збірки підтримують процес Tree Shaking (наприклад: Webpack, Rollup та інші). Ця технологія є важливою для оптимізації завантаження веб-застосунків, оскільки менший розмір файлів JavaScript дозволяє їм швидше завантажуватися та виконуватися в браузерах, що покращує відгук користувачів та продуктивність веб-сайтів. Тим паче це важливо в контексті бібліотек компонентів, оскільки бібліотека може мати багато компонентів не всі з яких будуть використані в цільовому застосунку, тому важливо прибирати зайвий код для зменшення розміру трафіку під час завантаження веб-сайту та первинного завантаження сторінки.

Вище перераховані проблеми вирішуються коректним налаштуванням конфігурацій збірки, адже саме під час збірки відбувається процес Tree Shaking та компіляція під конкретну платформу, тому розробнику бібліотеки важливо правильно вибрати інструменти збірки та вміти правильно їх налаштовувати.

1.5 Аналіз сучасних інструментів та фреймворків для створення бібліотек компонентів.

Оскільки об’єктом дослідження є бібліотеки компонентів для веб-інтерфейсів, що використовуються саме у фронт енд частині застосунків, потрібно визначити за допомогою які саме технології використовуються для цих задач.

Згідно з опитуванням компанії JetBrains у 2022 році (див. рис. 1.2) в якому брали участь близько 30 тисяч розробників [6] найпопулярнішими фронт енд фреймворками є:

- React - 55%;
- Vue.js - 35%;
- Angular - 17%;
- Angular JS - 7%;
- Svelte - 5%.

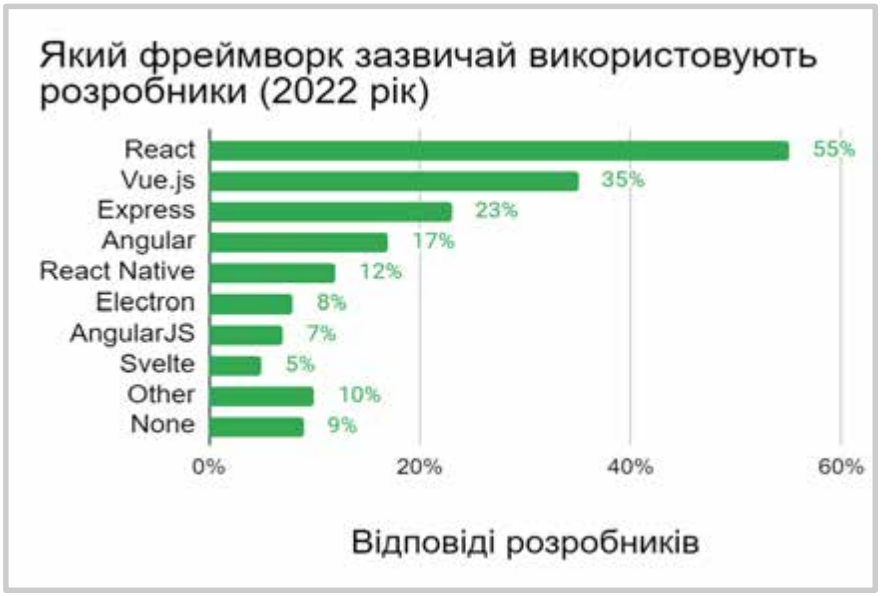

Рисунок 1.2 - результати опитування компанії JetBrains, щодо використання фреймворків розробниками за 2022 рік

Також в згідно цього опитування одним з найперспективніших технологій кореспонденти вважають “JavaScript та його фреймворки” (див. рис. 1.3), а саме 11% відповідей містили цю категорію.

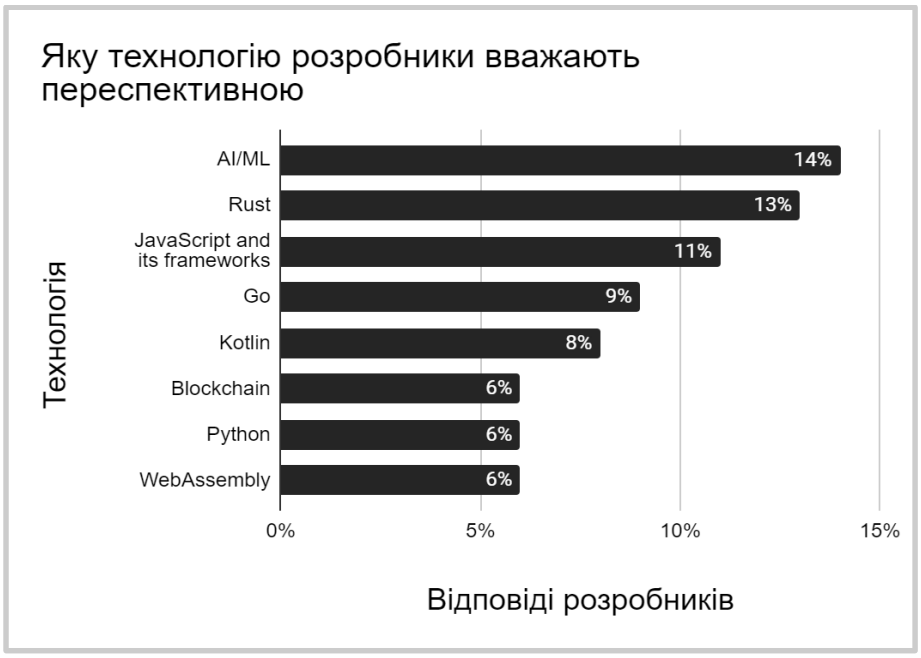

Рисунок 1.3 - результати опитування компанії JetBrains, щодо перспективних технологій на думку розробників за 2022 рік

Згідно з опитуваннями організації StateOfJS (див. рис. 1.4) найбільш використовуваними є: React, Vue.js, Angular, Svelte; 82%, 46%, 49%, та 21% відповідно [7].

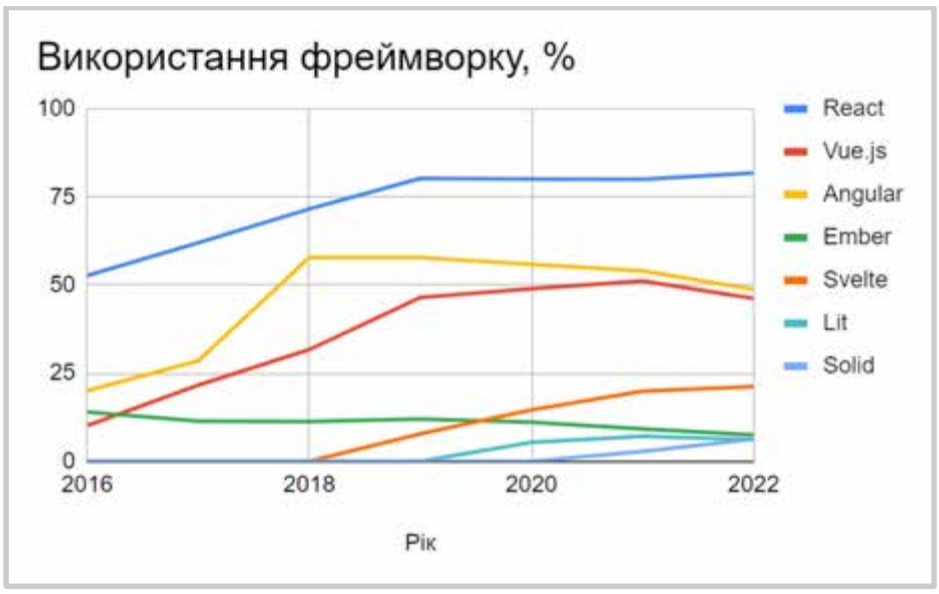

Рисунок 1.4 - результати опитування організації StateOfJS, щодо використання фреймворків у веб-розробці за 2016-2022 рік

Результати опитування компанії Stack Overflow за 2023 рік (див. рис. 1.5) мають схожі результати, але більше відрізняються від минулих [8]. Згідно з ними серед фронт енд (а також фулл стек) фреймворків найпопулярнішими є:

- React - 40.58%;
- JQuery - 21.98%;
- Angular - 17.46%;
- Vue.js - 16.38%
- WordPress - 13.38%;
- ASP.NET - 12.79%.

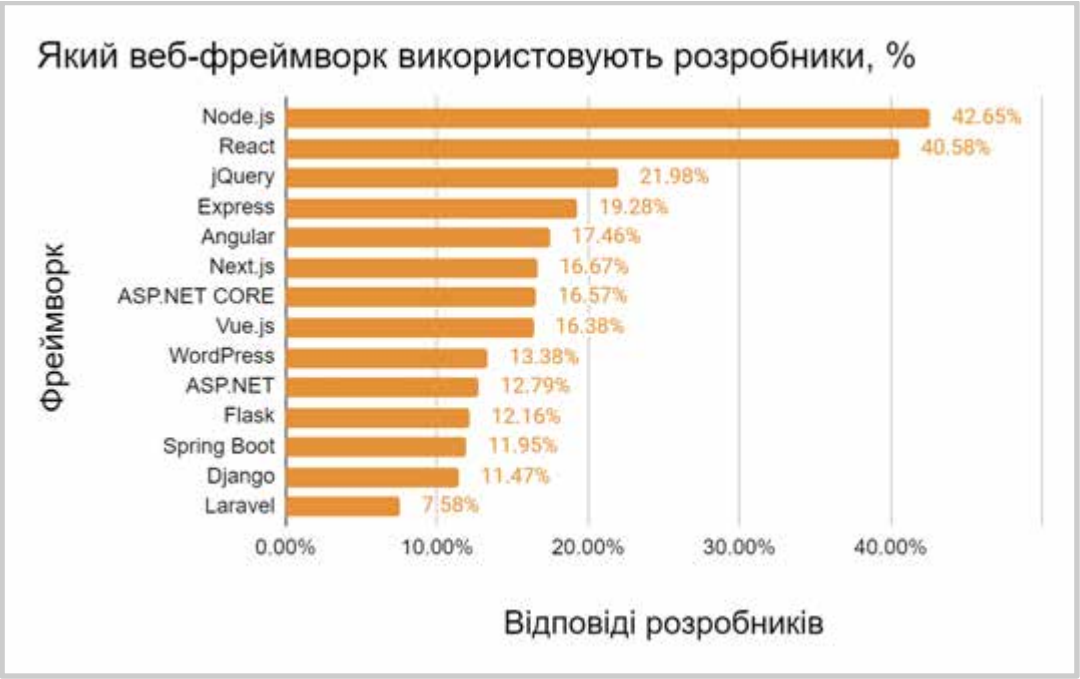

Рисунок 1.5 - результати опитування компанії Stack Overflow, щодо використання фреймворків у веб-розробці за 2023 рік

Як видно з вище зазначеної статистики здебільшого розробники використовують компонентно-орієнтовані фреймворки, для подальшого дослідження виділимо наступні фреймворки: React, Angular, Vue.js та Svelte.

В таблиці 1.1 наведена порівняльна характеристика, згідно з якою, можна виділити наступне:

- Всі фреймворки підтримують мову програмування TypeScript, а також мають власні засоби для реалізації властивості реактивності (здатність автоматично реагувати на зміни даних та оновлювати інтерфейс користувача відповідно до цих змін).
- Vue, Angular та Svelte - можуть бути компільовані у веб-компоненти, що дозволяє їх використання в різних веб-застосунках та фреймворках. В

React ці можливості можна реалізувати лише за допомогою сторонніх плагінів.

- Всі чотири фреймворки мають непогані показники продуктивності, проте Svelte сильно випереджає React та Angular. Щодо показників продуктивності Vue, хоч за останні роки вони покращились, Svelte все ж таки переважає з невеликим відривом, завдяки своїй компіляції на етапі збірки, відсутності віртуального DOM, та своєї реалізації реактивності [9].
- React та Angular мають велику та активну екосистему (різноманітні програмні засоби, бібліотеки, значні спільноти користувачів). Vue своєю чергою, хоч й має непогану за розміром і якістю екосистему, все ж поступається показникам React та Angular. Svelte, хоч і найменший з представлених фреймворків за розміром екосистеми, активно зростає та застосовується в проєктах компаній, які цінують його швидкість та простоту використання [10].
- Svelte на відміну від інших не можливо використовувати з іншими фреймворками.

Таблиця 1.1

Порівняльна характеристика фронт енд фреймворків: React, Angular, Vue.js та Svelte

Характеристика	**React**	**Vue**	**Angular**	**Svelte**
Можливість використання TypeScript	Так	Так	Так	Так

Закінчення таблиці 1.1

Характеристика	**React**	**Vue**	**Angular**	**Svelte**
Здатність до TreeShaking	Так	Так	Так	Так
Присутність Virtual DOM	Так	Так	Так	Ні
Тип реактивності	За допомогою хуків та класових компонентів	Реактивні властивості та директиви Vue.js	RxJS, Angular Binding Syntax	Змінні та директиви
Підтримка JSX чи інших шаблонізаторів	JSX	Так	Так	Ні
Здатність до SSR	Так	Так	Так	Так
Продуктивність (відносно вказаних фреймворків)	середня	висока	низька	дуже висока
Можливість компіляції у web components	Ні (лише за допомогою сторонніх плагінів)	Так	Так	Так

Загалом, вибір між цими фреймворками залежить від потреб та задач конкретного проєкту. React та Vue мають широку підтримку та розширюваність, Angular підходить для великих підприємств та проєктів, а Svelte має значні переваги у своїй високій продуктивності та простоті використання.

1.6 Огляд популярних бібліотек компонентів інтерфейсу користувача.

Після аналізу фреймворків, потрібно проаналізувати популярні бібліотеки компонентів, котрі створені для цих фреймворків. Після огляду наявних бібліотек було обрано найбільш популярні на платформі GitHub, а саме:

- MUI для React [11];
- Vuetify – Vue [12];
- Angular Material - Angular;
- Svelte Material UI - Svelte.

Після порівняльного аналізу, що продемонстрований в таблиці 1.2, було виявлено, що всі вказані бібліотеки загалом дуже схожі й мають не так багато відмінностей. Головними відмінностями є: різна сумісність зі сторонніми бібліотеками, кількості наявних компонентів у бібліотеці та можливість до розширення (головною відмінністю в цьому випадку є особливості логіки роботи Svelte).

Таблиця 1.2

Порівняльна характеристика бібліотек компонентів: MUI, Vuetify, Angular Material, Svelte Material UI

Критерії	**MUI**	**Vuetify**	**Angular Material**	**Svelte Material UI**
Підтримка SSR	Підтримка SSR через Next.js	Підтримка SSR через Nuxt.js	Підтримка SSR через Angular Universal	Підтримка SSR через SvelteKit
Здатність до TreeShaking	Так	Так	Так	Так
Кількість компонентів	50+	80+	60+	40+
Сумісність зі сторонніми бібліотеками	Гарна, але потребує додаткових налаштувань	Гарна	Гарна, але потребує додаткових налаштувань	Може бути обмежена
Сумісність з tailwind	Так	Потребує налаштувань	Ні	Так
Підтримка	Активно розвивається, регулярні оновлення	Активно розвивається, регулярні оновлення	Офіційна бібліотека Angular, довготривала підтримка	Може бути менше оновлюваною (новий проєкт)

Закінчення таблиці 1.2

Критерії	**MUI**	**Vuetify**	**Angular Material**	**Svelte Material UI**
Розширюваність	Здатний до розширення за допомогою плагінів та тем	Легко розширюється за допомогою плагінів та тем	Можливість розширення за допомогою Angular компонентів та модулів	Може бути менше гнучкості порівняно з іншими через специфіку Svelte

Загалом після аналізу популярних бібліотек стає видно, що всі бібліотеки мають багато спільного, а отже вони побудовані за певними загальними правилами та підходами й мають певні загальні вимоги, а отже можна вивести всі ці правила та вимоги та на основі них створити загальний метод створення бібліотеки компонентів.

Отже, як було продемонстровано всі бібліотеки дуже схожі, проте основа (фреймворки) для яких вони створюються можуть сильно відрізняться, після перегляду основних найпопулярніших фронт енд веб-фреймворків для основи було обрано Vue.js, бо він є доволі продуктивний, й хоч поступається Svelte, проте має більшу спільноту та екосистему, має компіляцію у web components на рівні фреймворку, а також є доволі гнучким, що стане в нагоді для опису загальних методів створення бібліотек компонентів.

РОЗДІЛ 2 МЕТОДИКА СТВОРЕННЯ БІБЛІОТЕКИ

2.1 Процес створення бібліотек компонентів

Після огляду різних бібліотек компонентів та веб-застосунків загалом, було виділено декілька головних задач, які потрібно вирішити розробнику бібліотеки (див. рис. 2.1), а саме:

- Налаштувати експорти в packege.json - у файлі package.json бібліотеки компонентів, налаштовуються різні експорти для модулів, щоб визначити, які компоненти або функції мають бути доступні ззовні. Це особливо важливо, для створення бібліотек компонентів, яку інші розробники будуть використовувати у своїх проєктах.
- Налаштувати конфігурації процесу збірки - більш детально цей процес буде описано в наступних розділах, але загалом основні складнощі полягають у виборі оптимального збірника та його подальшім налаштуванні.
- Написання плагінів для оптимізації збірки - збірники мають можливість виконувати плагіни користувача, котрі допомагають більш детально та гнучко підлаштовувати процес збірки.
- Налаштування інтеграцій з фреймворками - потрібно налаштувати параметри бібліотеки під різні фреймворки, наприклад: Nuxt, Laravel, Astra і т.п. Оскільки ці фреймворки мають трохи різні конфігурації, потрібно це враховувати, адже ці фреймворки забезпечують роботу з бекендом застосунку [13].
- Вибір/створення архітектури - для бібліотеки потрібно створити власну, або використати готову архітектури. Сюди входять задачі: вибору фреймворку основи; структуризація проєкту (поділ проєкту на директорії, розділення між кодом та ресурсами); продумати спосіб тестування компонентів; вибір інструментів для шаблонізації, стилів, логіки -

компонентів. При цьому потрібно дотримуватися стандартів та забезпечити можливості для подальшого розширення бібліотеки.

- Написання документації для компонентів - потрібно розробити чітку та докладу документацію для кожного компонента та роботи з бібліотеки загалом. Для кращого розуміння роботи та тестування компонентів потрібно також розробити демонстраційні приклади використання для кожного компонента, в яких потрібно показати всі можливі способи його використання.

Рисунок 2.1 - Діаграма прецедентів створення бібліотеки компонентів

Отже, процес створення бібліотеки є доволі складним тому, що вимагає від розробника досвіду роботи з веб-фреймворками та розробки компонентів, а також мати навички та знання про роботу процесу збірки.

2.2 Способи написання компонентів

Розберемо більш детально задачі з вибору інструментів для шаблонізації, задання стилів та логіки компонентів. Оскільки основою для бібліотеки було вибрано веб-фреймворк Vue.js, розглянемо його інструменти та способи розв'язання цих задач.

2.2.1 Інструменти шаблонізації

У Vue.js є кілька способів описання компонентів: Template Syntax, JSX (JavaScript XML) та Render Function (функція рендерингу). Кожен з цих підходів має свої особливості, розглянемо кожен із них детальніше.

Template Syntax (див рис. 2.2) - це найпоширеніший спосіб описання компонентів у Vue.js [14]. Його можна використовувати з HTML-подібним синтаксисом для визначення структури компонента та його поведінки. Важливо відзначити, що Vue компілює цей синтаксис у JavaScript, що дозволяє використовувати його у браузері.

```template
<div>
  <div v-if="ok">yes</div>
  <span v-else>no</span>
</div>
```

Рисунок 2.2 - Приклад опису компонента за допомогою Template Syntax

Перевагами такого способу описання компонентів є:

- Простий та зрозумілий синтаксис, який нагадає звичайний HTML.
- Легке відлагодження. Для багатофункціональних компонентів використання шаблонів може зробити відлагодження та налагодження процесу значно простіше та менш складним.
- Швидке розгортання: для простих проєктів та великих проєктів з великою кількістю команд, Template Syntax може бути швидшим способом розгортання компонентів.

Недоліки використання:

- Менша гнучкість: в порівнянні з JSX та Render Function, Template Syntax може бути менш гнучким для складних випадків, таких як динамічне створення компонентів або обробка подій.
- Обмежені можливості JavaScript: шаблони не можуть безпосередньо взаємодіяти з JavaScript кодом у порівнянні з іншими підходами, такими як JSX або Render Function.
- Обмежені можливості оптимізації: шаблони можуть бути менш ефективними для оптимізації рендерингу у деяких випадках, особливо для складних вузлів рендерингу, бо їх потрібно компілювати в звичайний JavaScript.

JSX (див. рис. 2.3) - це розширення синтаксису JavaScript, яке дозволяє вбудовувати HTML-подібний код безпосередньо в JavaScript [15]. Цей підхід надає більшу гнучкість і зручність для опису структури компонента, але вимагає підключення Babel або іншого компілятора JSX для перетворення коду.

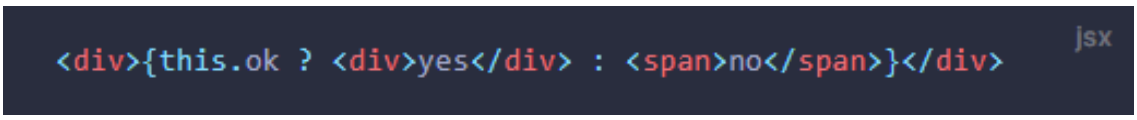

Рисунок 2.3 - Приклад опису компонента за допомогою JSX

Переваги JSX:

- Гнучкість та ясність: JSX надає гнучкість, дозволяючи вбудовувати JavaScript логіку безпосередньо в HTML-подібний синтаксис. Через те, що використовуються простий JavaScript код стає простішим.
- Розширені можливості: JSX дозволяє використовувати всі можливості JavaScript, включаючи змінні, умови, цикли та функції безпосередньо для рендерингу. Це полегшує складні операції, такі як динамічне створення компонентів та управління подіями.

Недоліки JSX:

- Складний синтаксис: в порівнянні з шаблонами синтаксис JSX може бути складнішим в розумінні, а також більш заплутаним в структурі верстки.
- Конфігурація: для використання JSX потрібно налаштувати інструменти, такі як Babel, для трансляції коду до стандартного JavaScript, що може вимагати додаткового часу та конфігурацій.
- Комплексність файлів: використання JSX може спричинити роздрібненість синтаксису в JavaScript-файлах, що при деяких умовах може зробити код менш читабельним.

Render Function (див. рис. 2.4) - це найгнучкіший спосіб опису компонентів, який використовує JavaScript-функції для рендерингу компонентів.

Цей підхід надає повний контроль над процесом рендерингу, але вимагає глибокого розуміння внутрішньої структури Vue та віртуального DOM [15].

```js
h('div', [this.ok ? h('div', 'yes') : h('span', 'no')])
```

Рисунок 2.4 - Приклад опису компонента за допомогою RenderFunction

Переваги Render Function:

- Повний контроль над компонентом: рендер функції надає повний контроль над процесом рендерингу компонентів та надає можливість динамічно створювати та управляти структурою компонентів.
- Універсальність: рендер функції можуть бути використані для створення будь-якої структури DOM, включаючи складні діаграми, анімації або інші специфічні інтерфейси.
- Полегшене тестування: через те, що рендер функції - це JavaScript, їх легше тестувати за допомогою різноманітних фреймворків для тестування.
- Гарна оптимізація: рендер-функції можуть бути оптимізовані для уникання зайвого рендерингу, використовуючи мемоїзацію (англ. “Memoization”) та інші оптимізаційні техніки.

Недоліки Render Function:

- Складність та велика кількість коду: рендер-функції можуть бути складнішими для вивчення та розуміння, особливо для новачків. Вони можуть також виглядати дуже довгими, особливо для складних компонентів.

- Відсутність обов'язкової структури: ендер-функції можуть легко перетворитися на заплутані та важкі для розуміння, якщо не дотримуватися певних патернів програмування.
- Велика складність для малих проєктів: для невеликих компонентів та задач, рендер-функції можуть бути занадто складними.

Отже, як видно з аналізу кожен метод має свої переваги та недоліки. Наприклад Render Function має високу гнучкість та продуктивність, проте являється дуже складним в розумінні. JSX формат є більш оптимізований ніж Template Syntax, проте все ж таки має складніший синтаксис для звичайної верстки. Template Syntax є найближчим до звичайного HTML, але через це є менш продуктивним, але через свій легкий синтаксис, саме його частіше всього використовуються для розробки компонентів під Vue.js.

2.2.2 Способи задання стилів

Стилі - це набір правил, які визначають зовнішній вигляд елементів на веб-сайтах або в веб-застосунках. Ці правила вказують, як елементи мають бути відображені на сторінці: розмір, колір, шрифт, відступи, рамки тощо.

Для задання стилів використовують відповідні таблиці стилів, наприклад для Vue.js можна використовувати: звичайний CSS, SCSS, Styled Components, CSS-in-JS [16].

Розглянемо їх трохи детальніше.

Стандартний CSS (Cascading Style Sheets) можна використовувати для стилізації компонентів у Vue.js. Стилі можна вказувати безпосередньо у секції

“<style>” компонента в одному файлі або в окремих файлах і підключити їх за допомогою import або require у вашому компоненті. Приклад використання продемонстровано на рисунку 2.5.

```
<template>
  <div class="my-component">Hello, Vue!</div>
</template>

<style>
.my-component {
  color: red;
  font-size: 18px;
}
</style>
```

Рисунок 2.5 - Приклад використання звичайного CSS для задання стилю

SCSS (див. рис. 2.6) - це розширення синтаксису CSS, яке додає багато корисних функцій, таких як змінні, вкладені селектори, міксіни тощо, при цьому не замінюючи функціонал оригінального CSS. У Vue.js, можна використовувати SCSS для визначення стилів, також за допомогою секції “<style>”.

```
<template>
  <div class="my-component">Hello, Vue!</div>
</template>

<style lang="scss">
.my-component {
  color: $primary-color;
  font-size: 18px;
}
</style>
```

Приклад 2.6 - Приклад використання SCSS для задання стилю

CSS-in-JS - це підхід, де стилі пишуться безпосередньо в JavaScript об'єкті. У Vue.js можна використовувати бібліотеки, такі як Emotion або styled-components, для застосування цього підходу, приклад зображено на рисунку 2.7.

```
<template>
  <div :css="styles">Hello, Vue!</div>
</template>

<script>
import { css } from 'emotion';

export default {
  data() {
    return {
      styles: css`
        color: red;
        font-size: 18px;
      `
    };
  }
};
</script>
```

Рисунок 2.7 - Приклад використання CSS-in-JS для задання стилю

Styled Components - це бібліотека, яка за допомогою CSS-in-JS дозволяє писати CSS як JavaScript код за допомогою шаблонних рядків. Це дозволяє динамічно змінювати стилі на основі змінних та властивостей компонентів, приклад на рисунку 2.8.

```vue
<template>
  <div class="my-component">Hello, Vue!</div>
</template>

<script>
import styled from 'vue-styled-components';

const MyComponent = styled.div`
  color: ${props => props.primary ? 'red' : 'blue'};
  font-size: 18px;
`;

export default {
  components: {
    MyComponent
  }
};
</script>

<template>
  <MyComponent primary>Hello, Vue!</MyComponent>
</template>
```

Рисунок 2.8 - Приклад використання Styled Components для задання стилю

Загалом можна підсумувати:

- Використання звичайного CSS найпоширеніший спосіб стилізації в Vue.js. Це простий синтаксис, який підтримується у всіх браузерах. Недоліком може бути менша гнучкість в порівнянні з іншими способами.
- SCSS - додає корисні функції, такі як змінні та вкладені селектори. Це полегшує написання та розширення CSS, але вимагає компіляції до звичайного CSS, що потребує додаткових інструментів.
- CSS-in-JS - забезпечує ізольованість стилів для компонентів та можливість використовувати динамічні стилі. Але, такий підхід може бути незвичним для розробників, які звикли до традиційних CSS-підходів.

- Styled Components - дозволяє створювати динамічні стилі на основі властивостей компонентів, але має такі ж проблеми розуміння, що й CSS-in-JS.

Отже, для опису стилів компонентів бібліотеки в загальному випадку краще використовувати SCSS, адже він має дуже схожий синтаксис до звичайного CSS, при цьому має багато зручних додаткових елементів для написання стилів.

2.2.3 Способи задання логіки

У Vue.js для написання логіки можна використовувати мови програмування JavaScript або TypeScript. Також у Vue.js існує два способи описання компонентів: опційне та композиційне [17].

Опційне API - це традиційний спосіб використання Vue.js, де ви описуєте компоненти за допомогою об'єктів, які містять різні властивості та методи, такі як data, methods, computed, watch і т.д. Це є стандартним і найбільш поширеним способом роботи з Vue.js до версії 3.0. Опційне API дозволяє легко зрозуміти, як працює компонент, та є дуже популярним у розробці.

Композиційне API - це новий спосіб роботи з Vue.js, який був представлений у версії 3.0. Замість використання одного великого об'єкта з опціями компонента, композиційне API дозволяє розбити логіку компонента на менші та частини, які можна легше повторно використовувати, за допомогою функцій композиції. Це робить код більш читабельним, структурованим та підтримується відмінною системою типізації.

Візьмемо для прикладу компонент провідника папок із GUI Vue CLI: цей компонент відповідає за такі логічні проблеми:

- Відстеження поточного стану папки та відображення її вмісту.
- Керування навігацією папками (відкриття, закриття, оновлення...).
- Керування створенням нової папки Перемикання показу лише улюблених папок.
- Перемикання показу прихованих папок.
- Обробка змін поточного робочого каталогу.
- Оригінальна версія компонента була написана в Options API.

Якщо ми надамо кожному рядку коду колір на основі логічної проблеми, з якою він має справу, з використанням різних API то отримаємо два файла, як видно з рисунка 2.9 при опційному API логічні блоки розташовані вперемiш, а при композиційному вони згруповані разом.

Рисунок 2.9 - Порівняння використання опційного та композиційного API

Отже, в результаті можемо побачити, що Vue.js третьої версії має доволі гнучкі способи для опису компонентів, адже в цій версії можна використовувати, як опційне, так і композиційне API в залежності від поставлених задач та бажання розробників.

В рамках розробки бібліотеки компонентів краще використовувати все ж таки композиційне API, адже при великій кількості логічних задач компонента цей підхід підвищує розуміння та чистоту коду.

2.3 Побудова бібліотек компонентів

Як було вище зазначено, важливим етапом створення бібліотек компонентів є процес збірки, відповідно для розробника бібліотеки важливо правильно вибрати інструменти збірки та налаштувати їх.

Збірник (англ. “bundler”) - це інструмент розробки веб-застосуків, який використовується для об'єднання (або збирання) різних файлів і ресурсів в один або декілька більших файлів, які зазвичай мають оптимізований формат для розповсюдження через мережу Інтернет [18]. Процес збірки містить в собі саму збірку і стискання (мініфікацію) файлів JavaScript, CSS, зображень та інших ресурсів, а також керування залежностями між ними.

Основною метою збірника є полегшення процесу розробки веб-застосунків та оптимізація з точки зору завантаження для користувачів.

Головними аспектами роботи збірника є:

- Об'єднання файлів: об’єднання декількох файлів у один або декілька більших файлів. Наприклад, може бути об'єднано декілька файлів

JavaScript або CSS в один файл, щоб зменшити кількість запитів, які браузер повинен робити для завантаження сторінки.

- Мініфікація: процес мініфікації (стискання) коду - зменшує розмір файлів шляхом видалення зайвих пробілів, коментарів та інших зайвих символів.
- Керування залежностями: керування залежностями між різними модулями та бібліотеками, розв'язуючи потенційні конфлікти та допомагаючи встановити правильний порядок завантаження файлів.
- Інші оптимізації: деякі збірники також включають різні оптимізації, такі як Tree Shaking, оптимізація зображень, оптимізація шляхів та інші.

Розберемо процес збірки більш детально покроково (див. рис. 2.10):

1. Налаштування проєкту:
    - Встановлення залежностей: встановлення необхідних пакетів та бібліотек збірки за допомогою інструменту керування пакетами, такого як npm або yarn.
    - Конфігурація файлу збірки: створення конфігураційного файлу для інструменту збірки, де вказуються правила збірки, включаючи шляхи до вихідних та вихідних файлів, правила трансформації, та інші опції.
2. Збірка Ресурсів:
    - Збірка JavaScript: збірка JavaScript-файлів, включаючи модулі та залежності, мініфікація коду та видалення невикористовуваних частин (Tree Shaking).
    - Збірка CSS: збірка та мініфікація CSS-файлів, включаючи обробку препроцесорів, якщо вони використовуються.

- Оптимізація зображень та медіафайлів: оптимізація розміру та стиснення зображень та медіафайлів для зменшення їхньої ваги.

3. Обробка та Збірка HTML-сторінок:
    - Збірка HTML: генерація HTML-файлів або шаблонів, вставка шляхів до зібраних ресурсів (JavaScript, CSS, зображень тощо).
    - Підключення скриптів та стилів: додавання посилань на зібрані JavaScript- та CSS-файли до HTML-файлів.
4. Вивід Результату:
    - Генерація збірки: створення одного або декількох збірок зі зібраних та оптимізованих файлів.
    - Вивід збірки: збереження зібраної аплікації (бандлу) у вказаному вигляді (один або кілька файлів) для розгортання на веб-сервері.

Рисунок 2.10 - Алгоритм процесу збірку

Одними з найпопулярніших сучасних збірників є Webpack, Rollup та Parcel. Вони допомагають розробникам створювати ефективні та оптимізовані веб-застосунки, забезпечуючи оптимальну продуктивність та швидке завантаження веб-сторінок.

2.4 Конфігурація збірників.

Налаштування конфігурації для збірки веб-застосунку залежить від конкретного інструменту збірки (наприклад, Webpack, Rollup, Parcel тощо) та конкретних потреб проєкту.

Загалом конфігураційний файл - це файл, який містить набір параметрів та правил для інструменту збірки, що допомагає визначити, як саме має виглядати та поводитися збірка вашого веб-застосунку.

Загальними підходами до налаштування конфігурацій є:

1. Встановлення Основних Параметрів:
    - Вхідні файли (Entry Points): посилання на основні файл(и) вихідного коду, з яких починається збірка.
    - Вихідні файли (Output Files): шляхи куди зберігати зібрані файли та як їх назвати.
2. Робота з Модулями та Залежностями:
    - Завантажувачі (Loaders): правила для обробки різних типів файлів, наприклад, компіляція ES6 до ES5 або обробка CSS препроцесорами.
    - Плагіни (Plugins): додаткові інструменти, які допомагають в різних аспектах збірки, такі як оптимізація, генерація HTML-файлів, мініфікація тощо.
3. Оптимізація та Розширення:
    - Мініфікація та Tree Shaking: інструкція, як мініфікувати код та вилучати невикористаний код.
    - Оптимізація зображень та медіафайлів: правила для автоматичної оптимізації зображень.
    - Кешування (Caching): налаштування для правильного кешування файлів для оптимізації завантаження.
4. Розширення Функціональності:
    - Інтеграція з іншими інструментами: налаштування для інтеграції з іншими інструментами розробки, такими як сервери розробки, лінтери та тестувальні інструменти.

- Розширення можливостей вашого збірника: вибір плагінів та додаткових інструментів, які надають конкретні функції або покращують процес збірки.

Конфігураційний файл може бути написаний в форматі JSON, JavaScript або інших, залежно від інструменту збірки. Важливо враховувати, що кожен інструмент має свій власний синтаксис та особливості налаштувань. Кожен інструмент, який використовується, потребує окремої уваги для правильного налаштування конфігураційного файлу.

2.5 Плагіни для збірників.

Після аналізу найпопулярніших збірників, а саме: Webpack, Rollup, Parcel - було обрано Rollup через його швидкість роботи та простоти API для розробки плагінів. У Rollup, плагіни та завантажувачі (loaders) грають ключову роль у збірці веб-застосунків. Плагіни дозволяють розширити можливості Rollup, додаючи різні функції, в той час, як завантажувачі використовуються для обробки різних типів файлів під час збірки.

У конфігурації Rollup, плагіни та завантажувачі вказуються у полі plugins. При цьому важливо враховувати послідовність, в якій вони вказані, оскільки це може вплинути на результати збірки. Плагіни виконуються в порядку, вказаному у конфігурації, зліва направо. Це означає, що порядок завантажувачів та плагінів може мати значення, особливо якщо один плагін модифікує код, який інший плагін обробляє.

Для оптимізації процесу збірки потрібно за допомогою плагінів розв'язати наступні проблеми:

- Підключення стилів для збірки під SSR, адже при збірці на стороні сервера можливі помилки при не коректному використанню стилів компонентів;
- Корегування імпортів для забезпечення Tree Shaking;
- Підключення вкладених стилів у webcomponents.

Отже, за допомогою плагінів збірки можна більш точно керувати процесом збірки, та підлаштовувати її під потреби розробника бібліотеки.

2.6 Документація бібліотеки компонентів.

Важливим елементом бібліотеки компонентів є її документація, адже саме вона пояснює користувачам як правильно працювати та використовувати розроблені компоненти, тому забезпечення якісної та зрозумілої документації має бути однією з найпріоритетніших задач для розробника бібліотеки [19].

Загалом після огляду декількох бібліотек компонентів і їх документації можна сформувати ряд порад для формування, або покращення документації, а саме:

- Інструкції щодо підключення та встановлення бібліотеки для різних платформ.
- Кожен компонент та повинен мати сформовану метаінформацію, а саме: список компонентів з яких він складається, його аргументи («props») та їх типи, методи, події («events») та його текстовий опис.
- Кожен компонент повинен мати демонстраційну сторінку, на якій будуть представлені всі можливі варіанти використання, що забезпечить можливість для тестування роботи цих компонентів, та надалі формування

сторінки документації цього компонента з використанням цих демонстрацій.

- Кожен приклад використання в документації повинен надавати доступ до свого коду (у вигляді тексту коду, або посилання на пісочницю (“sandbox”) з використанням цього компонента).
- При кожному оновлені бібліотеки потрібно забезпечити підтримки актуальності документації.
- Для підтримки користувачів потрібно вказати (бажано декілька) каналів зв’язку, за допомогою яких користувачі зможуть давати зворотний відгук.

Отже, процес документування бібліотеки є вкрай важливим, та доволі часозатратним, тому при можливості потрібно максимально автоматизувати процеси його, постійно потрібно підтримувати актуальність та якість документації, а також максимально підтримувати зв’язок з користувачами для подальшого покращення бібліотеки.

РОЗДІЛ 3 ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Принципи розробки та архітектура бібліотек компонентів.

Після вище зазначеного аналізу способів та технологій побудови бібліотеки було обрано основні інструменти для подальшого формування методу та інструменту для його реалізації, а саме:

- В ролі фреймворка вибрано Vue.js.
- Мова програмування - TypeScript.
- Спосіб описання компонентів - Composition API.
- Формат шаблонізатора у Template Syntax.
- Описання стилів за допомогою SCSS.
- Збірник - Rollup, Vite, Unbuild.

Далі, потрібно визначити структуру бібліотеки, після дослідження ряду бібліотек під різні фреймворки було переглянуто загальну структуру бібліотеки (див. рис. 3.1).

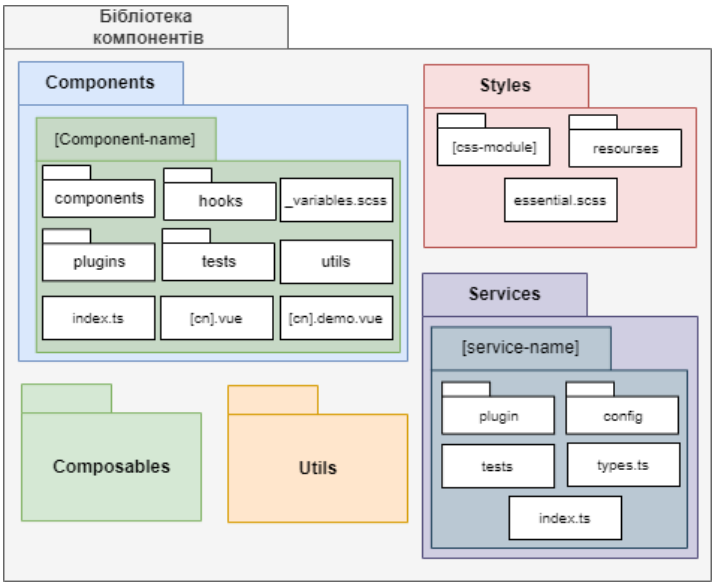

Рисунок 3.1 - Діаграма пакетів загальної архітектури бібліотек компонентів

Оскільки для реалізації методу було обрано фреймворк Vue.js, то потрібно внести корегування саме для цього фреймворку, але через його особливості та модульності й гнучкості структури даний варіант можна легко підлаштувати під інші фреймворки.

Як видно з представленої діаграми пакетів на рисунку 3.1, бібліотека складається з таких пакетів:

- Components - кожен компонент є ізольованим модулем, який містить код для рендерингу компонента, стилі та скрипти для реалізації поведінки компонента. Сам по собі компонент є файлом формату .vue, який містить:
  - Template для опису структури компонента за допомогою мови HTML.
  - Style для опису стилів компонента на мові SCSS (яка компілюється у CSS).
  - Script для написання логіки роботи компонента на мові TypeScript.Скрипт зазвичай дуже великий порівняно зі структури, тому часто окремі частини логіки роботи компонента винесено у спеціальні структури composables (або hooks) - це дає змогу розділити відповідальність окремих логічних модулів на різні файли.
  - Інколи компоненту потрібно мати свій плагін, який розширює можливості фреймворку, додаючи до нього глобальні метод, стан тощо. Наприклад, функція закриття усіх модальних вікон, функція для створення повідомлення та інше.
  - Для кожного компоненту є візуальні та unit-тести. Візуальні тести представленні у вигляді спеціальних story. Кожна story покриває можливе використання компонента. Перед створенням нової версії компонента story перевіряються на коректність роботи.

- Styles - пакет з ізольованими CSS модулями, які користувач зможе окремо підключати. Такі стилі не є обов'язковими та не впливають на роботу компонентів. Можна сказати це компоненти, які не потребують структури чи логіки. Окрім необов'язкових модулів до стилів відносяться:
  - Resources - функції на мові SCSS, які використовуються для спрощення написання стилів та не компілюються у фінальну збірку. До таких функцій відноситься часто повторюваний код, наприклад, функція зміни кольору "scrollbar", функція додавання контуру при фокусі та інше.
  - Essential.scss - спеціальний стильовий модуль, без якого робота бібліотеки вважається неможливою. До таких стилів можна віднести стилі кольорової теми, шрифти та інше.
- Composables (композиційні функції) - окремі частини коду на мові TypeScript, які відповідають за роботу окремої функціональності компонента. Часто компоненти мають схожу логіку роботи, яка повторюється. Для запобігання повторенню використовується композиційне API та спеціальні композиційні функції, які розширюють функціонал компонента. До таких маленьких частин функціональності легко писати тести. Прикладом композиційних функцій може бути функція отримання стану фону компонента, отримання стану елемента у фокусі, генерація стилів для компонента "dropdown" тощо.
- Utils - маленькі частини коду, які не потребують створення стану і не розширяє сам компонент, а спрощує саме написання коду. До таких функцій відноситься, наприклад, функція прокрутки до елементу.
- Services - великі, відокремлені модулі, які використовуються компонентами для коректної роботи. До таких сервісів відноситься, наприклад, сервіс кольорової схеми, переклади для тексту у компонентів тощо.

Отже, виділивши основні інструменти, структуру бібліотеки та описавши у минулому розділі загальні рекомендації та правила було сформовано загальну архітектуру бібліотеки компонентів, й хоча в конкретному прикладі вона спеціалізована під Vue.js, через особливості сучасних веб-фреймворків і їх спільні риси, дана архітектура є доволі гнучкою і може використовуватися й під інші фреймворки, а не тільки з Vue.js.

3.2 Розробка інструменту для реалізації створеної методики побудови бібліотеки компонентів

Описавши загальну архітектуру та методи розробки бібліотек компонентів було вирішено створити допоміжний інструмент-утиліту для реалізації створеної методики. В процесі аналізу та створення методики було виділено наступні вимоги (див. рис. 3.2):

- Користувач програми - це розробник бібліотеки.
- Користувач повинен мати змогу налаштувати конкретні конфігурації для збірки бібліотеки для вказаних ним цільових платформ.
- При запуску збірки бібліотеки повинен також формуватися шаблон документації за метаінформацією бібліотеки про власні компоненти.

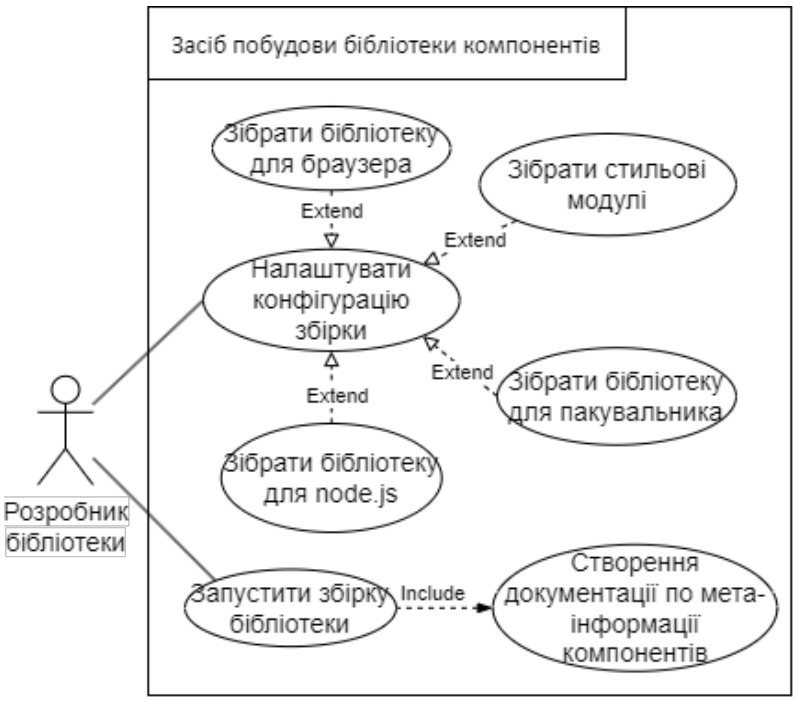

Рисунок 3.2 - Діаграма прецедентів для засобу побудови бібліотек компонентів

В процесі проєктування було вирішено, що інструмент для збірки бібліотеки компонентів має бути у вигляді програми для командного рядка. Програма має мати мінімальний набір аргументів, оскільки основне завдання такого інструменту стандартизувати розробку бібліотек. До таких аргументів відносяться:

- Entry file - вхідний файл, з якого експортується сама бібліотека, її компоненти та модулі.
- Out directory - вихідна директорія, де будуть зберігатися зібрані файли бібліотеки під різні формати.
- Target - бажані формати для збірки. Наприклад, користувач не хоче, щоб його бібліотеку можна було використовувати як Web Components.

Програма використовує vite для збірки бібліотеки під різні формати: es, esm-node, cjs, iffe, web-components. Для генерації TypeScript типів використовується надбудова над TypeScript Compiler (TSC) оптимізована під VueJS - VueTSC. VueTSC генерує типи компонентів беручі до уваги їх атрибути, слоти та події.

Для оптимізації збірки під SSR і TreeShaking було створено наступні плагіни: “Append component CSS”, “Component v-bind fix”, “Web components nested styles”, “Lodash import fix”.

Append component CSS - налаштування імпортів css-фалів для ES-збірки (див. рис. 3.3). Якщо не оптимізувати цей процес, то компоненти, залишатися без стилів, або потрібно буде додавати всі стилі в один файл й тоді стилі компонентів, які не використовуються, додадуться до загального файлу стилів й будуть займати зайве місце. Плагін додає стилі компонента в сам компонент, якщо цей компонент не буде використовуватися, то він і його стилі не увійдуть в фінальну збірку.

```
export const appendComponentCss = createDistTransformPlugin({
  name: 'append-component-css',

  transform (componentContent, path) {
    const { name, dir } = parsePath(path)

    const componentName = extractComponentName(name)

    if (!componentName) {
      return
    }

    const distPath = resolve(this.outDir)
    const relativeDistPath = relative(dir, distPath)
    const relativeFilePath = relativeDistPath + '/' + componentName.replace(/-.*$/, '') + '.css'

    // There are few cases how vite can store css files (depends on vite version, but we handle both for now):
    // CSS stored in dist folder (root)
    if (existsSync(resolve(dir, relativeFilePath))) {
      return appendBeforeSourceMapComment(componentContent, `\nimport '${relativeFilePath}';`)
    }

    // CSS stored in component folder
    const cssFilePath = `${dir}/${componentName}.css`

    if (existsSync(cssFilePath)) {
      return appendBeforeSourceMapComment(componentContent, `\nimport './${componentName}.css';`)
    }
  },
})
```

Рисунок 3.3 - Текст програми плагіну append-component-css

Component v-bind fix - через те, що при компіляції для SSR Vue.js видає помилки при спробі прочитання css-змінних, котрі використовуються через директиву “v-bind”, потрібно всі такі стилі перенести до атрибута “style” компонента, для коректного відображення при SSR (див. рис. 3.4).

```
export const componentVBindFix = (o: {
  sourcemap?: boolean
} = { sourcemap: false }): Plugin => {
  let cache = new Map<string, { code: string, map: any }>()

  return {
    name: 'component-v-bind-fix',
    enforce: 'pre',
    transform (code, id) {
      if (cache.has(id)) {
        return cache.get(id)
      }

      if (!/\.vue$/.test(id)) {
        return
      }

      const result = transformVueComponent(code)

      if (!result) { return }

      cache.set(id, result)

      if (o.sourcemap) {
        return result
      }

      return transformVueComponent(code)?.code
    },
  }
}
```

Рисунок 3.4 - Текст програми плагіну component-v-bind-fix

Web components nested styles - через те, що веб компоненти являються інкапсульовані відносно один одного [20] і знаходяться у власному Shadow DOM (технологія, що дає змогу приєднати дерево DOM до елемента та приховати внутрішні елементи цього дерева від JavaScript і CSS, що працюють на сторінці [21]) стилі дочірніх компонентів не підключаються автоматично під час збірки.

Тому плагін спеціально копіює стилі дочірніх компонентів та вставляє їх в Shadow DOM компонента, для коректного відображення (див. рис. 3.5).

```
/** Merge styles to custom element from child components */
export const webComponentsNestedStyles = createDistTransformPlugin({
  name: 'web-components-nested-styles',

  transform: async (componentContent, path) => {
    if (!path.endsWith('.js')) { return }

    /** Component store own styles from `<style>` in variables with `_style_` prefix */
    const vars = [...componentContent.matchAll(/[const|var] (_style_.*) = /gm)].map(([a, b]) => b)

    const componentList = getComponentsList(componentContent)

    if (componentList.length === 0) { return }

    return componentContent
      // Might be with styles or without styles
      // _export_sfc(_sfc_main, [["render", _sfc_render], ["styles", [/* variables */]]])
      .replace(/_export_sfc\(.*, \["styles", \[(.*)\]\]\]\)/, (str, substr) => {
        return str.replace(substr, generateComponentStylesAccessCode(componentList) + ', ' + vars.join(', '))
      })
      // _export_sfc(_sfc_main, [["render", _sfc_render]])
      .replace(/_export_sfc\(.*(\[\["render", _sfc_render\]\])\)/, (str, substr) => {
        return str.replace(substr, `[["render", _sfc_render], ["styles", [${generateComponentStylesAccessCode(
      })
  },
})
```

Рисунок 3.5 - Текст програми плагіну web-components-nested-styles

Lodash import fix - якщо якийсь компонент не правильно імпортує об’єкти з бібліотеки lodash, то в фінальну збірку додається вся ця бібліотека, що збільшує розмір цієї збірки зайвим кодом. Відповідно плагін просто виправляє неправильні імпорти на коректі, при яких до збірки додається код лише тих об’єктів які безпосередньо використовуються у компонентах (див. рис. 3.6).

```
const lodashPlugin: Plugin = {
  name: 'lodash-import-fix-plugin',
  transform(code, id) {
    if (id.endsWith('.ts') || id.endsWith('.js')) {
      const lodashImportRegex = /\bimport\s+\{\s*([^}]+)\s*\}\s+from\s+['"]lodash['"]/g;
      let transformedCode = code;
      let match;
      while ((match = lodashImportRegex.exec(code)) !== null) {
        const lodashImports = match[1]
          .replace(/\n/gm, '')
          .split(',')
          .map((importSpecifier) => importSpecifier.trim())
          .filter(Boolean);

        let imports = ""
        lodashImports.forEach((importSpecifier) => {
          const [originalName, alliasName=originalName] = importSpecifier.split(' as ');

          imports += `import ${alliasName} from 'lodash/${originalName.trim()}.js';\n`
        });
        transformedCode = transformedCode.replace(
          match![0],
          imports
        );
      }
      return {
        code: transformedCode,
        map: null,
      };
    }
    return null;
  },
};
```

Рисунок 3.6 - Текст програми плагіну lodash-import-fix

Отже, після налаштування плагінів для збірки, були розроблені окремі компоненти програми, котрі створюють вказані користувачем конфігурації збірки. Загальна схема застосунку продемонстрована на діаграмі компонентів (див. рис. 3.7) згідно з якою:

- Користувач (“розробник бібліотеки”) - отримує доступ до програми “утиліти для побудови бібліотеки” через командний рядок термінала.
- “Утиліта для побудови бібліотеки” за допомогою інших компонентів (код програми компонентів продемонстровано в додатку 2) створює конфігурації для збірки та запускає процес збірки за допомогою компонентів “Vite” та “TSC”.

- Компонент “Nuxt Module Generator” - модуль, що створює конфігурації для збірки бібліотеки під Nuxt;
- Компонент “Генератор метаінформації компонентів” - компонент, що проходиться по всім компонентам, та зберігає їх інформацію в окрему структуру, в результаті чого виходить структура з інформацією про всі компоненти бібліотеки;
- Компонент “Генератор конфігурацій” - компонент, що в залежності від вхідних даних, генерує конфігурації для збірки під формати: es, esm-node, cjs, iffe, web-components.

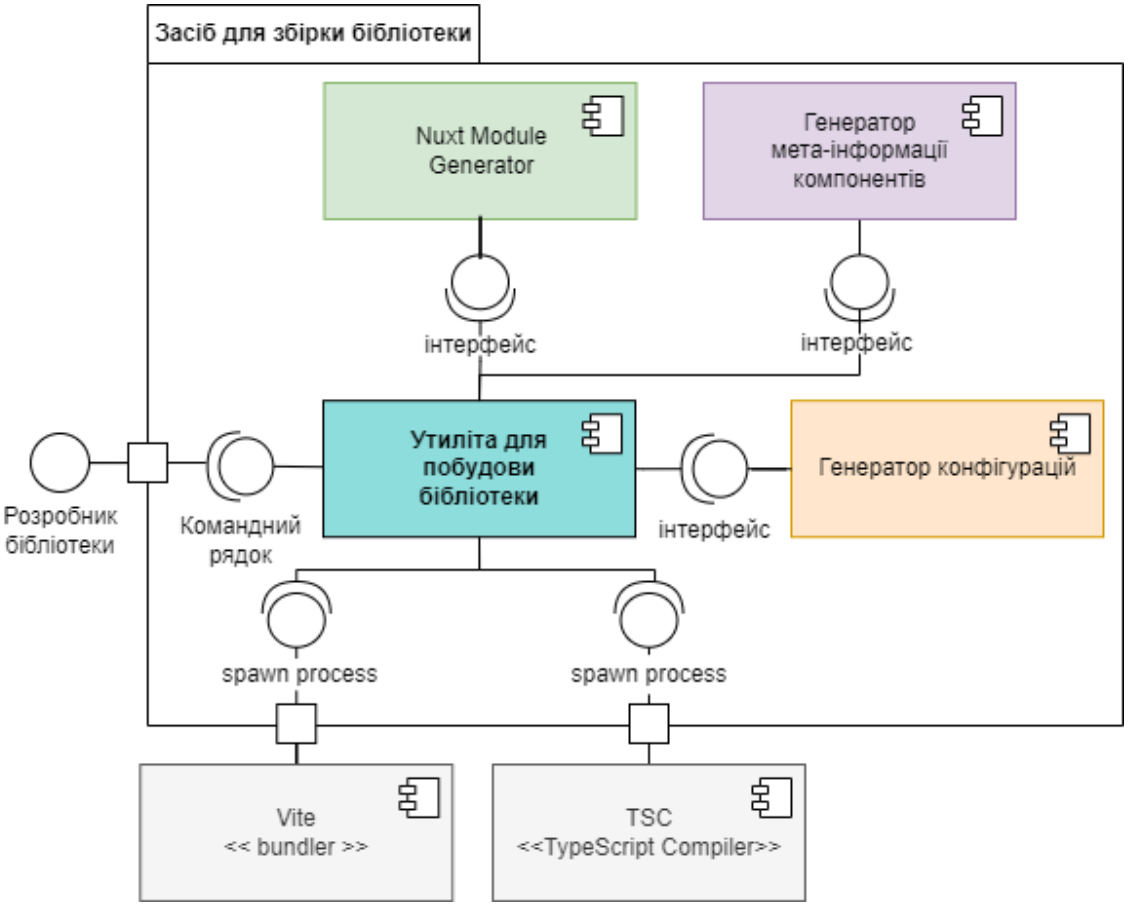

Рисунок 3.7 - Діаграма компонентів програми

Отже, було проаналізовані основні вимоги до застосунку для полегшення процесу збірки бібліотеки компонентів. Також було спроєктовано та розроблено застосунок утиліту та плагіни для оптимізації процесу збірки.

3.3 Тестування

Після розробки утиліти потрібно було розробити систему тестування для перевірки коректності роботи та правильної роботи бібліотеки після збірки. Для цього було розроблено ряд unit-тестів та e2e-тестів, що будуть забезпечувати правильність бажаної роботи застосунку.

Unit тести - це вид тестування програмного забезпечення, який спрямований на перевірку окремих компонентів чи функцій програми, щоб переконатися, що вони працюють коректно і відповідають специфікації.

Для застосунку були розроблені наступні unit-тести:

- Тести для перевірки роботи плагінів збірника;
- Тест котрий перевіряє коректність створених файлів конфігурації. Цей тест створює невелику штучну бібліотеку, що імітує структуру та роботу оригінальних бібліотек, після цього тест запускає утиліту зі збіркою для всіх можливих платформ, й перевіряє наявність та коректність створених конфігурацій.
- Тест для перевірки списку метаінформації про компоненти. Потрібно переконатися, що всі наявні компоненти бібліотеки, мають потрібну метаінформацію у вихідних даних утиліти.

E2E (end-to-end) тести - це вид тестів, які спрямовані на перевірку реальної взаємодії між компонентами чи системами програмного забезпечення у середовищі подібному до кінцевого користувача, тобто перевірка взаємодії з програмою, як це робить користувач у реальному використанні [22]. Ці тести

моделюють поведінку користувача, натискання кнопок, навігацію по сторінках, та інші дії в програмі, щоб перевірити, як система працює в цілому, а не лише окремі її компоненти.

Для застосунку було створено тести, котрі виконують наступне:

- Після процесу збірки, тест бере демонстраційний приклад використання кожного елемента, та формує з цих прикладів одну загальну сторінку, на якій знаходяться всі компоненти.
- Після отримання сторінки зі всіма компонентами, за допомогою різних docker-контейнерів запускається веб-застосунок в різних середовищах (nuxt-spa, nuxt-ssr, vite, vite-cjs, webpack).
- В кожному середовищі в браузері віртуальної машини контейнера відкривається сторінка веб-застосунку зі всіма компонентами та перевіряється консоль браузера на наявність помилок. Таким чином симулюється використання користувачем компонентів з бібліотеки у своєму застосунку, та перевіряється коректність роботи компонентів бібліотеки.

Отже, для застосунку-утиліти було створено ряд unit- та e2e-тестів, які забезпечують перевірку коректності роботи інструменту та бібліотек, котрі збираються за допомогою нього. В ході створення було виявлено, що надалі потрібно додати візуальне тестування в e2e-тестах для бібліотеки, щоб окрім коректності роботи програмно, також перевірявся зовнішній вигляд компонентів.

РОЗДІЛ 4 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

4.1 Розроблений метод

В ході дослідження було розроблено метод створення бібліотеки компонентів, він включає наступні елементи:

1. Використання компонентно-орієнтованих веб-фреймворків зокрема Vue.js;
2. Використання Template Syntax як шаблонізатор для верстки;
3. Використання SCSS для описання стилів компонентів;
4. Використання Compostion API для описання компонентів;
5. Використання розробленої файлової структури бібліотеки;
6. Створення демонстрацій використання компонентів для їх тестування та подальшого використання в документації;
7. Використання розробленої утиліти для зручної збірки бібліотеки під різні платформи.

Розроблений метод є доволі гнучким й при необхідності деякі елементи можна змінювати, загалом цей метод підходить для проєктів різного рівня, й головним чином розв'язує для користувача (розробника бібліотеки) проблеми вибору технологій, інструментів, структури та збірки бібліотеки компонентів.

4.2 Розроблений програмний інструмент

Для реалізації розробленого методу повною мірою було розроблено застосунок-утиліту, який повинен полегшити процес збірки бібліотеки компонентів.

Для використання цього засобу користувач повинен його завантажити та за допомогою терміналу взаємодіяти з утилітою.

Для прикладу було завантажено бібліотеку компонентів з відкритим кодом Vuestic UI та завантажено утиліту. Після цього було запущено утиліту з аргументами (див. рис. 4.1).

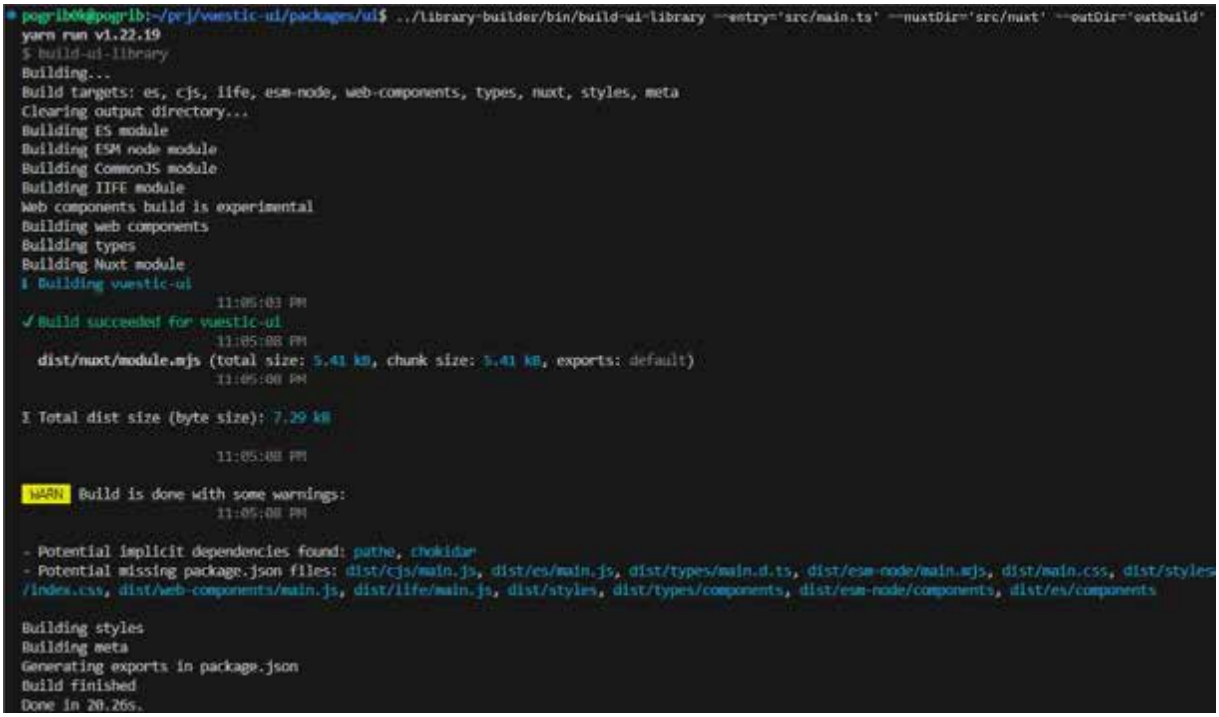

Рисунок 4.1 - Приклад використання утиліти

За замовчуванням утиліта запускає збірку під всі можливі платформи (див. рис. 4.2): cjs, es, esm-node, iife, nuxt, web-components. А також зібрані стилі компонентів в директорії “styles” та метаінформація про компоненти в фалі “meta.json”.

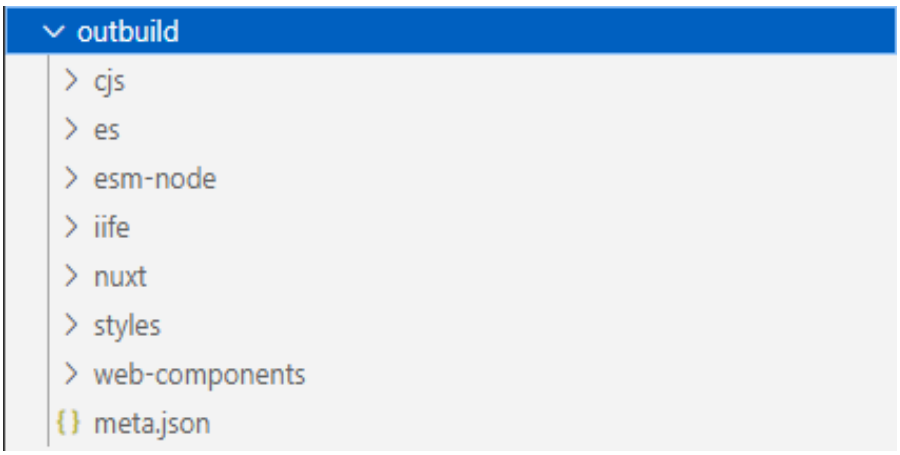

Рисунок 4.2 - Приклад зібраної бібліотеки під різні платформи

Також, якщо користувачу не потрібно збирати бібліотеку під конкретний формат, то потрібно передати як аргумент назву цього формату зі значенням “false” (див. рис. 4.3), й цей формат пропуститься під час збірки.

```
pogrib0k@pogrib:~/prj/vuestic-ui/packages/ui$ ../library-builder/bin/build-ui-library --entry='src/main.ts'
 --nuxtDir='src/nuxt' --outDir='outbuild' --cjs=false --iife=false
```

Рисунок 4.3 - Приклад запуску утиліти без збірки під платформи cjs та iife

При використанні даного інструмента бажано мати запропоновану файлову структуру бібліотеки (див. рис. 4.4), інакше можливо буде потрібно підлаштовувати деякі скрипти.

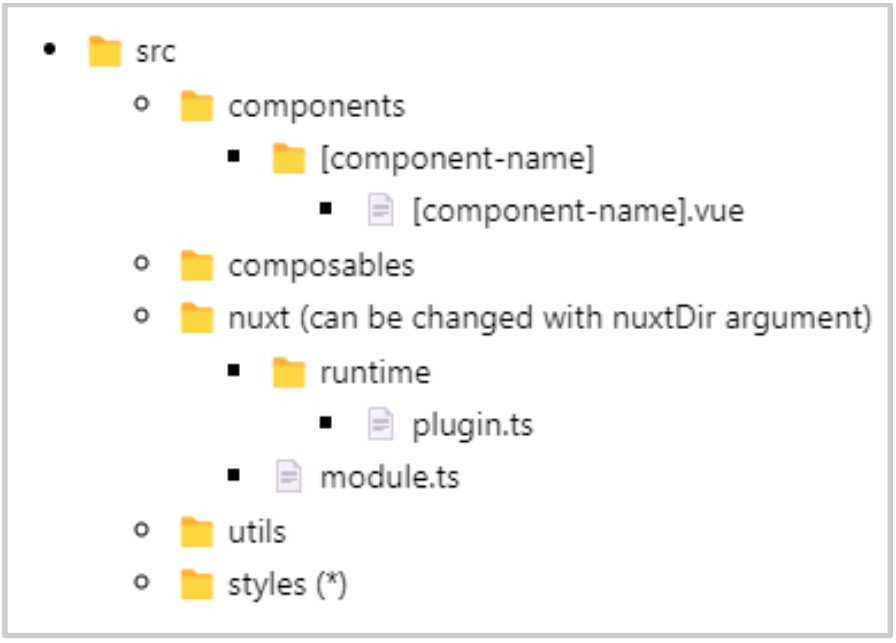

Рисунок 4.4 - Рекомендована файлова структура бібліотеки

Використовуючи вказану структуру утиліта автоматично може знаходити необхідні файли модулів, що прибирає зайвий процес конфігурації шляхів модулів та компонентів бібліотеки.

В ході використання цього засобу було виявлено, що хоча він й надає засоби для створення метаінформації компонентів та в методиці є рекомендації щодо створення демонстрацій роботи компонентів для формування документації, проте цей процес все ще залишається часозатратним й надалі в майбутніх дослідженнях було б непогано автоматизувати процес створення документації, щоби при кожній збірці бібліотеки документація могла автоматично підлаштовуватися під зміни автоматично. На разі є деякі інструменти (наприклад, Docus, Storybook [23] і т.п.), але для автоматизації їх роботи потрібно більш детальне їх вивчення. Додавання такого автоматизованого процесу сильно б покращило процес створення бібліотек загалом.

ВИСНОВКИ

В ході виконання магістерської роботи було проаналізовано сучасні бібліотеки компонентів користувацьких веб-інтерфейсів, їх інструменти та методи розробки. Був проведений аналіз й порівняння сучасних веб-фреймворків на яких базуються сучасні бібліотеки компонентів. На основі проведеного аналізу було визначено основні принципи, інструменти, технології проєктування та створення бібліотек компонентів. Була описана загальна структура бібліотек компонентів.

Також був розроблений метод для створення бібліотек компонентів та створений програмний застосунок з використанням фреймворку Vue.js для реалізації розробленої методики. Розроблений застосунок полегшує для потенційного розробника процес створення структури, вибору методів й інструментів для побудови (збірки) бібліотеки компонентів. Для оптимізації процесу збірки було створено декілька плагінів, які вирішують ряд проблем оптимізації збірки під різні платформи. Запропонований метод можна використовувати при задачах створення нових бібліотек компонентів у веб-орієнтованих програмних застосунках різного рівня, для оптимізації часу розробки таких застосунків.

В ході тестування розробленого програмного застосунку виявилось, що задача з формуванням документації бібліотеки є доволі складною, й надалі можна використати напрацювання цього дослідження для створення інструменту автоматизації процесів створення документації компонентів бібліотеки. На основі виявлених методів можна зробити покращення у напрямку документації та тестування бібліотек компонентів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Intro to Frontend Development [Електронний ресурс] – Режим доступу до ресурсу: https://flexiple.com/frontend/deep-dive. Дата доступу: жовтень 2023.
2. Web frameworks – overview and classification [Електронний ресурс] – Режим доступу до ресурсу: https://www.ionos.com/digitalguide/websites/web-development/web-frameworks-an-overview/. Дата доступу: жовтень 2023.
3. Derleth T. Corporate Component and Service Libraries — A concept for creating, maintaining and managing a company-specific user interface component library in the field of frontend web development / Derleth Thomas – Stuttgart, 2018. – 11 с.
4. Порівнюємо способи генерації сторінок: CSR, SSR, SSG, ISR. Гайд на основі стеку React [Електронний ресурс] – Режим доступу до ресурсу: https://dou.ua/forums/topic/41585/. Дата доступу: жовтень 2023.
5. Tree shaking [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking. Дата доступу: жовтень 2023.
6. The State of Developer Ecosystem 2022 [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/lp/devecosystem-2022/. Дата доступу: жовтень 2023.
7. Front-end Frameworks [Електронний ресурс] – Режим доступу до ресурсу: https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/. Дата доступу: жовтень 2023.
8. 2023 Developer Survey [Електронний ресурс] – Режим доступу до ресурсу: https://survey.stackoverflow.co/2023/. Дата доступу: жовтень 2023.
9. Levlin M. DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte : дис. канд. техн. наук : 122 / Levlin Mattias – Turku, 2020. – 95 с.
10. React vs Vue vs Angular vs Svelte [Електронний ресурс] – Режим доступу до ресурсу: https://dev.to/hb/react-vs-vue-vs-angular-vs-svelte-1fdm. Дата доступу: жовтень 2023.
11. The top React UI libraries and kits in 2023 [Електронний ресурс] – Режим доступу до ресурсу: https://blog.logrocket.com/top-react-ui-libraries-kits/. Дата доступу: жовтень 2023.
12. Pick the right UI Library for your Vue 3 or Nuxt 3 Project. [Електронний ресурс] – Режим доступу до ресурсу: https://ui-libs.vercel.app/. Дата доступу: жовтень 2023.

13. Macrae C. Vue.js: Up and Running / Callum Macrae. – Sebastopol: O\'Reilly, 2018. – (O\'Reilly Media, Inc.).
14. Template Syntax [Електронний ресурс] – Режим доступу до ресурсу: https://vuejs.org/guide/essentials/template-syntax.html. Дата доступу: жовтень 2023.
15. Render Functions & JSX [Електронний ресурс] – Режим доступу до ресурсу: https://vuejs.org/guide/extras/render-function.html. Дата доступу: жовтень 2023.
16. Class and Style Bindingshttps [Електронний ресурс] – Режим доступу до ресурсу: https://vuejs.org/guide/essentials/class-and-style.html. Дата доступу: жовтень 2023.
17. Composition API FAQ [Електронний ресурс] – Режим доступу до ресурсу: https://vuejs.org/guide/extras/composition-api-faq.html. Дата доступу: жовтень 2023.
18. What Is A JavaScript Bundler? [Електронний ресурс] – Режим доступу до ресурсу: https://www.codejourney.net/what-is-a-javascript-bundler/. Дата доступу: жовтень 2023.
19. Sinisalo A. Web frontend component quality model : дис. канд. техн. наук : 122 / Aleksi Sinisalo – Turku, 2017. – 83 с.
20. Web Components [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/Web_components. Дата доступу: жовтень 2023.
21. Using shadow DOM [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM. Дата доступу: жовтень 2023.
22. What is end-to-end testing? [Електронний ресурс] – Режим доступу до ресурсу: https://circleci.com/blog/what-is-end-to-end-testing/. Дата доступу: жовтень 2023.
23. Automatic documentation and Storybook [Електронний ресурс] – Режим доступу до ресурсу: https://storybook.js.org/docs/react/writing-docs/autodocs. Дата доступу: жовтень 2023.