ЛЬВІВСЬКИЙ ТОРГОВЕЛЬНО-ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ МІЖНАРОДНИХ ЕКОНОМІЧНИХ ВІДНОСИН ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра комп'ютерних наук, прикладної та вищої математики

КВАЛІФІКАЦІЙНА РОБОТА

на тему: **Проектування та реалізація бібліотеки компонентів для створення UI компонентів з використанням препроцесорів**

Виконав: здобувач першого (бакалаврського) рівня вищої освіти ОПП "Комп'ютерні науки" спеціальності 122 "Комп'ютерні науки"

_____ **Андруневчин Софія Миколаївна**
(підпис)

Науковий керівник к.т.н., доц.
_____ **Олійник Р.В.**
(підпис)

Львів – 2025

ЛЬВІВСЬКИЙ ТОРГОВЕЛЬНО-ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ

Факультет міжнародних економічних відносин та інформаційних технологій

Кафедра комп'ютерних наук, прикладної та вищої математики

Перший (бакалаврський) рівень вищої освіти

Спеціальність 122 "Комп'ютерні науки"

Освітньо-професійна програма "Комп'ютерні науки"

“ЗАТВЕРДЖУЮ”

Завідувач кафедри комп'ютерних наук, прикладної та вищої математики

_____ З. Г. Новосад

28.10. 2024 р.

Завдання

до кваліфікаційної роботи

Андруневчин Софія Миколаївна

(прізвище, ім’я, по батькові)

1.Тема роботи Проектування та реалізація бібліотеки компонентів для створення UI компонентів з використанням препроцесорів

керівник роботи _к.т.н., доц. Олійник Роман Володимирович_____

затверджена наказом по Університету №1041/03 від "28" жовтня 2024 р.

2.Термін здачі здобувачем вищої освіти роботи 5 червня 2025 р.

3. Вхідні дані до роботи література, документація, матеріали та сайти Internet, що пов’язані з розробкою спеціалізованих бібліотек UI компонентів з використанням препроцесорів та систем task runners.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які підлягають розробці)

1) Дослідження засобів організації UI; 2) Проектування бібліотеки ui-компонентів з використанням препроцесорів; 3) Програмна реалізація бібліотеки ui компонент.

5. Перелік графічного матеріалу

_____

6. Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання 28.10. 2024 р.

Календарний план

№	Найменування етапів виконання кваліфікаційної роботи	Терміни виконання етапів	Примітка
	Отримання завдання і аналіз інформаційних джерел за темою роботи	до 11.12.2024	виконано
	Дослідження	до 13.01.2025	виконано
	Аналіз	до 07.02.2025	виконано
	Проектування структури генератора UI компонентів	до 01.04.2025	виконано
	Проектування GULP, реалізація генератора бібліотеки UI компонентів	до 15.04.2025	виконано
	Тестування генератора бібліотеки компонент	до 22.04.2025	виконано
	Оформлення кваліфікаційної роботи. Подання роботи на перевірку керівнику	до 16.05.2025	виконано
	Доопрацювання кваліфікаційної роботи з врахуванням зауважень керівника. Подання роботи для перевірки на рівень запозичень	до 20.05.2025	виконано
	Подання кваліфікаційної роботи керівнику для отримання відгуку	до 27.05.2025	виконано
	Представлення кваліфікаційної роботи на зовнішнє рецензування. Подання кваліфікаційної роботи на кафедру комп'ютерних наук	до 5.06.2025	виконано
	Попередній захист кваліфікаційної роботи	05.06.2025-06.06.2025	виконано
	Захист кваліфікаційної роботи на засіданні екзаменаційної комісії	згідно графіку роботи ЕК	

Здобувач вищої освіти _____ Андруневчин С.М.

Керівник роботи _____ к.т.н., Олійник Р.В.

РЕФЕРАТ

Робота складається із вступу, трьох розділів, висновків, списку використаних джерел з посилань та додатків. Основний зміст роботи викладено на **61 сторінці** і містить **19 рисунків** та **3 таблиці**.

Об'єктом дослідження є процес створення користувацького інтерфейсу з використанням бібліотеки компонентів.

Предметом дослідження є технології і методи розробки UI-компонентів, їх проектування та оптимізація.

Метою дослідження є розробка бібліотеки UI-компонентів з використанням сучасних технологій та автоматизації процесів.

В першому розділі проведено аналіз поняття користувацького інтерфейсу (UI), компонентного підходу до його розробки та огляд технологій створення компонентів. **В другому розділі** запропоновано проектування бібліотеки компонентів, визначено її архітектуру та особливості. **В третьому розділі** описано програмну реалізацію бібліотеки, організацію файлів і приклади використання.

Ключові слова: UI, UX, компонентний підхід, SCSS, бібліотека UI-компонентів, автоматизація, Gulp, адаптивність.

ABSTRACT

The work consists of an introduction, three chapters, conclusions, a list of references with links, and appendices. The main content of the work is presented on **61 pages** and includes **19 figures** and **3 tables**.

The object of the research is the process of creating a user interface using a component library.

The subject of the research is the technologies and methods for developing UI components, their design, and optimization.

The aim of the research is to develop a library of UI components using modern technologies and process automation.

The first section analyzes the concept of a user interface (UI), the component approach to its development, and a review of component creation technologies. **The second section** proposes the design of the component library, its architecture, and features. **The third section** describes the program implementation of the library, file organization, and examples of usage.

Keywords: UI, UX, component-based approach, SCSS, UI component library, automation, Gulp, adaptability.

ЗМІСТ

ВСТУП 8

1 ДОСЛІДЖЕННЯ ЗАСОБІВ ОРГАНІЗАЦІЇ UI 10

1.1 Поняття користувацького інтерфейсу (UI) 10

1.2. Дослідження компонентного підходу до розробки інтерфейсу 14

1.3 Огляд технологій для створення UI-компонентів 18

2 ПРОЕКТУВАННЯ БІБЛІОТЕКИ UI-КОМПОНЕНТІВ З ВИКОРИСТАННЯМ ПРЕПРОЦЕСОРІВ 22

2.1 Аналіз особливостей інтерфейсу користувача 22

2.2 Постановка задачі та вимоги до бібліотеки компонентів 25

2.3. Архітектура бібліотеки UI-компонентів 28

3 ПРОГРАМНА РЕАЛІЗАЦІЯ БІБЛІОТЕКИ UI КОМПОНЕНТ 33

3.1 Вибір інструментального програмного забезпечення 33

3.2 Розробка компонентів бібліотеки 37

3.3 Приклад використання бібліотеки UI компонент 46

ВИСНОВКИ 52

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 54

ДОДАТКИ 56

Додаток А Фрагмент Конфігурації Gulp 56

Додаток Б Фрагмент коду Gulp 57

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

UI	User Interface (користувацький інтерфейс)
UX	User Experience (досвід взаємодії користувача)
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
JS	JavaScript
SCSS	Sassy CSS (препроцесор для CSS)
JSON	JavaScript Object Notation
PNG	Portable Network Graphics
SVG	Scalable Vector Graphics
GIF	Graphics Interchange Format
SEO	Search Engine Optimization
IDE	Integrated Development Environment
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol

ВСТУП

У світі, де цифрові технології є невід'ємною частиною життя, вимоги до UI/UX-дизайну постійно зростають, особливо в контексті адаптивності та швидкості розробки. Стрімкий розвиток інтернет-технологій диктує нові стандарти, які спрямовані на покращення інтерактивності, доступності та інтуїтивності інтерфейсів. Варто зазначити, що в умовах глобалізації ринку та зростаючої конкуренції між компаніями, які надають цифрові послуги, користувацький досвід (UX) і користувацький інтерфейс (UI) стали важливими факторами успіху будь-якого продукту. Окрім того, вимоги до адаптивного дизайну обумовлюють необхідність створення інтерфейсів, які бездоганно працюють на різних пристроях — від мобільних телефонів до великих дисплеїв не вимагаючи при цьому великої кількості ресурсів.

Бібліотеки UI-компонентів, що базуються на сучасних технологіях, таких як препроцесори SCSS [1], забезпечують можливість стандартизації, повторного використання компонентів та автоматизації процесів розробки. Це особливо важливо в умовах динамічних змін у потребах користувачів та вимогах до якості програмного забезпечення. У свою чергу готові широковикористовувані фреймворки, що постачають набори UI елементів не завжди є достатньо гнучкими для стилізації та можуть потребувати велику кількість додаткових елементів для роботи. Відтак, такі інструменти для автоматизації запуску задач та збірки проектів як Gulp [2], допомагають розробникам швидше виконувати рутинні задачі та зосереджувати увагу на написанні якісного оптимізованого коду кожного компонента бібліотеки UI. Таким чином, дослідження та впровадження компонентного підходу до розробки бібліотек UI-компонентів не тільки відповідає потребам сучасної індустрії, але й відкриває нові можливості для оптимізації роботи розробників та підвищення якості цифрових продуктів.

Об’єктом дослідження є процес проектування і реалізації користувацького інтерфейсу з використанням компонентного підходу.

Предметом дослідження можна вважати методологію створення бібліотек UI-компонентів із застосуванням сучасних технологій, таких як препроцесори SCSS і task runners, зокрема GULP.

Мета кваліфікаційної роботи - Розробка бібліотеки UI-компонентів, яка забезпечує гнучкість, адаптивність та автоматизацію створення користувацьких інтерфейсів.

Завдання

1. Провести аналіз особливостей проектування UI/UX.
2. Дослідити компонентний підхід до розробки інтерфейсів.
3. Описати сучасні технології, що застосовуються для створення UI-компонентів.
4. Спроектувати архітектуру бібліотеки компонентів.
5. Реалізувати програмну частину бібліотеки із застосуванням SCSS та Gulp.
6. Випробувати бібліотеку в умовах реального проекту.

Теоретичне значення Дослідження сприяє формуванню сучасних підходів до проектування інтерфейсів, впроваджуючи інновації у сфері UI/UX-дизайну.

Практичне значення Розроблена бібліотека UI-компонентів може використовуватися для створення веб-додатків, що відповідають сучасним вимогам зручності й адаптивності, знижуючи витрати часу на розробку.

Структура роботи Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. У першому розділі досліджено концепцію UI/UX і компонентного підходу. У другому розділі розроблено архітектуру бібліотеки компонентів. У третьому розділі представлено програмну реалізацію бібліотеки та приклад її використання.

1 ДОСЛІДЖЕННЯ ЗАСОБІВ ОРГАНІЗАЦІЇ UI

Розвиток інформаційних технологій, а також зростання вимог до проектованого UI/UX зумовили появу нових підходів до проектування призначеного для користувача інтерфейсу.

1.1 Поняття користувацького інтерфейсу (UI)

Користувацький інтерфейс (UI, від англ. User Interface) — це сукупність засобів і рішень, які забезпечують взаємодію користувача з програмним забезпеченням, зокрема веб-додатками [3]. До складу UI входять різні графічні елементи (кнопки, поля вводу, меню, вкладки, тощо), шрифти, кольори, а також логіка їх розміщення та поведінки у взаємодії з користувачем (рисунок 1.1).



Рисунок 1.1 - Інтерфейс користувача веб сайту ЛТЕУ

Головне призначення UI — забезпечити зручну, ефективну та інтуїтивно зрозумілу взаємодію користувача з функціоналом додатку. Вдалий інтерфейс підвищує користувацький досвід (UX), полегшує навчання роботі з додатком, знижує кількість помилок та сприяє утриманню аудиторії [4].

Розробка користувальницьких інтерфейсів будується таким чином, щоб створити його максимально привабливим і зручним для оптимізації його взаємодії

з користувачем. Тому перед розробниками веб-інтерфейсів в будь-якому проекті ставиться завдання створення саме дружнього по відношенню до користувача інтерфейсу. Однак це не просте завдання, що вимагає певного досвіду проектування. Головні вимоги у будь-якому інтерфейсі – це зручність, практичність і інтуїтивна зрозумілість. Саме тому проектування інтерфейсу користувача умовно можна поділити на дві течії, а саме UX design і UI design, що об'єднуючись формують цілісний інтерфейс користувача User interface [4].

User Experience Design (UX) в перекладі означає «досвід взаємодії» і включає в себе різні UX-компоненти: інформаційну архітектуру, проектування взаємодії, графічний дизайн і контент. В цілому, головною парадигмою UX дизайну є комплексний підхід до взаємодії користувача з інтерфейсом. Тому при проектуванні UX розробник інтерфейсу повинен по можливості максимально врахувати всі дрібниці, починаючи від середовища користувача і типу електронного пристрою і закінчуючи способами введення і відображення інформації. (рисунок 1.2).

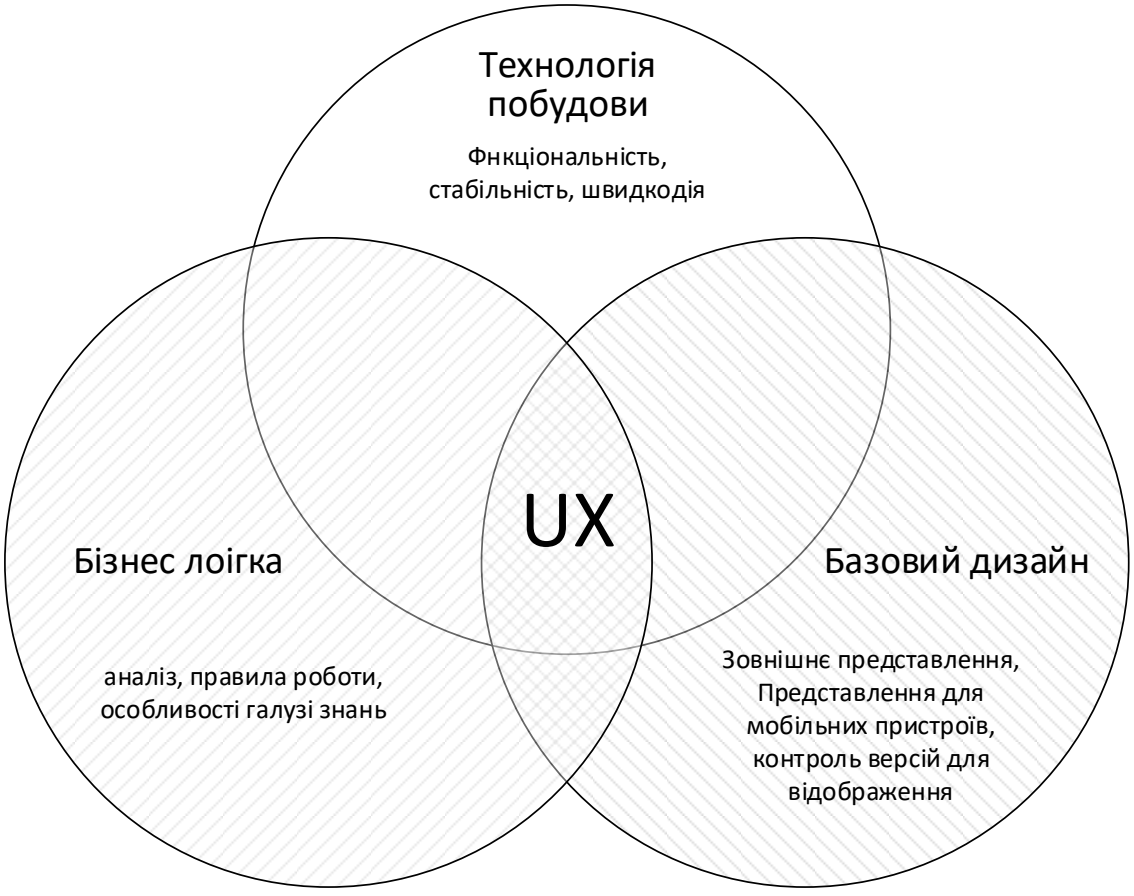

Рисунок 1.2 - Розподіл компонентів побудови UX-інтерфейсу користувача

UX, або досвід взаємодії передбачає поведінку людини, його ставлення, і емоції викликані і пов'язані з використанням продукту, системи або послуги. Досвід взаємодії включає в себе практичні, засновані на досвіді, ефективні, значимі і цінні аспекти взаємодії людини з комп'ютером або продуктом [5]. Крім того, UX містить у собі сприйняття користувачем характеристик системи, таких як корисність, простота використання і ефективність. Досвід взаємодії носить суб'єктивний характер, адже йдеться про індивідуальному сприйнятті і оцінці системи. UX - поняття непостійне, адже його параметри постійно змінюються разом з системами, обставинами, технологіями і т. д. Розробка UX починається з визначення базового контингенту і створення цільового «користувача» та його характеристик. Ґрунтуючись на цих знаннях вводяться вимоги до проекту, що розробляється. Після чого розробляється інформаційна архітектура і проектується ієрархія вмісту, що допомагає на етапі прототипування. Далі вибирається найбільш оптимальний метод прототипування. Він повинен бути достатньо економічним, проте і ефективним настільки, щоб була можливість збору зворотного зв'язку у швидкій і легкій формі [5]. Як показав проведений аналіз прототипи умовно модна поділити на горизонтальні, де реалізується широкий набір функцій, але невелика глибина їх функціональності); Вертикальні, що реалізують глибоку функціональність, але вузький набір функцій; Т- прототипи, де більша частина конструкції робиться на поверхневому рівні, але деякі функції реалізовані на більшій глибині; Місцеві прототипи - використовуються, щоб придумати альтернативу дизайну конкретної моделі взаємодії. Після вибору типу прототипу починається процес інтеграції UX у розроблювані UI компоненти. Такими компонентами можуть бути як атомарні сутності – кнопки, поля вводу так і цілі форми та комплексні об’єкти розроблюваного UI [3].

Основною метою при розробці інтерфейсу для взаємодії людини з машиною є створення тісно інтегрованого UX + UI у єдиний інтерфейс, що складається з різних компонентів головною ціллю якого є забезпечення легкого (інтуїтивного), ефективного отримання результату під час роботи з машиною.

Ідеальним результатом вважається той, при якому користувач виробляє мінімальні зусилля для введення, отримуючи від машини бажаний висновок, при цьому машина мінімізує висновок зайвих даних. Таким чином UI і UX – це різні поняття різні поняття (рисунок 1.3.).

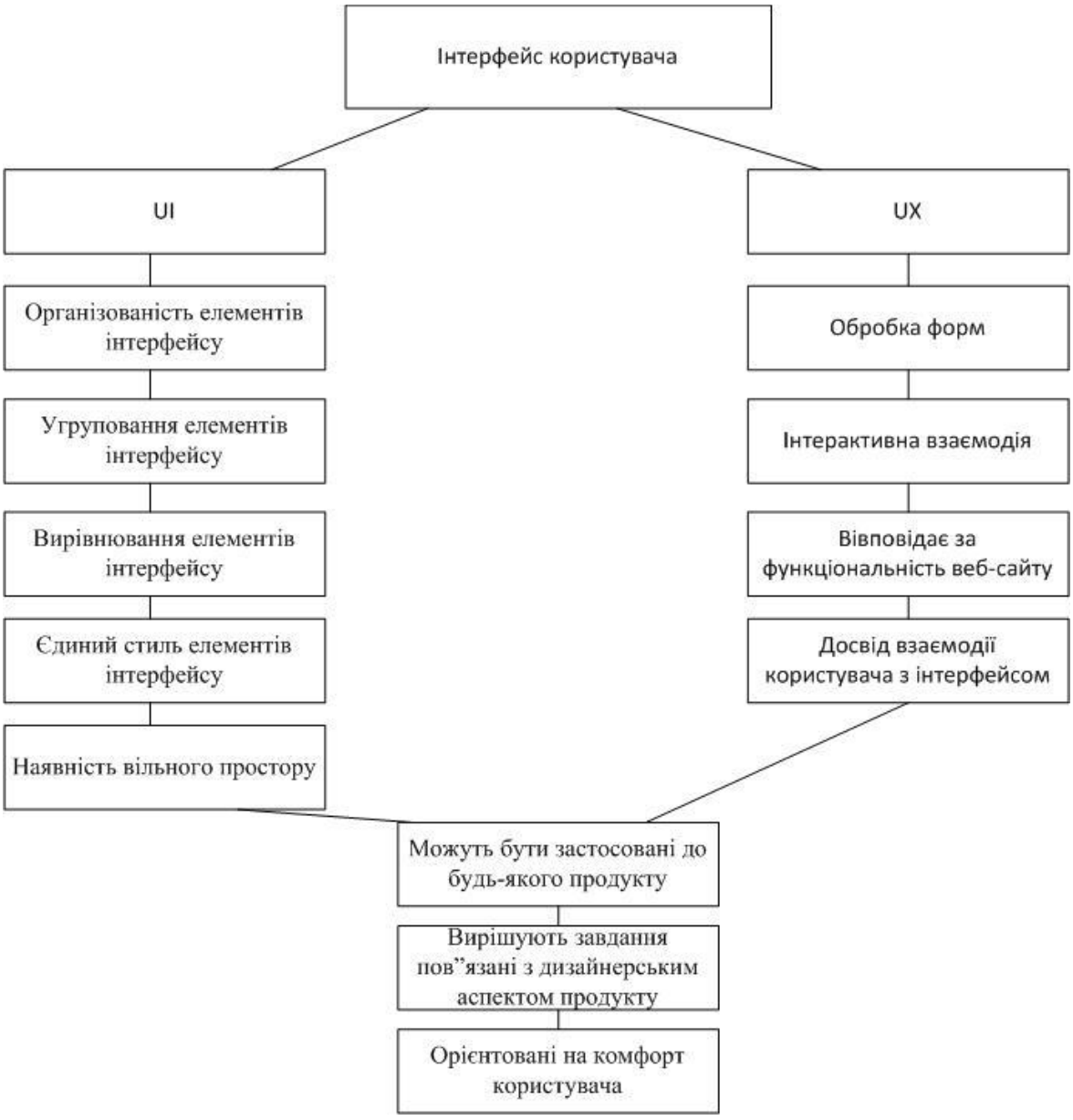

Рисунок 1.3 - Розподіл інтерфейсу на UI та UX

Однак вони повинні працювати в парі, зацікавити користувача у продукті та максимально швидко й якісно передати затребувану інформацію. До переваг правильного використання UI/UX можна віднести те, що UX може існувати і працювати дуже ефективно з досить простим інтерфейсом користувача [5]. Приміром, можна зустріти додаток з гарним дизайном, який складно

використовувати (хороший UI, поганий UX). Також бувають додатки, які виглядають непоказно, але виявляються дуже зручними у використанні (поганий UI, хороший UX). Відповідно поняття UI та UX є взаємопов'язані, а якісний веб інтерфейс на сьогодні не може бути утворений або з використанням UX або UI.

1.2. Дослідження компонентного підходу до розробки інтерфейсу

У сучасній розробці веб-додатків усе більшої популярності набуває компонентний підхід до побудови інтерфейсів користувача (UI) [6]. Цей підхід базується на принципі модульності, що передбачає створення інтерфейсу з набору незалежних та ізольованих компонентів. Така методологія значно підвищує ефективність розробки, покращує масштабованість проєкту та полегшує подальшу підтримку й оновлення. Відтак компонентним підходом до розробки інтерфейсів можна вважати процес організації коду, при якому інтерфейс веб-додатку розбивається на окремі, незалежні, багаторазово використовувані модулі (компоненти) де кожен компонент інкапсулює власну структуру, стилі та поведінку, що забезпечує гнучкість, масштабованість і зручність підтримки коду (рисунок 1.4).

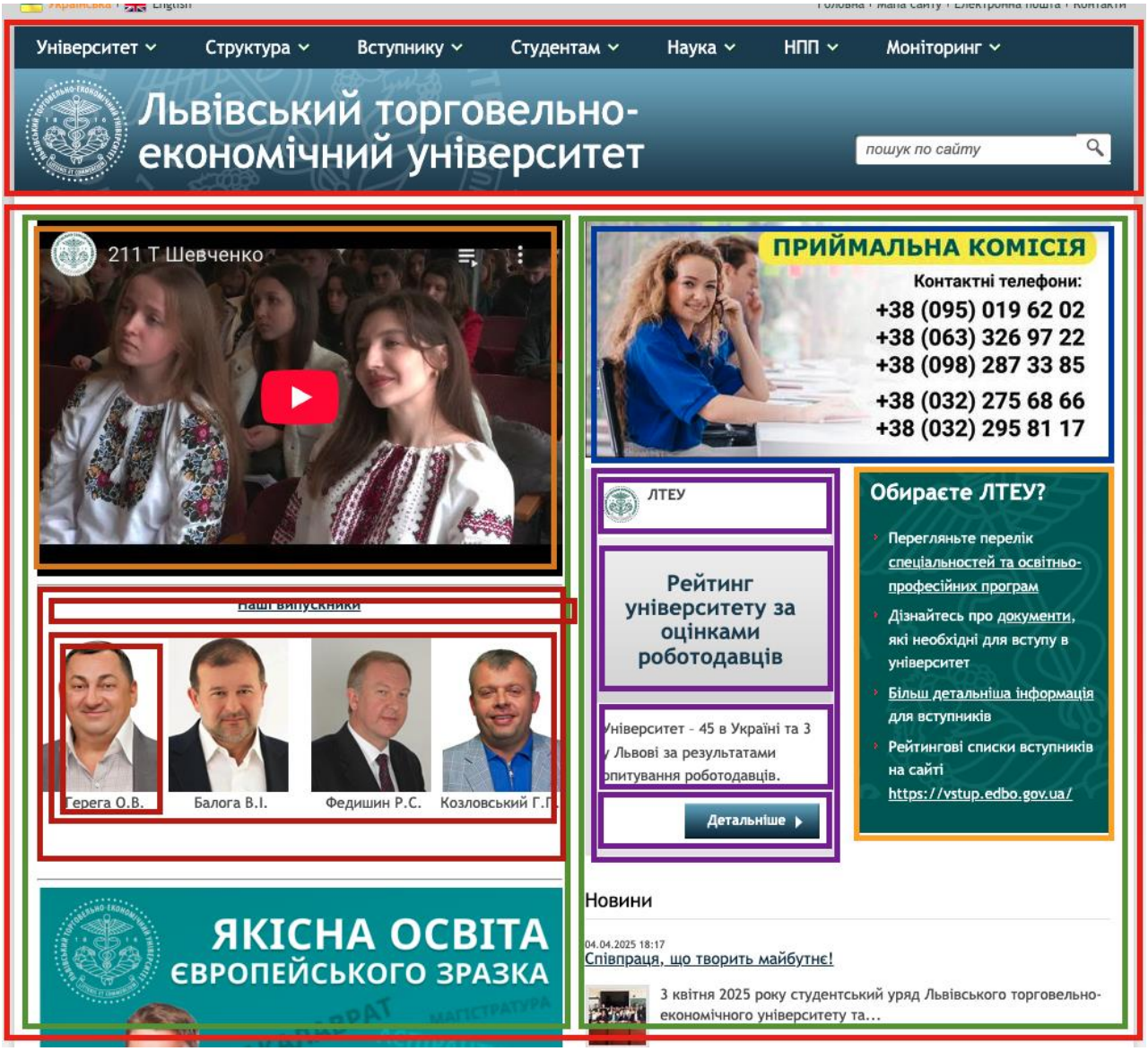

Рисунок 1.4 - Часткова схема компонент веб-сайту ЛТЕУ

У розділеному багатокомпонентною UI компонентом можна вважати самостійний фрагмент інтерфейсу, який поєднує у собі структуру (HTML), стилі (CSS або препроцесори на кшталт SCSS/LESS) та логіку (JavaScript) [8]. Наприклад, кнопка, модальне вікно, форма або навігаційна панель. У свою чергу компоненти поділяються на прості, такі як кнопка з заданими стилями та складені як-от форма авторизації, яка включає кілька полів вводу, кнопку та повідомлення про помилки (рисунок 1.5).

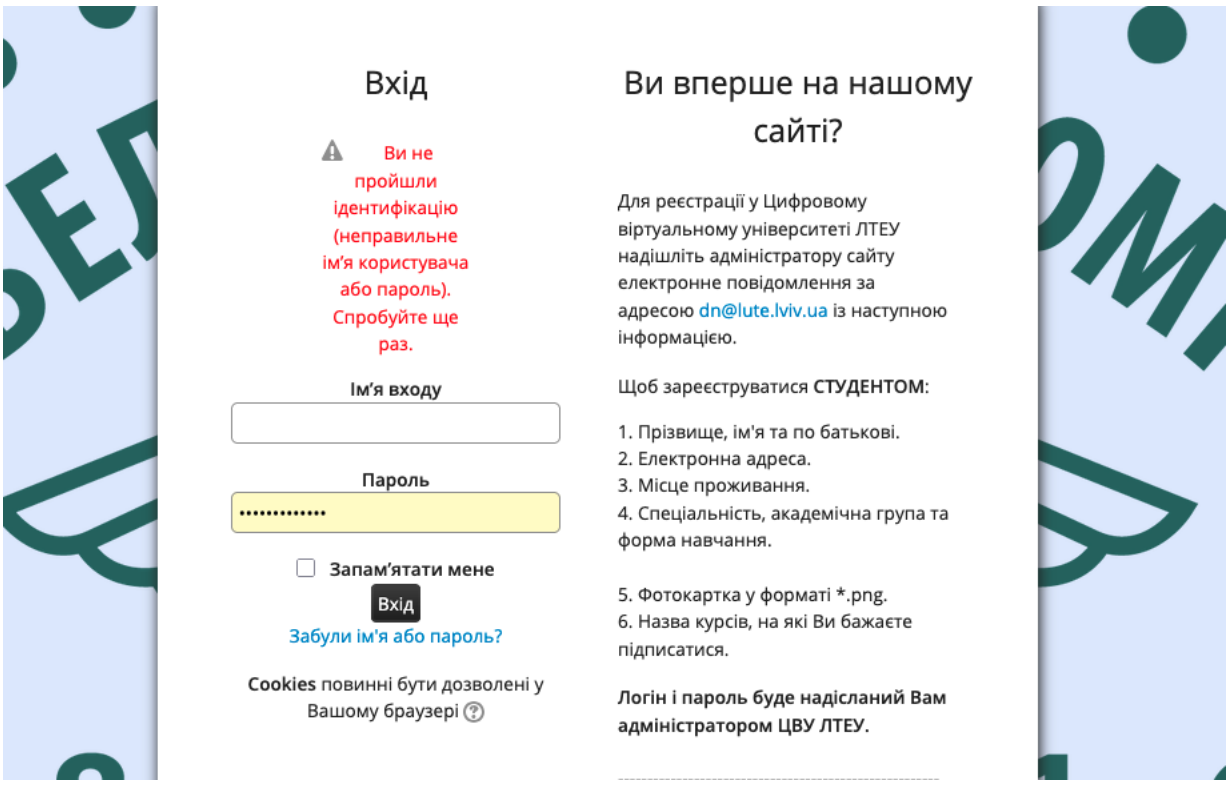

Рисунок 1.5 - Компонент – форма авторизації віртуального університету ЛТЕУ

Компонентний підхід до розробки інтерфейсів базується на тих самих принципах, що і об’єктно-орієнтоване програмування, оскільки кожен компонент фактично являє собою окремий об’єкт з його внутрішніми властивостями та функціоналом. Таким чином тут інкапсуляція є одним із фундаментальних принципів, на якому тримається вся ідея компонентного підходу розробки UI. Суть цього принципу полягає в тому, що всі елементи, пов’язані з функціонуванням компонента — структура (HTML), стилі (CSS або препроцесори) та поведінка (JavaScript) — зберігаються виключно в його межах, що забезпечує ізольованість компонента від інших частин інтерфейсу та запобігає виникненню конфліктів між стилями або логікою різних елементів [6]. Наприклад, стилі кнопки, яка належить до модального вікна, не вплинуть на кнопки, що використовуються в інших частинах сайту. Інкапслюювавши всі елементи UI компоненти можливе повторне використання одного і того самого компоненту у різних частинах додатку або навіть у різних проєктах без необхідності дублювання коду. Наприклад, якщо у проєкті створено компонент сповіщення про помилку, його можна використовувати як у формі авторизації,

так і у формі реєстрації без написання нового коду. Це значно економить час розробників та зменшує ризики помилок при копіюванні коду.

Композиція полягає в тому, що складні інтерфейсні елементи формуються шляхом об’єднання простіших компонентів. Наприклад, компонент «Картка товару» може складатися з кількох підкомпонентів: зображення, заголовка, опису, кнопки «Купити» тощо. Такий підхід дозволяє легко модифікувати окремі частини складного компонента без необхідності переписувати всю його структуру [4].

Заміщуваність означає, що будь-який компонент можна легко замінити на інший з аналогічною функціональністю без впливу на загальну архітектуру застосунку. Наприклад, у разі зміни дизайну кнопки або зміни логіки її поведінки, достатньо оновити лише відповідний компонент, не змінюючи весь інтерфейс, а чітке розмежування відповідальностей гарантує, що кожен компонент виконує лише одну конкретну функцію. Це дозволяє уникати надмірної складності та полегшує підтримку коду. Наприклад, компонент «Форма введення пошти» повинен відповідати лише за правильне відображення поля вводу та перевірку коректності електронної адреси, не займаючись відправкою даних чи обробкою відповіді сервера.

Використання компонентного підходу в розробці інтерфейсів має низку переваг, які є вирішальними при створенні великих та складних веб-додатків [4]. Так, проєкти, побудовані на компонентній архітектурі, легко розширюються. Якщо з часом виникає потреба в додаткових функціях або нових елементах інтерфейсу, їх можна реалізувати у вигляді нових компонентів або розширити вже наявні, не порушуючи існуючої логіки та структури. Також компонентна структура робить архітектуру UI більш прозорою та зрозумілою, де кожен елемент представлений окремо, що дозволяє швидко орієнтуватися в коді. Це особливо корисно для нових членів команди, які можуть швидше ознайомитися з проєктом. Завдяки ізольованості компонентів, різні учасники команди можуть одночасно працювати над окремими частинами UI, не заважаючи одне одному. Наприклад, один розробник може створювати UI компонент меню, тоді як інший

працює над формою зворотного зв’язку. Оскільки компоненти незалежні, їх легко тестувати окремо від інших частин застосунку. Це підвищує якість програмного продукту, дає змогу вчасно виявити помилки та забезпечити стабільну роботу системи. Найчастіше для цього використовуються юніт-тести, які перевіряють логіку роботи окремого компонента. Стандартизований підхід до розробки дозволяє використовувати вже перевірені рішення, що зменшує кількість помилок та пришвидшує розробку. Компоненти, які були протестовані та використані раніше, мають менше шансів викликати неочікувані проблеми в нових проєктах. Завдяки повторному використанню компонентів зменшується дублювання коду.

1.3 Огляд технологій для створення UI-компонентів

У сучасній розробці UI-компоненти організовуються у вигляді бібліотек або систем дизайну. Це дозволяє стандартизувати процес створення інтерфейсів, зменшити час на їх розробку і забезпечити високу якість продукту. Сьогодні відомо кілька стратегій створення UI-компонентів для побудови інтерфейсів - використання сторонніх UI-бібліотек, таких як Bootstrap, Material UI, або ж створення власної бібліотеки з нуля — із застосуванням CSS-препроцесорів (наприклад, SCSS) або чистого CSS (рисунок 1.6) [8].

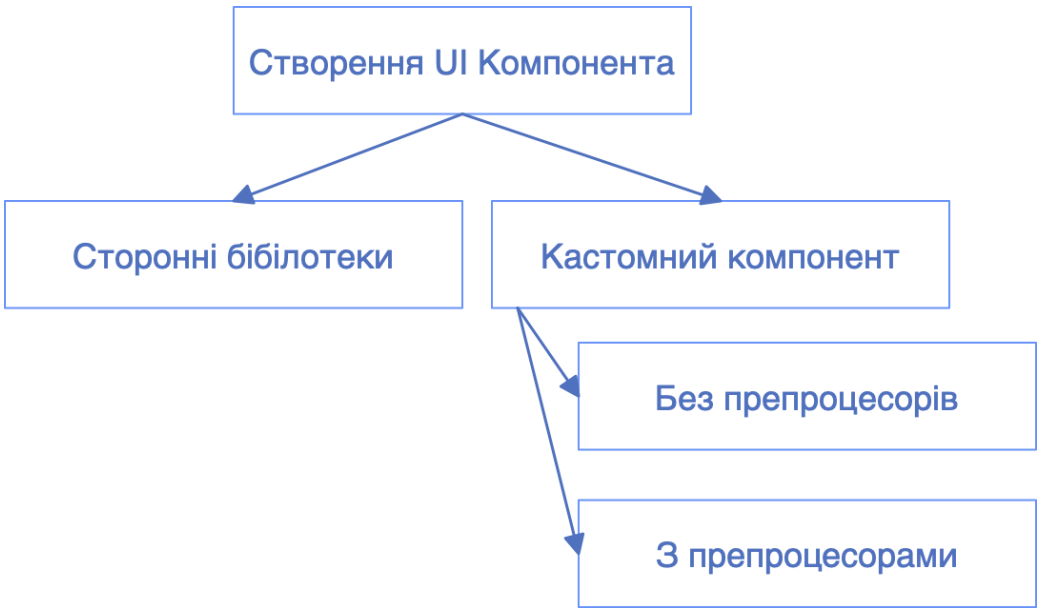

Рисунок 1.6 - Види створення UI компонентів

Сторонні UI-бібліотеки, особливо такі як Bootstrap надають велику кількість готових компонентів — кнопок, форм, модальних вікон, сітки — які легко адаптуються до різних розмірів екрану (рисунок 1.7.).



Рисунок 1.7 - Порівняльна таблиця актуальних фреймворків

До переваг можна віднести те, що такі компоненти вже протестовані, відповідають базовим стандартам доступності (accessibility), адаптивні й мають естетичний вигляд “із коробки”, а при використання веб-фреймворків інтеграція сторонніх бібліотек UI компонентів можлива двома шляхами:

- Як зовнішня CSS-бібліотека, підключена до шаблонів компонента (наприклад, у Angular або React).
- Через специфічні обгортки, такі як react-bootstrap або ngx-bootstrap, які реалізують Bootstrap-компоненти у вигляді нативних елементів фреймворку.

Хоча використання готових бібліотек, що розробляються великою групою програмістів та вирішують суттєвий набір завдань і є простим, проте у них є й недоліки. Так як уніфіковані бібліотеки повинні бути стійкими та однаково виглядати в різних середовищах, то можливість їх кастомізації є обмеженою,

тому зовнішній вигляд часто неможливо привести у відповідність до вимог дизайну продукту, оскільки тут обмежена гнучкість у змінах поведінки або стилів компонентів. Також важливим моментом є те, що часто використання таких бібліотек вимагає встановлення додаткових рідко або навіть повністю невикористовуваних залежностей, які збільшують розмір кодової бази, а часом призводять до конфліктів.

Якщо ж розглядати альтернативу стороннім бібліотекам, то тут можливо лише використовувати кастомні компоненти, які об’єднуються у внутрішню бібліотеку. Так, створення UI-компонентів вручну із застосуванням HTML, CSS та JavaScript без використання препроцесорів надає максимальний контроль над стилем і структурою компонентів. Цей підхід зазвичай обирається, коли потрібно створити унікальний, детально кастомізований UI та UX, який не можна досягти за допомогою стандартних бібліотек чи фреймворків. У цьому випадку розробник працює безпосередньо з кожним елементом інтерфейсу, що дає можливість налаштувати всі особливості вигляду та поведінки. Також при ручній розробці компонентів важливо приділяти увагу правильному іменуванню класів, щоб уникнути можливих конфліктів стилів, а також забезпечити адаптивність для різних розмірів екранів. Таким чином основною перевагою цього підходу є те, що розробник має повну свободу в реалізації стилів та структури компонентів, а відсутність залежностей від сторонніх бібліотек дає можливість створювати інтерфейси, які точно відповідають вимогам проекту. Проте варто зазначити, що такий підхід має й недоліки. Процес розробки може бути значно повільнішим, оскільки кожен елемент інтерфейсу потрібно стилізувати вручну, велику складність становить підтримка такого підходу в масштабних проєктах, де кожен компонент може вимагати окремої уваги. Крім того, немає можливості повторно використовувати логіку стилів без дублювання, що ускладнює обслуговування великих систем.

Створення UI компонентів з використанням CSS-препроцесорів, таких як SASS чи SCSS, надає значно більше можливостей для організації стилів та їх масштабування [9]. Препроцесори дозволяють створювати змінні, функції,

міксини, вкладені стилі та використовувати цикли і умови, що значно підвищує гнучкість розробки. Це особливо корисно при створенні великих або складних інтерфейсів, де необхідно підтримувати багато стилів, які можуть змінюватися залежно від контексту. Тут основною перевагою є покращене структурування коду. Замість того, щоб дублювати стилі, можна створювати модульні компоненти, що повторно використовуються. Крім того, такі компоненти легко підтримуються і змінюються в міру потреби. Проте цей підхід вимагає певної підготовки середовища для налаштування препроцесорів, особливо в React, де потрібно налаштувати Webpack або використовувати інші інструменти для обробки SCSS.

Проведене дослідження (таблиця 1.1), використання CSS-препроцесорів в поєднанні з фронтенд-фреймворками дозволяє розробляти більш організовані, масштабовані та легко підтримувані інтерфейси.

Також варто зауважити що у довгостроковій перспективі, особливо для проєктів з унікальним дизайном і великою кількістю кастомних компонентів, доцільно обирати **власну реалізацію з використанням SCSS**. Це дозволяє зберігати **узгодженість стилів**, підтримувати **чисту архітектуру** UI та забезпечити **глибоку інтеграцію** з обраним фреймворком.

Таблиця 1.1 - Порівняння характеристика CSS-препроцесоірв у фреймворках

Критерій	**UI-бібліотеки (Bootstrap, MUI)**	**Власна реалізація (без препроцесорів)**	**Власна реалізація (з препроцесорами)**
Інтеграція з фреймворками	Швидка, але залежить від обгорток	Гнучка, потребує ручної адаптації	Гнучка, потребує налаштування
Гнучкість кастомізації	Обмежена	Висока	Висока
Швидкість створення	Дуже висока	Низька	Середня
Масштабованість	Середня	Низька	Висока
Унікальність дизайну	Слабка	Повна	Повна
Підтримка та повторне використання	Часткова	Обмежене	Добре реалізоване
Вимоги до навичок	Низькі	Середні	Середні / високі
Продуктивність	Висока	Висока	Висока

2 ПРОЕКТУВАННЯ БІБЛІОТЕКИ UI-КОМПОНЕНТІВ З ВИКОРИСТАННЯМ ПРЕПРОЦЕСОРІВ

Розвиток інформаційних технологій, а також зростання вимог до проектованого UI/UX зумовили появу нових підходів до проектування призначеного для користувача інтерфейсу.

2.1 Аналіз особливостей інтерфейсу користувача

При проектуванні функціональних компонентів будь-якого інтерфейсу користувача спочатку необхідно здійснити дослідження складових частин проектованого інтерфейсу з метою встановлення типових функціональних блоків, які будуть спільними не тільки для конкретного середовища, а й будуть спільними для інших елементів UI розподіливши усі компоненти на різні логічні рівні функціонування (рисунок 2.1) [1,4,3].

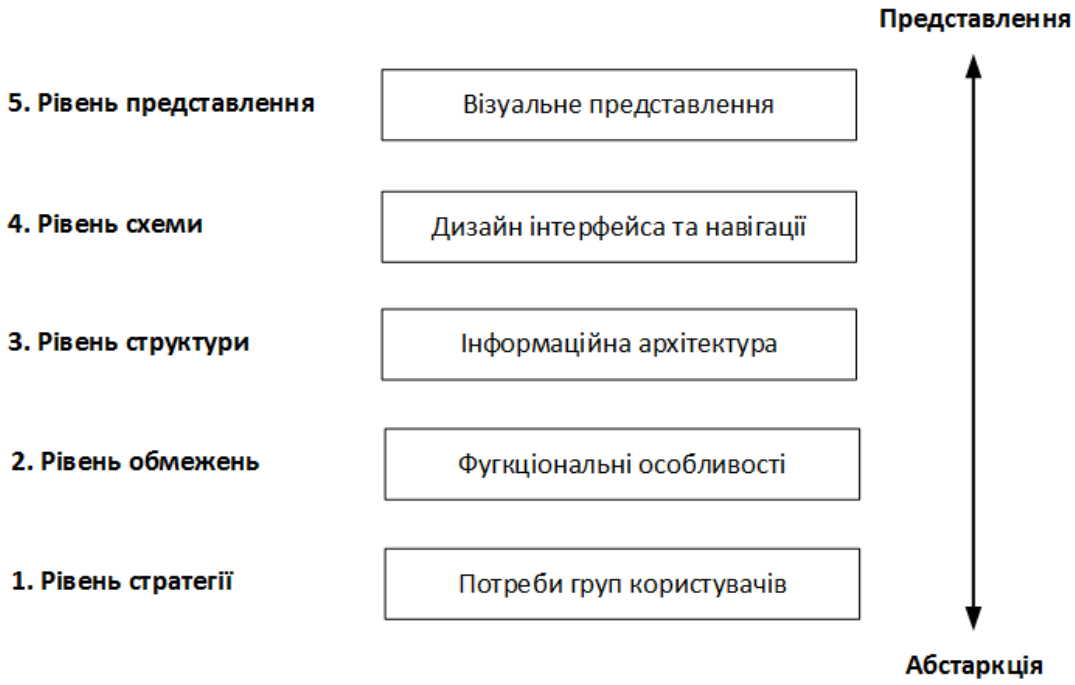

Рисунок 2.1 - Рівні функціональних компонентів

Рівень стратегії є абстрактним, тобто таким, що лише визначає які потреби взаємодії з користувачами проектованого інтерфейсу. Якщо такі потреби є лише

інформаційні, то взаємодія як така не передбачається і можна здійснювати лише видачу контенту, наприклад елементи сітки. В іншому випадку потрібно переходити на наступний рівень. Тут рівень обмежень передбачає визначення засобів які необхідні для взаємодії з інформаційним наповненням. Сьогодні такими засобами є елементи веб-форм, що слугують як засіб вводу/виводу інформаційного наповнення за допомогою інтерфейсу користувача, наприклад, здійснення вводу логіна та пароля здійснюється шляхом заповнення у формі відповідних даних, після чого користувач може переглянути отримати доступ до приватних елементів веб-сайту.

На рівені структури інформаційних зв'язків здійснюється вивід компонентів для взаємодії з сайтом. Ось тут власне лише на третьому рівні формується майбутній інтерфейс UX, який повинен набути зручного у користуванні вигляду. Четвертий рівень формує вигляд всіх сторінок та їх компонування на екрані веб-оглядача. На цьому рівні фактично формується мікрокод, що відповідає за розміщення функціональних блоків на екрані інтерфейсу. На останньому рівні здійснюється вивід сторінок, що відображують особливості функціонування UI.

Проектуючи функціональні елементи інтерфейсів необхідно визначати ті набори UX/UI з якими співпрацюють користувачі [4]. Як показав виконаний аналіз будь-який інтерфейс можна розбити на окремі функціональні групи (рисунок 2.2).

Наприклад, інтерфейс видання такого як інформаційний веб сайт ЛТЕУ містить блок динамічної взаємодії, що передбачає вивід динамічно сформованого контенту. Тут особлива увага приділяється UI, які мають фіксовані розміри для динамічних елементів, але самі являються гнучкими. Наступна велика група – це не функціональні елементи, які формують собою якісний UX, а саме так звану сітку інтерфейсу, відображають типографіку, тощо. Ця група елементів є однією з найважливіших як з точки зору реалізації, семантики так і з точки зору користувача. З метою забезпечення інтерактивності, поряд з динамічними компонентами є й елементи навігації а також веб форм. Саме ці компоненти, їх UI/UX дозволяють отримувати затребувану інформацію в найбільш звичний спосіб взаємодії. Окремою, часто необов'язковою, але

актуальною групою елементів є проектування UI/UX. з врахуванням потреб людей з обмеженими можливостями. Варто зауважити, що правильно спроектовані елементи HTML-відображення, веб-форми, а також елементи навігації частково самі формують Accessability проектованого інтерфейсу опираючись при цьому на стандарти мов програмування.

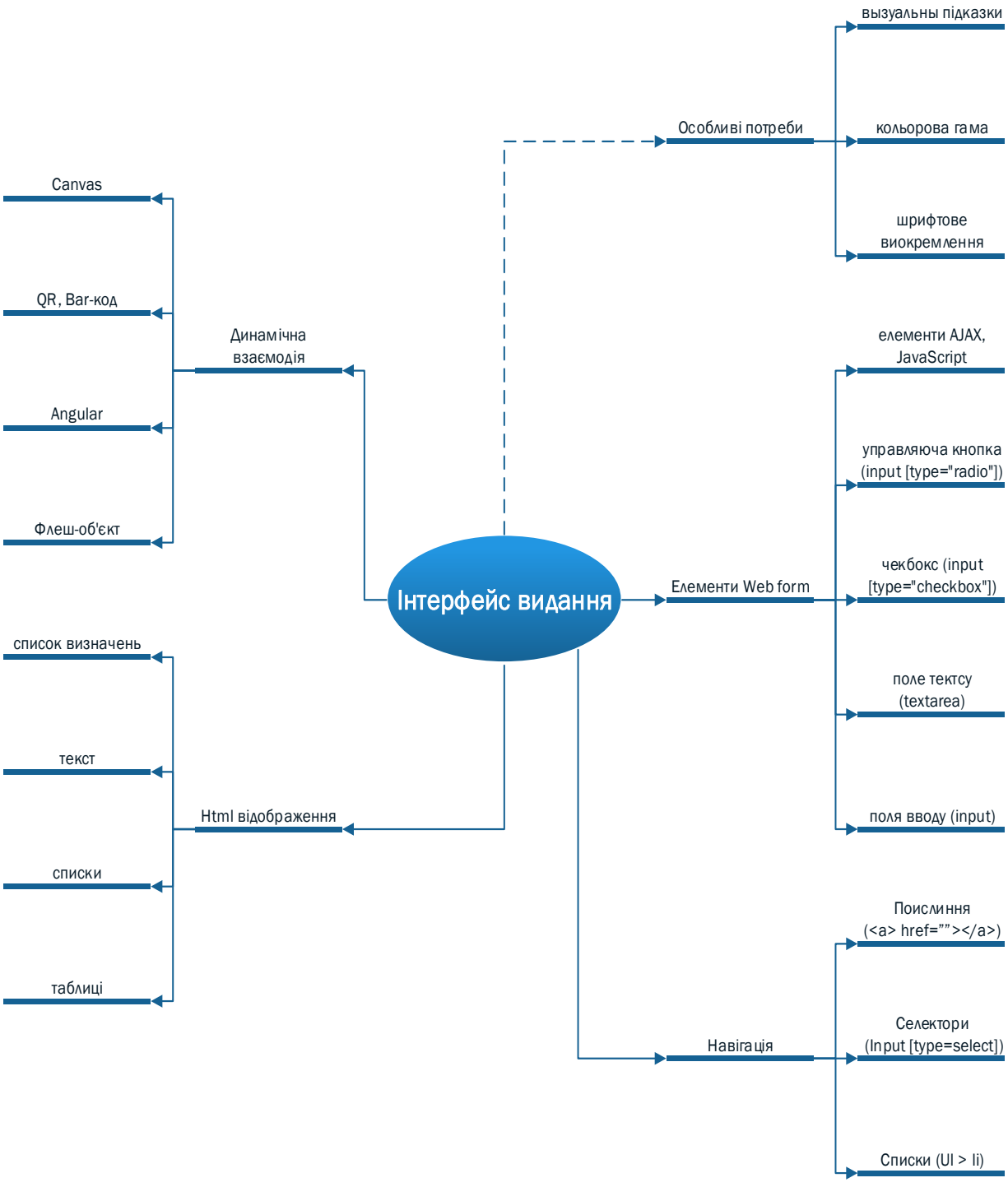

Рисунок 2.2 - Функціональні елементи Веб-інтерфейсу інформаційного видання

У результаті аналізу процесу проектування функціональних компонентів інтерфейсу користувача встановлено, що побудова якісного UI/UX повинна ґрунтуватися на поетапному підході, який охоплює всі рівні функціонування інтерфейсу — від стратегічного задуму до візуального відображення. Поділ інтерфейсу на логічні рівні дозволяє системно підійти до проектування, забезпечуючи ефективну взаємодію користувача з інформаційною системою. Зокрема, важливо визначати типові функціональні блоки, які можуть бути використані повторно в різних середовищах.

2.2 Постановка задачі та вимоги до бібліотеки компонентів

Проєктування будь-якого інтерфейсу користувача вимагає не лише креативного підходу, але й раціонального використання інструментів, що дозволяють ефективно керувати стилями, адаптивністю, структурою та взаємодією компонентів UX/UI. Сучасна веб розробка стає складнішою, зокрема, через використання більшої кількості елементі інтерфейсу користувача, тому одним із головних викликів є забезпечення узгодженості стилів, оптимізації ресурсів і зменшення навантаження на клієнтські пристрої. Тому постає необхідність у створенні інструменту, який може додати автоматизацію до ключових етапи розробки UI-компонентів і значно полегшити роботу розробника.

Головною метою є створення зручного і гнучкого генератора – task runner, який зможе автоматизувати обробку стилів, оптимізацію медіа-ресурсів та мініфікацію файлів. Для автоматизованого запуску завдань сьогодні можна виділити менеджер завдань Gulp, що являє собою інструмент збірки веб-додатків та дозволяє автоматизувати повторювані завдання, наприклад обєднанння і мініфікація CSS-, JS-файлів, запуск тестів, перезавантаження браузера і т.д. Тим самим Gulp прискорює і оптимізує процес веб-розробки (рисунок 2.3.) [2].

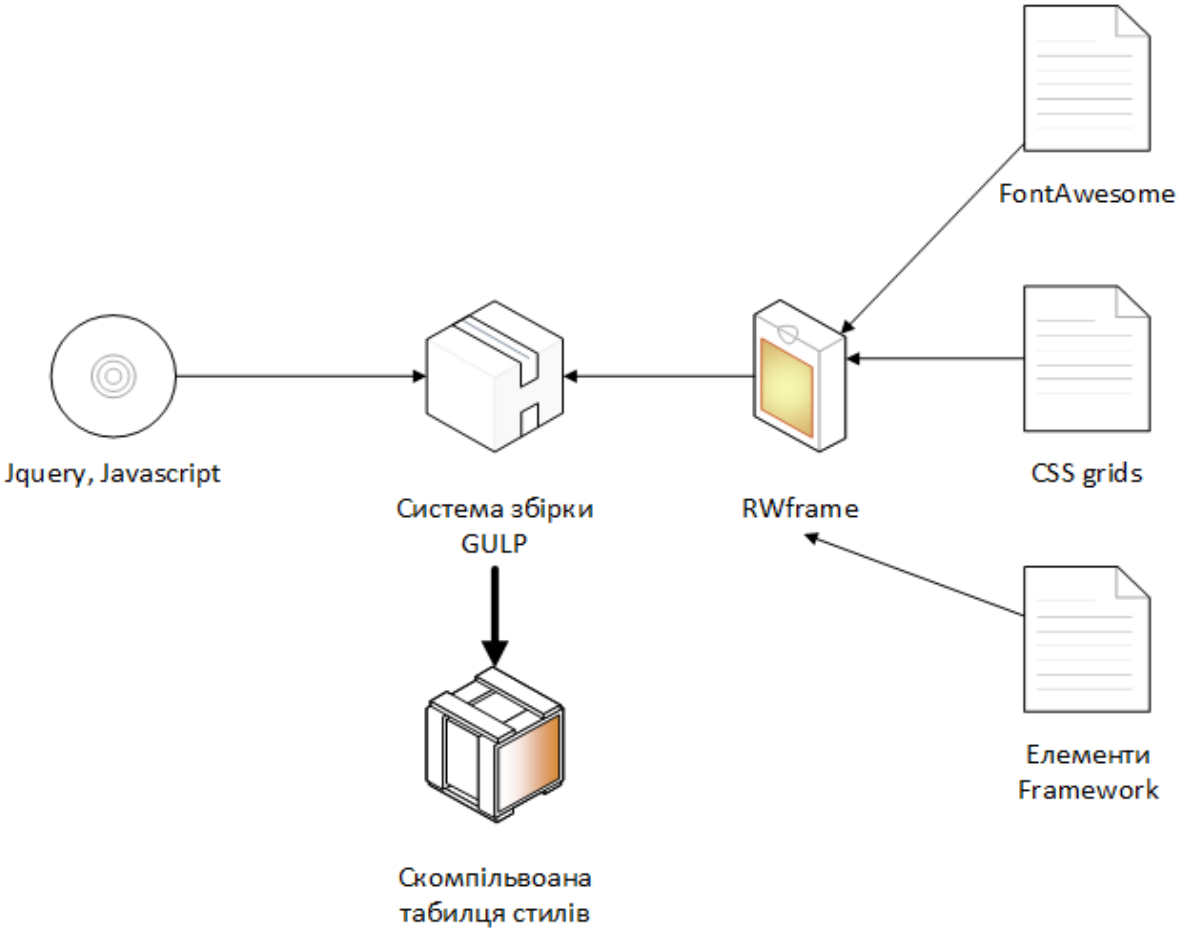

Рисунок 2.3 - Структура зв’язків системи збірки UI компонентів

Одним з найголовніших завдань у процесі розробки UI-компонентів є компіляція SCSS або іншого препроцесора у CSS, оскільки бразуер не має нативної підтримки жодного з препроцесорів, завдяки використанню препроцесорів розробник отримує значно більше інструментів для зручного та ефективного стилювання [1]. Замість довгих і заплутаних CSS-файлів можна створювати логічно структуровані блоки стилів із підтримкою вкладеності, яка дозволяє організовувати код відповідно до ієрархії HTML-розмітки [12]. Змінні дають можливість централізовано керувати параметрами дизайну — кольорами, відступами, шрифтами. Це зручно, особливо коли потрібно реалізувати можливість швидкої зміни теми або створити дизайн-систему з єдиними правилами [10]. Крім того, SCSS підтримує міксини — шаблонні фрагменти стилів, які можна багаторазово використовувати у різних компонентах без дублювання коду. Автоматизована компіляція SCSS у CSS за допомогою Gulp

дозволяє миттєво перетворювати сучасний SCSS-код у валідний CSS, який розуміють браузери, тим самим забезпечуючи безперервність розробки.

Ще однією важливою задачею, яку вирішує система на базі Gulp, є автоматичне додавання вендорних префіксів. Через те, що не всі браузери однаково підтримують нові CSS-властивості, існує необхідність у додаванні специфічних префіксів (наприклад, -webkit-, -moz-, -ms-) [18, 19]. Робити це вручну — не лише незручно, але й потенційно небезпечно: можна легко пропустити щось важливе. Використовуючи плагін Autoprefixer, Gulp автоматично визначає, які саме властивості потребують префіксів, і додає їх на основі актуальної статистики підтримки браузерами. Це дозволяє значно підвищити кросбраузерну сумісність без зайвих зусиль з боку розробника.

В рамках розробки сучасного веб-інтерфейсу важливо також оптимізувати медіа-контент, зокрема — зображення. Часто саме графіка є найбільш “важким” елементом на сторінці, що помітно впливає на швидкість завантаження. Завдяки таким інструментам, як gulp-imagemin, система збірки автоматично виконує оптимізацію зображень, зменшуючи їх розмір без втрати видимої якості. Це стосується не лише стандартних форматів (JPEG, PNG, SVG), а й новітніх — таких як WebP або AVIF, що забезпечують ще кращу компресію. У підсумку користувач отримує швидко завантажувану сторінку, а розробник — автоматизовану обробку зображень у робочому процесі.

Щоб зменшити загальний обсяг проєкту і прискорити завантаження сайту, необхідно також подбати про мініфікацію CSS та JavaScript-файлів. Це означає видалення всіх зайвих пробілів, коментарів, символів переходу на новий рядок та інших “непотрібностей”, які не впливають на виконання, але збільшують розмір файлу. Gulp, використовуючи відповідні плагіни (gulp-clean-css для CSS та gulp-uglify для JavaScript), автоматично виконує мініфікацію на етапі збірки. У результаті готовий продукт займає менше місця, а завантажується швидше, що особливо важливо для мобільних користувачів [22].

У сучасній розробці однією з кращих практик є об’єднання окремих стилів і скриптів у бандли (bundle-файли). Це дозволяє зменшити кількість HTTP-

запитів, які виконує браузер при завантаженні сторінки. Gulp виконує об’єднання всіх необхідних CSS- або JS-файлів у єдиний фінальний файл, з урахуванням порядку підключення. Наприклад, замість десятка окремих стилів компонентів користувач отримує один оптимізований і мініфікований style.bundle.css, що містить усі потрібні стилі. Це пришвидшує завантаження сторінки, знижує навантаження на сервер та забезпечує більш плавну взаємодію з веб-додатком.

Також проектуючи генератор бібліотеки компонентів необхідно передбачити використання так званого живого перегляду (Live Reloading). Це дозволить під час редагування стилів або коду компонентів автоматично перезавантажувати браузер за допомогою watchers, які автоматично відстежують зміни у вихідних файлах і миттєво оновлюють сторінку. Це реалізується завдяки підключенню до локального сервера, який фоново моніторить зміни в коді. Такий підхід значно економить час і дозволяє розробнику зосередитись на результаті, а не на технічних дрібницях .

Останнім, але не менш важливим етапом є генерація адаптивних сіток та UI-компонентів на основі SCSS. З огляду на велику кількість пристроїв із різними розмірами екранів, адаптивність стала стандартом. Проте замість ручного створення стилів для кожного окремого елементу розробник може скористатись динамічними SCSS-функціями та міксинами, які дозволяють автоматично формувати адаптивну поведінку елементів. Наприклад, розрахунок ширини блоків у відсотках, генерація колонок сітки, адаптивне масштабування зображень — усе це може відбуватись у рамках SCSS-функцій, які потім автоматично компілюються Gulp’ом у фінальний CSS.

2.3. Архітектура бібліотеки UI-компонентів

Проектована бібліотеки компонентів повинна забезпечувати принцип відкритості-закритості, а також можливість зміни конфігурації основних модулів. Так, проведений аналіз показав, що всі елементи проектованої бібліотеки повинні забезпечувати достатню гнучкість як у стилізації так і

підтримці того, чи іншого функціоналу. Відтак основні блоки включають різні компоненти та етапи обробки стилів, такі як система змінних (глобальні та локальні), адаптивність (з критичними розширеннями), візуалізація (анімування та трансформації), позиціонування (модель BOX), шрифти, а також генерація сітки колонок (рисунок 2.4).

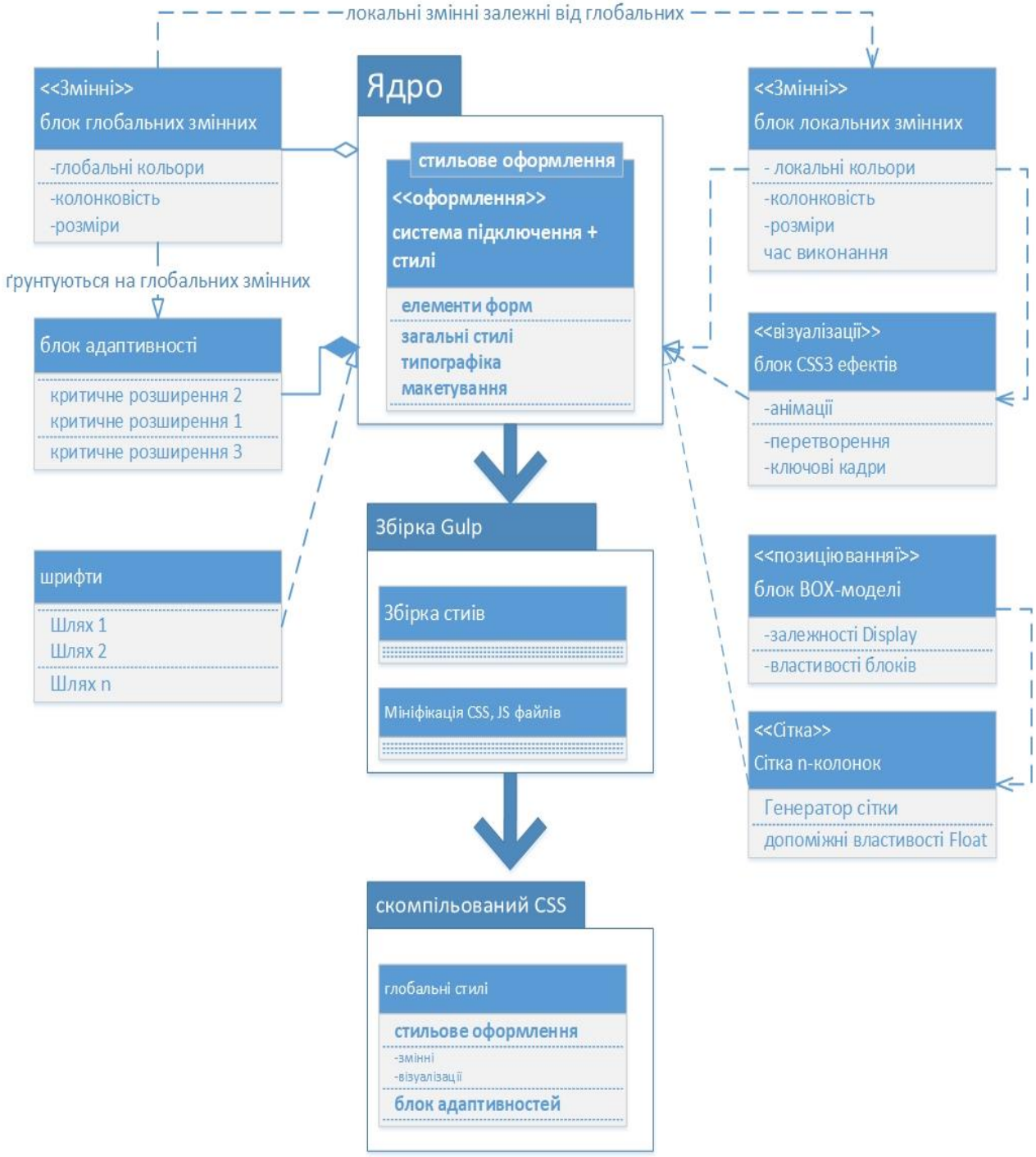

Рисунок 2.4. - Функціональна схема бібліотеки компонентів для створення UI компонент

У розроблюваній бібліотеці компонент «змінні» поділено на два типи – блок обов’язкових глобальних змінних. Тут визначаються головні кольори проектованого інтерфейсу, розміри контейнера колонок, розміри картинок, час анімацій, компоненти ключових кадрів для анімацій. Блок необов’язкових змінних може вказувати інше від глобального налаштування властивості для певного селектора, проте він завжди ґрунтується на батьківських змінних [17].

З метою забезпечення якості подавання візуальної інформації у блоці глобальних змінних визначено функцію, що відповідає за подання графічної та інформації контейнерного типу (flash canvas). Спроектована функція спрямована на те, що вписати картинку за більшим розміром у певне відведене місце, причому навідмінно від звичних засобів подання картинок тут реалізовано спосіб автоматичного вписування картинки у заданий розмір з повним збереженням пропорцій та вмісту (рисунок 2.5).

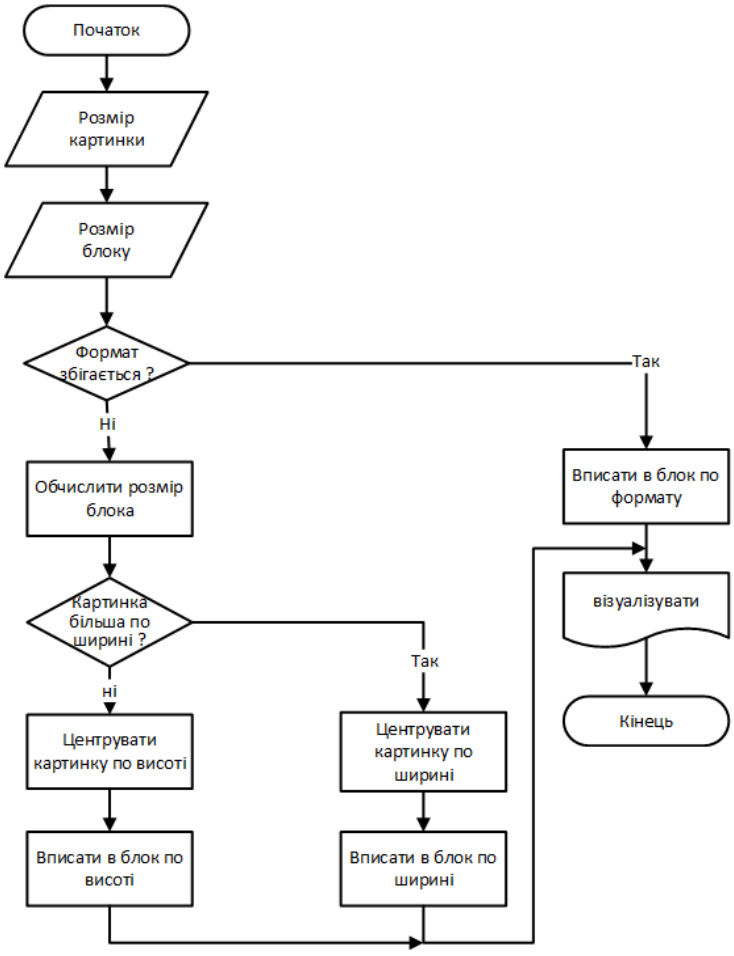

Рисунок 2.5 - Блок-схема автоматизованого масштабування зображень

Архітектура проектованої бібліотеки повинна передбачити генератор сіток у якому розмір колонок розраховується виходячи з розміру навколишніх блоків, що значно спрощуватиме процес побудови адаптивного UI. А у випадку застосування блоку адаптивностей усі розраховані колонки на попередні етапах автоматично масштабуються та перебудовують інтерфейс.

Блок Box-моделей повинен передбачати набір функцій-хелперів які дозволять змінювати поведінку компонента у блоковій моделі браузера за вимогою та реалізовуватимуть принципи атомарного CSS.

У проектованому фреймворку блок візуалізацій призначений для покращення сприйняття проектованого інтерфейсу, а також задля забезпечення логічного зв’язку між перебудовою інтерфейсу користувача.

Підключення сторонніх шрифтів — це невід’ємна частина створення унікального, сучасного і привабливого інтерфейсу. Веб-дизайн давно вийшов за рамки стандартних системних шрифтів, і сьогодні практично кожен проєкт використовує власні або сторонні шрифти, які відповідають бренду, стилістиці або загальному настрою інтерфейсу. Тому у проектованій бібліотеці важливо не просто підключити шрифт, а зробити це оптимально та адаптивно, щоб забезпечити швидке завантаження і гарний вигляд тексту на будь-якому пристрої. Так як шрифти можуть підключатися як з сторонніх так і з локальних сховищ даних, а розроблюваний модуль повинен підтримувати оптимізацію доставки, fallback рішення, перевіряти наявність того, чи іншого шрифта за вказаним посиланням.

З метою забезпечення функціональних можливостей для людей з особливими потребами необхідним є підключення JavaScript та зовнішніх бібліотек jQuery і jQuery-UI, що забезпечать реалізацію концепції активної взаємодії з клієнтом веб інтерфейсу.

Усі ці блок під’єднюються до ядра Gulp, розроблюваної бібліотеки, яка у свою чергу генерує собою головну таблицю стилів та формує візуальний інтерфейс користувача з UI компонент, що описані в стилях, скриптах, та html. Варто зауважити, що у проектованій архітектурі можна здійснювати

перевизначення всіх компонентів, які використовуються при створенні веб-інтерфейсу. Перевизначення відбувається за допомогою специфічності селекторів реперних точок, а також місця їх підключення. Решта ж модулів таких як шрифти, шрифтові іконки, тощо підключаються як додаткові блоки за потребою.

Таким чином у розроблюваній архітектурі бібліотеки компонентів для створення UI компонентів з використанням препроцесорів, розробник створює UI елемент відповідно до вимог, налаштовує загальні параметри (кольори, шрифти кількість колонок, відступи, анімації тощо), а система запуску задач Gulp відповідно до визначеної структури та спроектованих зав’язків між функціональними компонентами в автоматичному режимі збирає фінальний, мінімізований програмний код всіх UI компонентів в єдиний бандл придатний для рендеру браузером.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ БІБЛІОТЕКИ UI КОМПОНЕНТ

При розробці бібліотеки компонентів доцільно вибрати таке середовище проектування яке б могло працювати з різними веб-орієнтованими мовами програмування, для забезпечення єдиного цілісного процесу розробки та супроводження створюваного інтерфейсу користувача.

3.1 Вибір інструментального програмного забезпечення

Як відомо проектування інтерфейсу користувача вимагає поєднання одразу кількох мов програмування HTML5, CSS3, JavaScript, AJAX, а також під'єднання зовнішніх модулів у вигляді набору фреймворків [22, 23]. Тому задля забезпечення цілісності та уніфікації розроблення інтерфейсу постає потреба вибору такого інтегрованого середовища (IDE), що являють собою комплексні програмні засоби, для полегшення розробки та модифікації тексту програми. Такі програмні засоби містять багато корисних функцій та можливостей, беруть на себе значну кількість рутинної роботи.

При виборі інструментального середовища IDE необхідно зважати на здатність середовища для підсвічування програмного коду, що дозволяє пришвидшити навігацію та скоротити кількість помилок. Загалом така функція сьогодні реалізована у переважній більшості IDE (таблиця 3.1), але нажаль реалізація підсвітки синтаксису мови програмування є унікальна, оскільки кожна мова має свій набір функціонально значущих операторів, тому обираючи середовище розробки необхідною умовою є здатність програмного комплексу підтримувати якнайбільше варіантів синтаксисів мов.

```
// коментар
int main()
{
for (int i = 0; i < MAX; i++)
printf ("%d\n", i);
return 0;
}
```

Наступним критерієм є здатність здійснення контролю версій розробки. Цей критерій є чи не найголовніший при, адже дозволяє організувати командну роботу над проектом, а також здійснювати навігацію між старими та новими редакціями розроблюваного програмного продукту актуалізуючи при цьому версію розроблюваного програмного коду (рисунок 3.1)

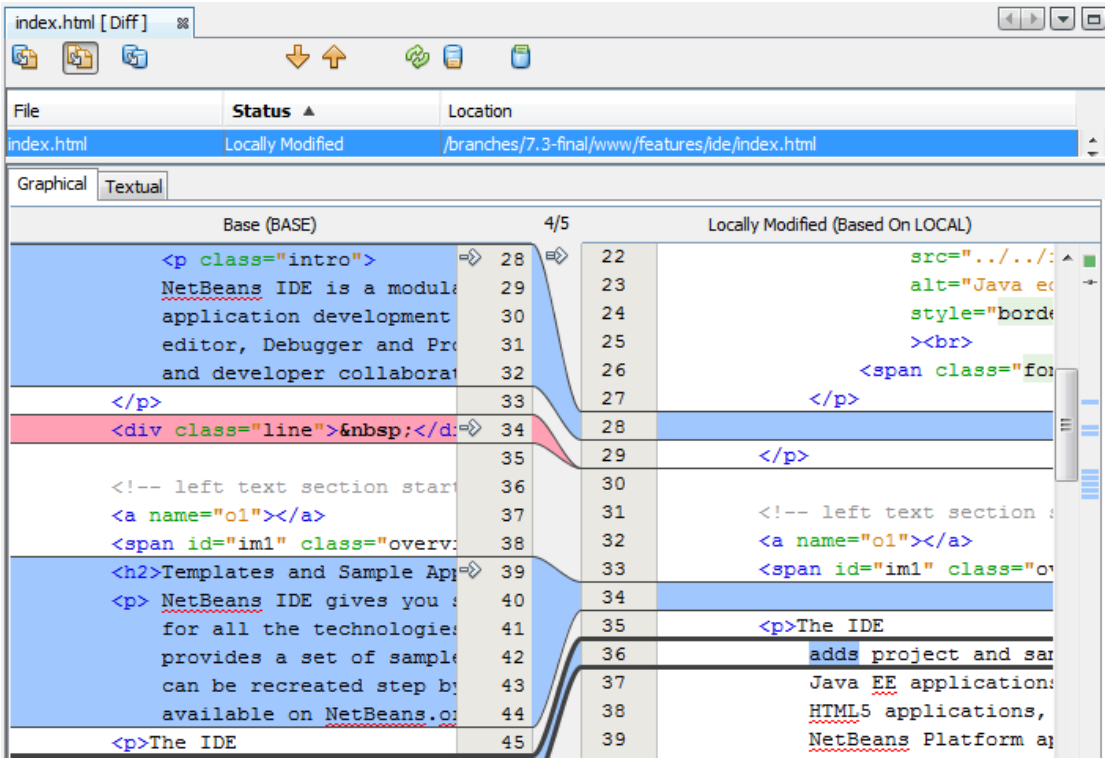

Рисунок 3.1 - Система контролю версій Diff

Навігатор по коду у IDE слугує для спрощеного переходу між ідентифікаторами, класами, об’єктами та їх властивостями. Цей критерій дозволяє пришвидшити суто навігаційні процеси, що значно підвищує зручність програмування та розвитку програмного продукту. В купі з навігатором забезпечити семантично провальну структуризації може засіб для форматування, автоматичного пошуку балансу дужок, тощо. Менш важливим аспектом при виборі IDE є наявність багатомовних версій, оскільки вважається прикладом поганого тону використання іншого середовища розробки ніж середовища з англійським інтерфейсом. Це зумовлено потребою комунікації між клієнтом та замовником. Проте «рідна» локалізація може бути зручним варіантом при роботі в єдиній системі та країні.

Таблиця 3.1 - Порівняння популярних IDE

№	Назва IDE/Редактора	Ліцензія	Плат форма	Плагіни	Швидкодія	Авто доповнення	Debug	Git	Інтер нет	Переваги (+)	Недоліки (–)
1	**VS Code**	Безкоштовна	Win, macOS, Linux	☑	Висока	☑	☑	☑	✕	+ Швидкий, розширюваний,	– Може «підвисати»
2	**WebStorm**	Платна	Win, macOS, Linux	☑	Висока	☑	☑	☑	✕	+ Професійний IDE, зручний дебаг, підтримка TypeScript	– Платна підписка, іноді ресурсоємний
4	**Sublime Text**	Умовно безкошт.	Win, macOS, Linux	☑	Дуже висока	Обмежено	✕	☑	✕	+ Легкий, швидкий, мінімалістичний	– Без доповнень функціонал обмежений,
6	**Brackets**	Безкоштовна	Win, macOS, Linux	Обмежено	Середня	☑	✕	✕	✕	+ Живий перегляд, зручний для верстки	– Низька розширюваність, застарілий інтерфейс
8	**NetBeans**	Безкоштовна	Win, macOS, Linux	☑	Середня	☑	☑	☑	✕	+ Готовий з коробки, підтримка JavaScript	– Менш популярний серед фронтендерів
9	**CodeSandbox** (Web IDE)	Freemium	Web	✕	Середня	☑	☑	☑	☑	+ Робота з браузера, інтеграція з GitHub	– Потрібен інтернет, обмеження у безкоштовній версії
10	**StackBlitz** (Web IDE)	Freemium	Web	✕	Середня	☑	☑	☑	☑	+ Швидкий старт, підтримка сучасних фреймворків	– Немає доступу до локальних файлів, залежність від мережі

Оскільки веб-інтерфейс являє собою по суті зовнішній програмний продукт для доступу до відомостей бази знань предметної галузі, то доцільно його виносити як окрему гілку спільного проекту, що забезпечує інтегрованість рішення разом з повністю незалежним підходом у програмуванні, то критерій, що відноситься до забезпечення IDE ведення проектів є важливим. Також до найбільш пріоритетних вимог слід віднести можливість рефакторингу коду. Суть рефакторингу зводиться до здійснення автоматичного пошуку та відновлення втрачених зав'язків файлової структури розроблюваного проекту в автоматичному режимі або за вимогою користувача [28]. (таблиця 3.1).

Для тестування розроблюваного продукту необхідним є розміщення файлів на зовнішньому або локальному сервері, якщо у випадку з локальним сервером усе зводиться до відкриття сторінки у браузері, то у випадку використання зовнішнього сервера необхідно здійснювати переміщення файлів за допомогою FTP- клієнта, тому наявність у IDE локального сервера, а також FTP- клієнта є корисною перевагою, яка знову ж таки покликана на збільшення швидкодії автоматизуючи дії користувача. Критерій модульності дозволяє здійснювати додавання затребуваного функціоналу у вигляді під'єднюваних модулів до системи. До таких під'єднюваних модулів відноситься компілятор препроцесора LESS – Node.JS.

Більшість сучасних середовищ розробки розробляються з тими чи іншими функціональними особливостями, тому часто є платними. Хоча плата за використання продукту дає певні привілеї, проте такий програмний засіб на завжди є кращим з точки зору юзабіліті. Окрім того при виборі IDE увагу слід звернути на кросплатформенність або наявність пакетів вихідних файлів з можливістю компілювання під свою операційну систему [27].

В результаті обумовдених критеріїв та здійсненого аналіу популярних середовищ розробки (IDE) обрано Visual Studio Code — легке, безкоштовне та потужне середовище, яке за кілька років стало фактичним стандартом серед фронтенд-розробників.

Однією з ключових причин цього вибору є його гнучкість і підтримка сучасного стеку технологій. Visual Studio Code чудово інтегрується з інструментами, необхідними у межах розробки бібліотеки — такими як Gulp, Sass/SCSS, PostCSS, Webpack та інші. Завдяки широкому спектру розширень, що легко встановлюються прямо з внутрішнього маркету, розробник може адаптувати середовище під власні потреби — наприклад, встановити автокомплітацію для SCSS, лінтери, підсвічування синтаксису або плагіни для зручної роботи з Git.

Ще однією сильною стороною VS Code є вбудована підтримка терміналу. Це дозволяє безпосередньо запускати скрипти збірки, перевіряти валідність коду або слідкувати за логами сервера розробки, не виходячи з середовища. Це особливо зручно при налаштуванні автоматичної компіляції SCSS у CSS, запуску Gulp-задач або використанні систем контролю версій.

Окрім того, Visual Studio Code має невисокі системні вимоги, швидко працює навіть на середньостатистичних комп’ютерах і не потребує складної конфігурації. Його підтримка кросплатформеності дозволяє працювати як на Windows, так і на macOS чи Linux без втрати функціоналу.

3.2 Розробка компонентів бібліотеки

Проектована бібліотека компонентів, використовує для своєї роботи модульну структуру, яка базується на відокремленні логічних модулів з яких складається UI елемент [11]. В загальному випадку типовий UI елемент, який використовується для побудови веб сайту або додатку складається з каркасу – розробленого на мові HTML, представлення – CSS та функціональної частини, якщо такий компонент передбачає можливість інтерактивної взаємодії або анімації. Усі розроблені компоненти повинні сформувати собою логічні блоки, нарпиклад колонку новим, меню навігації тощо. Такі функціональні блоки розміщують у так званій сітці (рисунок 3.2) [19].

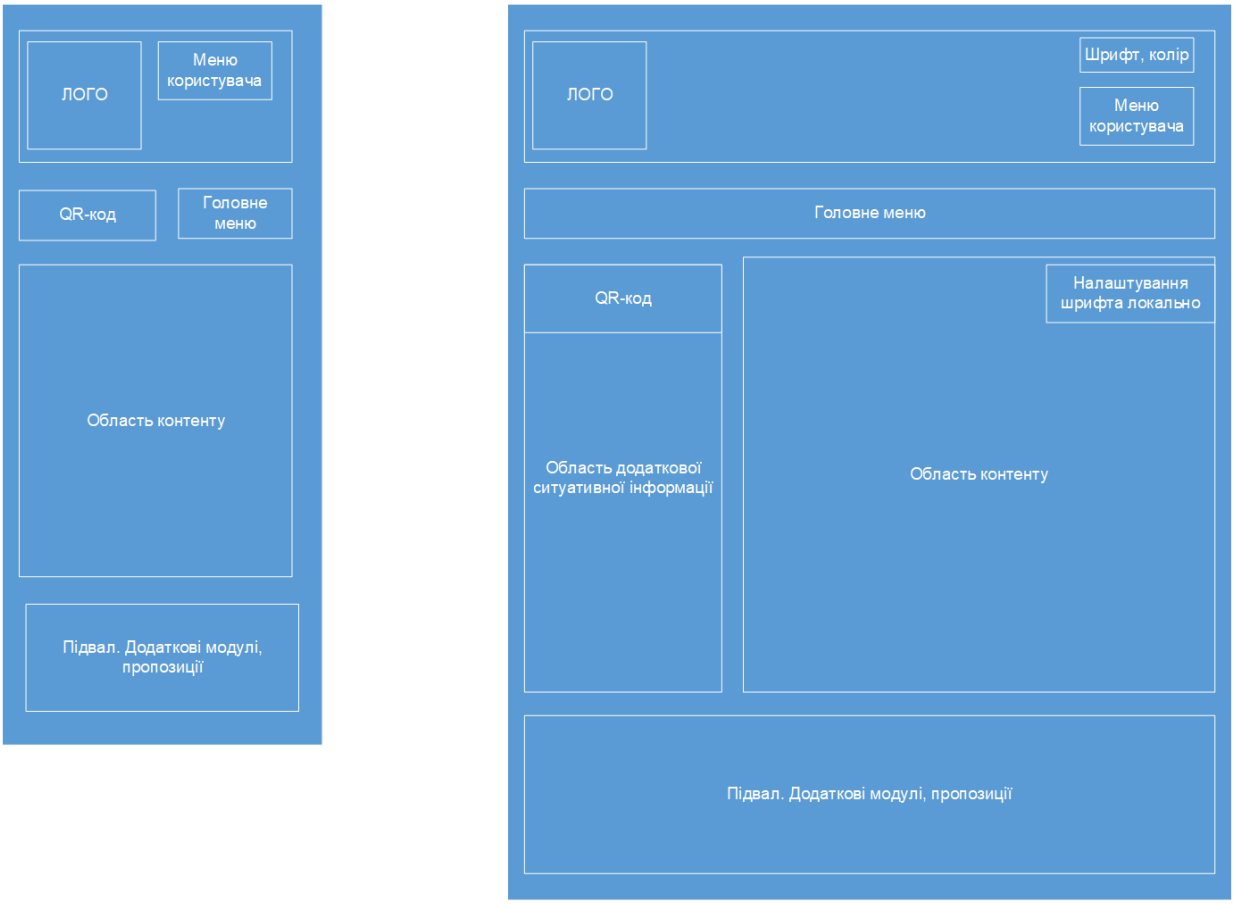

Рисунок 3.2 - UI-компоненти, що формують сторінку

Так, як усі елементи з яких формується той, чи інший веб ресурс є доволі типовими, то проектування доцільно почати з укладання файлової структури. В такій структурі всі HTML, CSS, JS та інші самостій одиниці розміщуватимуться в окремих папках, щоб відповідати принципу розділення відповідальностей - Seapration of concerns. Використання SoC дозволяє структурувати проєкт так, щоб кожен UI/UX компонент займався лише своєю справою. Наприклад, у веб-розробці часто окремо зберігаються HTML (структура), CSS (оформлення), JavaScript (логіка) [14]. Це дозволяє змінити дизайн без впливу на логіку, або оновити скрипти, не змінюючи саму структуру сторінки. Так як збірка проекту проводитеметься за допомогою спеціалізованого task runner, то весь згенерований фронтенд повинен зберігатись в окремомму місці для спрощення доступу до готових ресурсів. Відтак запропоновано розмістити проект у двох головних длиректоріях – *app*, що відповідає за ресурси для проекту і *build* — структура вихідної збірки. Вона описує, де повинні зберігатися готові HTML,

CSS, JS, зображення, шрифти тощо. Цей розділ дозволяє легко змінити місце призначення зібраних файлів, не змінюючи безпосередньо Gulp-задач (рисунок 3.3)

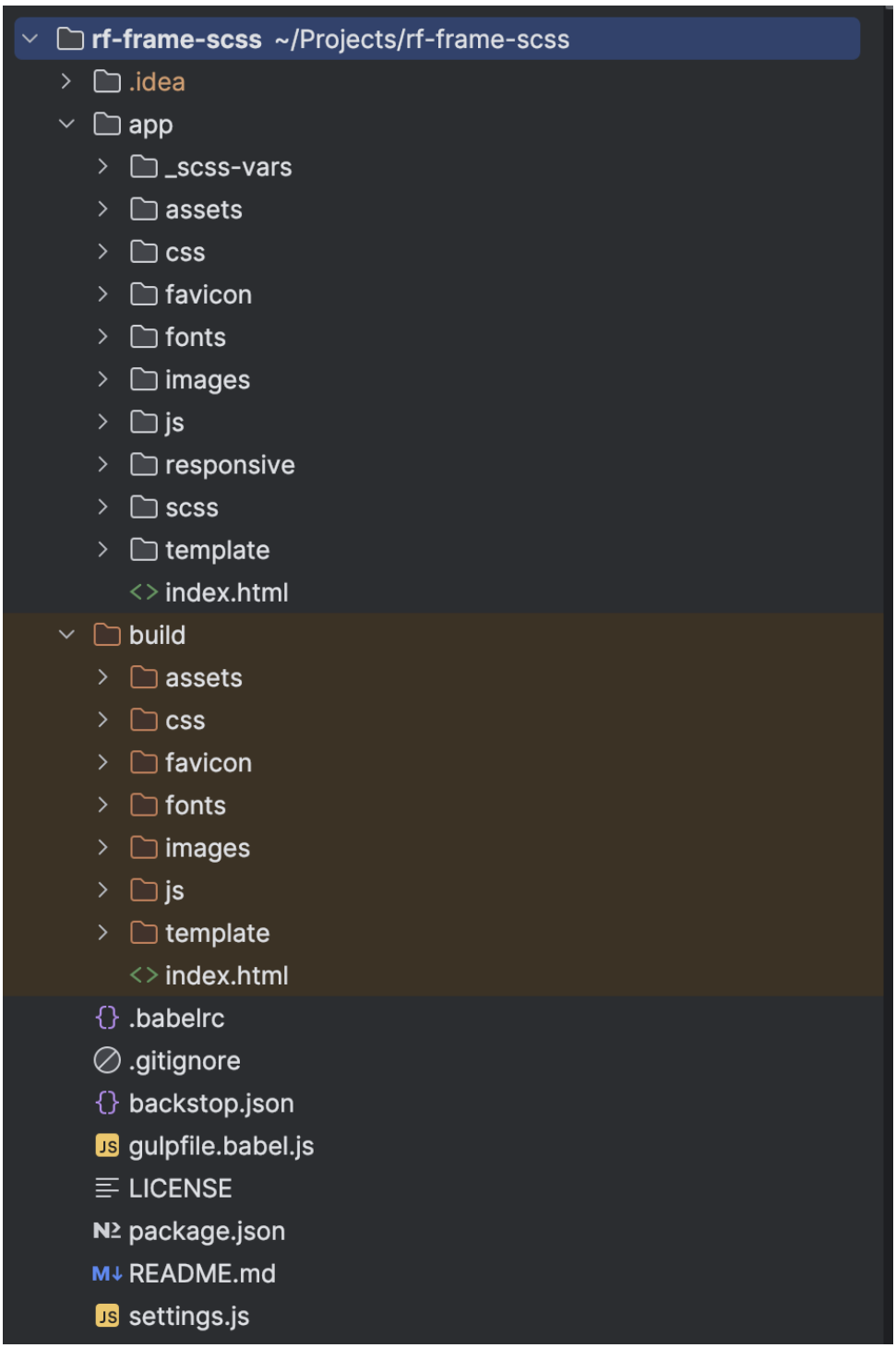

Рисунок 3.3 - Файлова структура проєкту

Відтак в проектованій бібліотеці Gulp налаштовано на перегляд змін у файлах директорії app. Кожна зміна викликає генерацію відповідних ресурсів з

перезаписом у папці build. Якщо взяти до уваги те, що відповідно до завдань проектованого інтерфейсу структура може змінюватись, в такому випадку необхідно передбачити JSON-файл конфігурації (Додаток А). Як показало проведене дослідження логічним є розбиття всього процесу розробки на окремі блоки або логічні секції. Було зрозуміло, що кожна задача (компіляція стилів, обробка зображень, підключення шаблонів, тощо) потребує окремих конфігурацій: шляхів до вихідних файлів, директорій для збереження результату, а також специфічних параметрів (наприклад, якість зображення чи формат мініфікації).

Так з’явилися основні секції конфігураційного об’єкта:

- build — структура вихідної збірки. Вона описує, де повинні зберігатися готові HTML, CSS, JS, зображення, шрифти тощо. Цей розділ дозволяє легко змінити місце призначення зібраних файлів, не змінюючи безпосередньо Gulp-задач.
- src — шляхи до вихідних файлів, з якими буде працювати Gulp. Тут зібрано всі ресурси, які потребують обробки: від SCSS до фавікона. Це забезпечує централізоване керування та легке масштабування: при зміні структури проєкту достатньо змінити лише цей об’єкт.
- responsiveImage — було додано для створення адаптивних зображень. Це стало особливо актуально з поширенням мобільних пристроїв, де потрібні полегшені версії зображень з меншою шириною та якістю, що зберігає продуктивність.
- htmlInlcudeSettings — конфігурація HTML-шаблонізації, яка дає змогу включати загальні компоненти (хедер, футер, меню тощо) через спеціальний префікс. Це спрощує підтримку коду і дозволяє використовувати модульний підхід.
- clean — окрема вказівка на директорію, яку необхідно очищати перед збіркою. Це дозволяє уникнути накопичення застарілих файлів.

- isSprite_RASTER, isSprite_VECTOR — параметри для увімкнення чи вимкнення генерації спрайтів. Це дозволяє легко адаптувати збірку під проєкти, де спрайти використовуються або ні.
- compress_Css, legacyGrid — додаткові параметри компіляції SCSS, які забезпечують гнучкість у форматуванні коду та підтримку сіток для старіших браузерів.
- browser_sync, isProxy, isProxy_path, port — конфігурація для автоматичного оновлення сторінки при зміні файлів, що значно пришвидшує тестування і відлагодження інтерфейсу.

Таким чином, settings — це не просто набір шляхів чи параметрів. Це результат усвідомленого підходу до автоматизації фронтенд-розробки, сформований з урахуванням принципів **Separation of Concerns** (розділення відповідальностей), повторюваності задач, адаптивності проєкту та зручності підтримки. Така структура робить проєкт більш надійним, масштабованим і легким для нових учасників команди.

Наступним кроком розробки є налаштування gulp. Тут необхідно здійснити вибір необхідних плагінів, оскільки сам gulp лише виконує команди та забезпечує зручний інтерфейс для асинхронної роботи.

Головним ядром є власне Gulp — інструмент автоматизації, що дозволяє створювати та запускати задачі. Завдяки Gulp можна об’єднувати серії дій: компіляцію SCSS, мініфікацію зображень, підключення SVG, запуск локального сервера тощо. Він лежить в основі всіх процесів і дозволяє організувати збірку максимально зручно.

```
const gulp = require("gulp");
```

Для того, щоб забезпечити зручну оброку помилок використано plumber, який рятує від краху всього Gulp-процесу у випадку помилки. Наприклад, якщо при компіляції стилів виникає синтаксична помилка, gulp-plumber дозволяє вивести повідомлення в консоль і продовжити збірку, не зупиняючи все повністю. Це особливо зручно при розробці — коли хочеться бачити результат навіть з помилками.

Для Компіляції SCSS: gulp-sass, autoprefixer, sourcemaps, rename обрано `gulp-sass`. Цей плагін компілює SCSS у звичайний CSS. SCSS — це розширення CSS з можливістю використання змінних, міксинів, вкладеності та інших зручних речей. Ми використовуємо node-sass як компілятор, що забезпечує високу продуктивність. Autoprefixer Автоматично додає вендорні префікси (наприклад, -webkit-, -moz-) до CSS-властивостей, що потребують підтримки у старіших браузерах. Це спрощує кросбраузерність без ручного втручання.

`const sourcemaps = require("gulp-sourcemaps")` - Дозволяє створити мапи відповідності між скомпільованим CSS і оригінальним SCSS-файлом. Це дуже корисно під час налагодження: в інструментах розробника браузера можна побачити, в якому SCSS-файлі міститься конкретне правило.

`const rename = require("gulp-rename");` - Дозволяє перейменовувати файли. Наприклад, додати суфікс .min до мініфікованих файлів (style.min.css).

Робота з HTML і SVG: gulp-file-include, gulp-inject-svg

```
const fileinclude = require("gulp-file-include");
```

Надає можливість включати в HTML інші HTML-файли за допомогою директиви @@include. Це дуже зручно для створення шаблонів (наприклад, повторне використання header, footer).

```
const injectSvg = require("gulp-inject-svg");
```

Вставляє SVG-файли безпосередньо в HTML, що дозволяє керувати ними через CSS. Наприклад, можна змінювати колір і розмір іконок, задавати ефекти ховера, а також уникати зайвих HTTP-запитів.

Мініфікація JavaScript та JSON: gulp-uglify, gulp-jsonminify

```
const uglify = require("gulp-uglify");
```

Цей плагін “стискає” JavaScript, видаляючи коментарі, пробіли і скорочуючи змінні. У результаті виходить легкий .min.js файл, який швидше завантажується.

```
const jsonminify = require("gulp-jsonminify");
```

Схожим чином, jsonminify стискає JSON-файли. Це зручно, коли ваш проєкт включає локалізаційні файли, налаштування чи інші JSON-структури.

Оптимізація зображень: gulp-imagemin, gulp-sharp-responsive

```
const imagemin = require("gulp-imagemin");
```

Зменшує розмір зображень без втрати якості. Підтримує формати JPG, PNG, GIF, SVG. Завдяки цьому сторінки сайту швидше завантажуються.

```
const sharpResponsive = require("gulp-sharp-responsive");
```

Цей плагін створює адаптивні версії зображень у кількох розмірах і форматах (JPEG, WebP, AVIF). Це дозволяє використовувати зображення, оптимізовані під різні екрани: мобільні, планшети, десктопи.

Гнучке підключення плагінів: gulp-load-plugins

```
var $ = require("gulp-load-plugins")();
```

Автоматично підключає всі Gulp-плагіни, які починаються з gulp-. Це дозволяє звертатися до них через змінну $, наприклад, $.rename() або $.plumber(). Зменшує кількість імпортів і покращує читабельність.

Робота з іконками та спрайтами: gulp.spritesmith, merge-stream, vinyl-buffer, gulp-svg-sprites

```
const spritesmith = require("gulp.spritesmith");
```

Створює растрові спрайти (PNG) — об’єднання декількох зображень в одне, що зменшує кількість HTTP-запитів. До спрайта автоматично створюється SCSS або CSS-файл з позиціонуванням кожного елемента.

```
const merge = require("merge-stream");
```

Цей плагін дозволяє об’єднувати кілька потоків Gulp в один — наприклад, коли треба обробити зображення та відповідний SCSS-файл одночасно.

```
const buffer = require("vinyl-buffer");
```

Застосовується у зв’язці зі spritesmith для обробки потоків Gulp, коли потрібен буферизаційний доступ до файлів (наприклад, для подальшої оптимізації).

```
const svgSprite = require("gulp-svg-sprites");
```

Генерує SVG-спрайти — набір SVG-іконок, які вставляються як один великий файл і можуть використовуватись через <use>. Це сучасна та ефективна техніка для роботи з векторною графікою.

Ці всі плагіни разом створюють повноцінний, масштабований, оптимізований процес побудови сучасної бібліотеки UI-компонентів, з урахуванням кросбраузерності, швидкості завантаження та адаптивності (Додаток Б).

Наступним кроком є створення компонентів бібліотеки з використанням SCSS (рисунок 3.4).

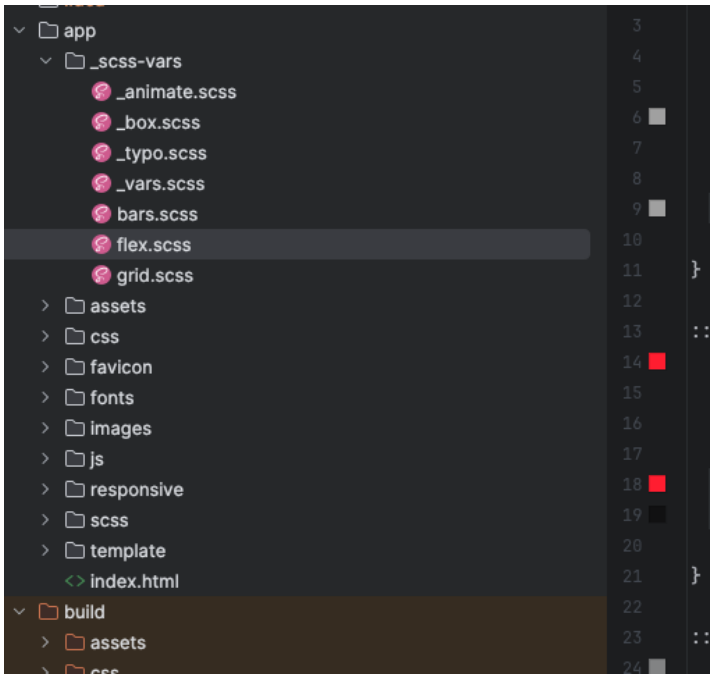

Рисунок 3.4 - Файлова структура модулів бібліотеки

До основник компонентів віднесено модуль анімацій, головним завданням якого забезпечити прийнятий стандартизований міксин з можливістю налаштування для реалізації однакових невеликих анімаційних ефектів натискань на кнопки або переходів.

```
@mixin animate($all:all, $time:0.5s, $efect:ease, $delay:0s){
  transition: $all $time $efect $delay;
}
/* YEAH it is keyframes ))) */
@mixin keyframes($name) {
  @keyframes #{$name} {
    @content;
  }
}
@mixin animation_keyframes($keyframes_name, $fillmode, $delay_kf:0) {
  animation: $keyframes_name;
  animation-fill-mode: $fillmode;
  animation-delay: $delay_kf;
}
```

Наступним модулем є модуль забезпечення роботи з BOX-моделюю браузера, а також цей модуль надає інтерфейс для роботи з пропорціями медіа та таблицями. Модуль _typo.scss – управляє підключенням шрифтів забезпечуючи генерацію

CSS з відповідними зворотними сумісно тями, наприклад міксін font-face отримує набір аргументів, а повертає згенерований набір шрифтів.

```
@include font-face(Samplino, fonts/Samplino);
@include font-face("Samplina Neue", fonts/SamplinaNeue, bold, italic);
@include font-face(Samplinoff, fonts/Samplinoff, null, null, woff);
@include font-face(Samplinal, fonts/Samplinal, 500, normal, eot woff2
woff);
```

Модуль _vars.scss – є головною відправною точку, оскільки тут конфігуруються кольори, розміри колонок та зворотна сумісність з непідтримуваними властиваостями

Модулі flex.scss та grid.scss – генерують сітку та допоміжні класи як з використанням CSS flex так і CSS grid.

Відтак отримана структура (таблиця 3.1)модулів бібліотеки забезпечує можливість стандартизації розробки кастомізованих UI компонент, що базуються на компонентах бібліотеки

Таблиця 3.1 - Фрагмент SCSS mixins бібліотеки

Значення	Опис
.font-face(@fontname, @fontfile)	**Сімейство зовнішніх шрифтів** Параметри: @fontname: назва, @fontfile: шлях до фізичного шрифта
.col(@col_count:3, @col_width:1, @col_padding:10px)	**Колонки макету** Параметр (@col_count – кількість колонок, @col_width: ширина, @col_padding: внутрішній відступ
.radius(@radius:1em)	**Радіус елемента** Параметр: значення в пікселях
.boxsizing	Обчислення блочної моделі без врахування внутрішніх відступів
.zero_m(@margin:0px; @padding:0px)	Обнулення відступів зовнішніх і внутрішніх
.uls(@liststyle:none;)	Обнулення відступів списків
.opacity(@opacity:1)	**Прозорість елемента** Параметр: @opacity задається від 1 до 100 % з кроком в 1
.shadows(@shadow)	**Тіні блочного елемента сторінки** Параметр: @shadow: глобальна змінна перевизначається у файлі локальних змінних
.text_shadows(@txt_shadow)	**Тіні тектсів** Параметр: @shadow: глобальна змінна перевизначається у файлі локальних змінних
.ellipsis	Додавання «…» засобами CSS для текстів з фізично заданим максимальним одним рядком
.img_cont_proportion (@width:100%; @prop:0.67)	**Обчислення гнучкого контейнера для картинок (адаптивність)**

	Параметр :@width: ширина; @prop: коефіцієнт пропорцій ширина/висота
.image_center	Кросбраузерне центрування картинок пропорційно розмірам блоків в автоматичному режимі
.animate(@all:all, @time:0.5s, @efect:ease, @delay:0s)	**Анімація поведінки елемента** Параметр @all: властивість яку анімуємо, @time:час сек., @efect:ефект, @delay:затримка сек.
.animate_transform(@value:rotateY(360deg))	**Поворот блока** Параметр @value: кут в градусах
.animate_transition (@value:all 0.5s ease 0s)	**Перетворення блока** Параметр @value: властивість
.scale (@factor)	**Масштабування** Параметр @factor: перевизанчається локальною змінною
.keyframes(@name; @arguments) { @-moz-keyframes @name { @arguments(); } @-webkit-keyframes @name { @arguments(); } @keyframes @name { @arguments(); } }	**Ключові кадри + анімація** Параметр @name: назва анімації; @arguments аргументи які приймаємо (набір властивостей)

Таким чином застосування бібліотеки компонентів для створення UI компонентів дозволяє гнучко згенерувати потрібний інтерфейс у автоматичному режимі, а засоби автоматизованого управління графічною інформацією дозволяють відображати графічну інформацію без спотворень пропорційно вписуючи її у передбачене розташування.

3.3 Приклад використання бібліотеки UI компонент

Для того, щоб мати змогу використовувати спроектовану бібліотеку компонентів необхідно клонувати проект з репозиторію GIT або завантажити та розархіувати ядро у обрану теку. Після цього необхідно, якщо немає, то встановити Node.js як головне середовище виконання GULP.Далі необхідно здійснити базове налаштування бібліотеки, а саме встановити шлях (якщо відмінні) до source та вихідних фалів у файлі settings.js (Додаток А). Наступним кроком є налаштування використовуваних модулів та головного файлу CSS. Для цього у фалйі scss/css.scss (рисунок 3.5) необхідно розкоментувати майбутні для використання модулі.

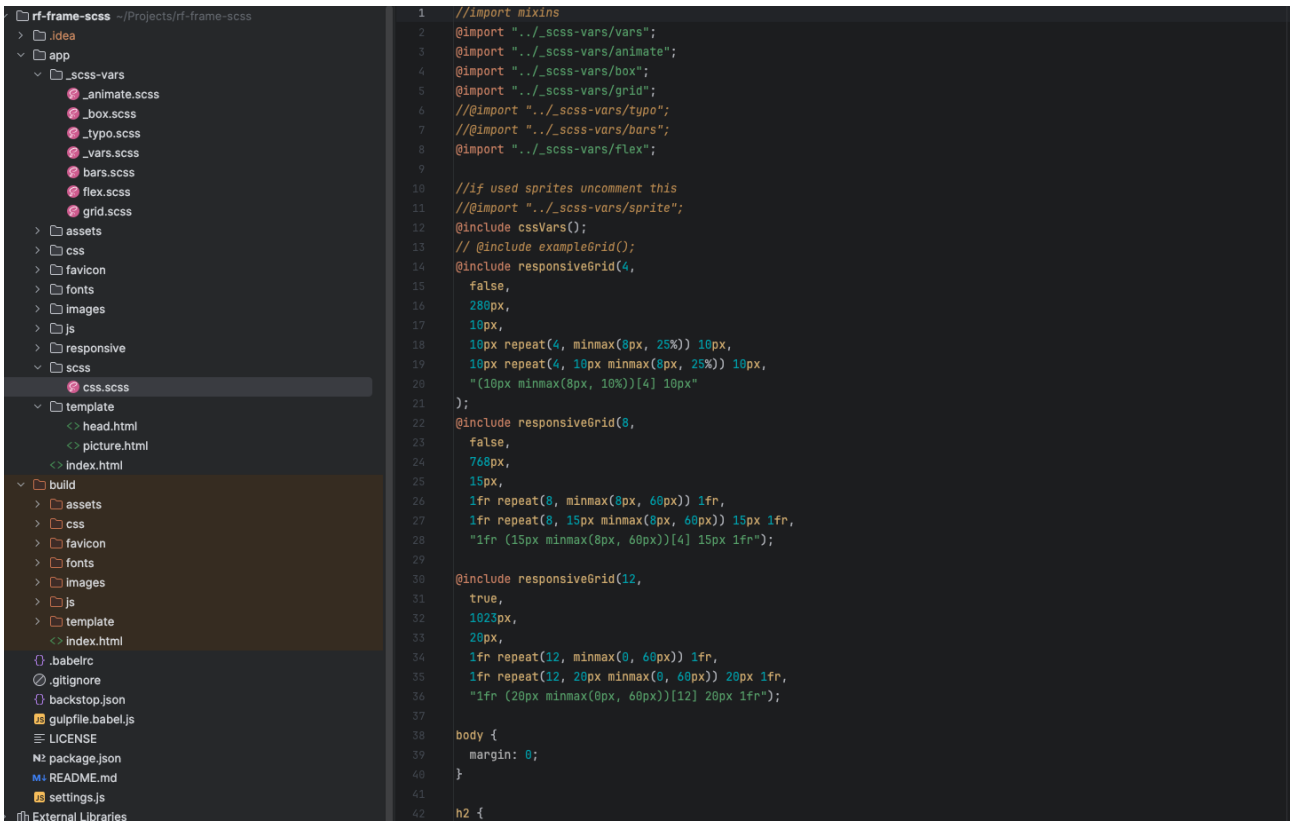

Рисунок 3.5 - Налаштування модулів та сіток бібліотеки компонентів

Після налаштувань компонентів можна переходити до програмування свої UI елементів. Для тестового запуску обрано генерацію компонентів сітки з візуальним відображенням ліній легенди для зручної навігації, та тестового макету, який симулює стандартний декілька колонковий макет.

Для цього створено HTML-файл у який розміщено функціональні UI-елементи макету сайту:

```
<article class="content">Content</article>
   @@include('head.html')
   <!-- @@loop('PATH TO FILE', 'PATH TO JSON DATA FILE') -->
   <nav class="menu">Menu</nav>
   <section class="banner">Banner</section>
   <section class="news">News</section>
   <section class="image">@@include('picture.html',
{"name":"example"})</section>
   <footer class="footer">Footer</footer>
```

Тут приведений фрагмент коду відповідає стандартизоване семантичне відображення елементі типового веб сайту де є. шапка, підвал, блок навігації, секції новин, банеру та окремо винесений модуль-генератор UI компоненту картинки, що дозволить генерувати семантичний компонент картинок .

Для демонстрації можливості генератора сіток функціональні компоненти сайту обгорнуто в UI -елемент – grid, який буде створено автоматично.

Для демонстрації генерації FLEX-сіток додано HTML код, який буде відповідним чином трансформовано у компонент UI:

```
    <section class="flex-container-fluid flex">
        <div class="flex-row">
            <div class="flex-col-6 content-container ">
                <div class="content-body">
                </div>
            </div>
            <div class="flex-col-3 content-container">
                <div class="content-body"></div>
            </div>
            <div class="flex-col-2 content-container">
                <div class="content-body"></div>
            </div>
            <div class="flex-col content-container">
                <div class="content-body"></div>
            </div>
        </div>
    </section>
    <section class="flex-container flex">
        <div class="flex-row">
            <div class="flex-col-12 flex-col-small-6 content-container
">
                <div class="content-body"></div>
            </div>
            <div class="flex-col content-container">
                <div class="content-body"></div>
            </div>
            <div class="flex-col content-container">
                <div class="content-body"></div>
            </div>
            <div class="flex-col content-container">
                <div class="content-body">

                </div>
            </div>
        </div>
    </section>
```

Створивши тестовий файл, що складається з HTML та запустивши збірку буде згенеровано теку *build* з основними компонентами, які так і залишились неповними, оскільки тут немає ніяких стилів, але сам HTML вже доповнено кодом, який виводить на екран семантично правильний тег picture, де розміри картинки, кількість підготованих зображень, а також картинка для сумісності

вже є згенеровані та розміщені у теку *build/assets*. У теці app сама картинка знаходиться фізично у теці *app/responsive*.

```
    <main class="grid">
        <article class="content">Content</article>
        <section class="header">
        Header
</section>
        <!-- @@loop('PATH TO FILE', 'PATH TO JSON DATA FILE') -->
        <nav class="menu">Menu</nav>
        <section class="banner">Banner</section>
        <section class="news">News</section>
        <section class="image"><picture>
    <source srcset="assets/example-320.webp 1x, assets/example-320.webp
2x" media="(max-width: 320px)" type="image/webp">
    <source srcset="assets/example-480.webp 1x, assets/example-480.webp
2x" media="(max-width: 480px)" type="image/webp">
    <source srcset="assets/example-800.webp 1x, assets/example-800.webp
2x" media="(min-width: 800px)" type="image/webp">
    <img src="assets/example.jpg" alt>
</picture></section>
        <footer class="footer">Footer</footer>
    </main>
```

Для того, щоб компоненти стали повноцінними та їх можна було перевикористовувати у проекті необхідно додати стильове наповнення (Додаток В), яке повинно інкапсулювати всю логіку кожного компонента в css. Оскільки в тестовій збірці використовується обмежений набір компонент з метою кращого ілюстрування можливостей бібліотеки стилі UI елемента одразу винесено в головний CSS:

```
.header {
  @include grid-column(1, 15, 1, 28);
  @include grid-row(1, 2);
  background-color: violet;
}
.menu {
  @include grid-column(10, 14, 19, 26);
  @include grid-row(2, 7);
  background: red;
}
.banner {
  @include grid-column(2, 8, 5, 14);
  @include grid-row(2, 3, 3, 4);
  background-color: green;
}
.news {
  @include grid-column(8, 10, 15, 18);
  @include grid-row(2, 3, 3, 4);
  @include var(background-color, red);
}
```

```
.content {
  @include grid-column(2, 10, 5, 18);
  @include grid-row(4, 6, 5, 7);
  background-color: violet;
}
.image {
  @include grid-column(2, 13, 5, 26);
  @include grid-row(6, 7, 7, 8);
  padding: 0;
  img {
    width     : 100%;
    height    : auto;
    object-fit: cover;
  }
  picture {
    width  : 300px;
    display: inline-block;
  }
}
.footer {
  @include grid-column(1, 15, 1, 28);
  @include grid-row(7, 8, 8, 9);
  background-color: black;
  color           : white;
}

.flex {
  border: 1px dashed gray;
}

.content-container {
  border: 1px dotted lightgrey;
}
.content-body {
  border          : 1px dotted lightskyblue;
  height          : 100px;
  background-color: lightslategray;
}
```

Тут, наприклад UI-компонент *header* формується з компоненту сітки бібілотеки:

```
.header {
  @include grid-column(1, 15, 1, 28);
  @include grid-row(1, 2);
  background-color: violet;
}
```

Та буде трансформований до виду CSS вавтоматичному режимі:

```
.header {
  -ms-grid-column: 1;
  -ms-grid-column-span: 27;
  grid-column: 1 / 15;
  -ms-grid-row: 1;
  -ms-grid-row-span: 1;
  grid-row: 1 / 2;
  background-color: violet;
}
```

Після того, коли додано всі SCSS стилі до кожного UI-компонента необхідно запустити консольну команду у корені проекту

```
Npm run gulp
```

Після чого буде показано хід виконання (рисунок 3.6) та відбудеться автоматичне відкриття браузера у якому буде відображено згенерований документ, який складається з UI-компонентів (рисунок 3.7)

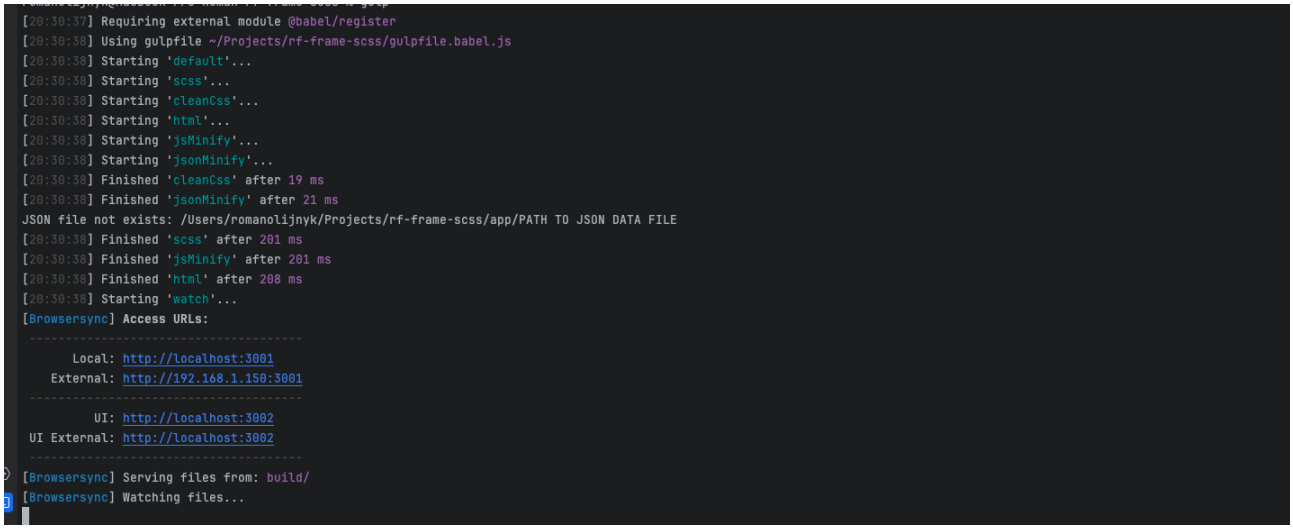

Рисунок 3.6 - Генерація компонентів у консолі

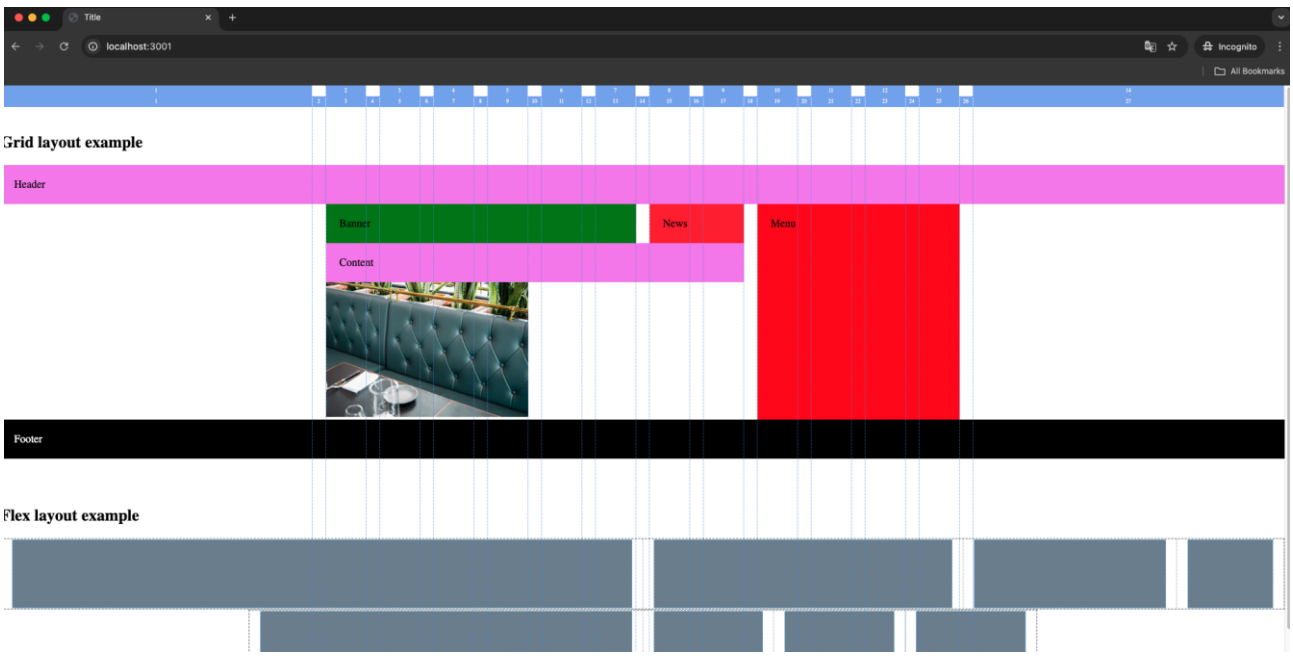

Рисунок 3.7 - Згенерований веб-сайт

ВИСНОВКИ

У ході проведеного дослідження були розглянуті сучасні підходи до проєктування та реалізації бібліотек UI-компонентів з використанням SCSS та task runners, таких як Gulp. Підтверджено, що впровадження компонентного підходу є одним із ключових факторів для досягнення ефективності, гнучкості та масштабованості розробки веб-додатків. Основним завданням цієї роботи стало створення універсальної бібліотеки UI-компонентів, яка спрощує процес розробки користувацьких інтерфейсів і відповідає сучасним вимогам адаптивності та автоматизації.

Аналіз процесу розробки інтерфейсів показав, що основою якісного дизайну UI/UX є компонентна архітектура, що враховує принципи повторного використання, ізоляції та розподілу відповідальності. Вона дозволяє створювати модульні інтерфейси, які легко модифікувати, тестувати та інтегрувати в проєкти різної складності. Використання препроцесорів SCSS дозволило значно покращити організацію стилів, впроваджуючи змінні, міксини, вкладені селектори та інші механізми, які спрощують роботу розробників.

Автоматизація через використання Gulp забезпечила оптимізацію процесів компіляції стилів, мінімізації файлів, генерації адаптивних сіток і інтеграції SVG-графіки. Інструменти автоматизації допомагають зменшити рутинні завдання та підвищити продуктивність розробників, водночас забезпечуючи високу якість кінцевого продукту. Також важливим аспектом стало впровадження живого оновлення (Live Reloading), що значно спростило процес налагодження і пришвидшило розробку.

Практична апробація бібліотеки продемонструвала її ефективність і універсальність у реальних умовах. Тестові компоненти, створені на основі запропонованої бібліотеки, забезпечили високу адаптивність, естетичність і функціональність інтерфейсу. Система налаштувань, структура папок і конфігураційний файл дозволяють легко масштабувати проєкт, додаючи нові компоненти або змінюючи наявні.

Таким чином, розроблена бібліотека UI-компонентів може бути корисною не лише для професійних розробників, але й для тих, хто тільки починає працювати в галузі веб-розробки. Її впровадження дозволить скоротити час розробки, стандартизувати підходи до створення інтерфейсів і забезпечити високу якість кінцевого продукту. Подальший розвиток бібліотеки може включати інтеграцію новітніх підходів до адаптивного дизайну та автоматизації, таких як використання CSS Grid і сучасних JavaScript-фреймворків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Frain B. Responsive Web Design with HTML5 and CSS. – Birmingham : Packt Publishing, 2020. – 400 с.
2. Fixate IT. Structuring your Gulp workflow [Електронний ресурс]. – Режим доступу: https://www.fixate.it/blog/structuring-your-gulp-workflow.
3. Бондарчук А. П., Золотухіна О. А. Проектування інтерфейсу користувача : навчальний посібник. – Київ : Державний університет телекомунікацій, 2017. – 110 с.
4. Чемерис Г. Ю. UX/UI дизайн : навчальний посібник для здобувачів ступеня вищої освіти бакалавра спеціальності «Дизайн» освітньо-професійної програми «Графічний дизайн». – Запоріжжя : ЗНУ, 2021. – 290 с.
5. Введення в UX дизайн [Електронний ресурс] / Технологічний інститут Джорджії. – Режим доступу: https://www.coursera.org/learn/user-experience-design.
6. Flexiple. Intro to Frontend Development [Електронний ресурс]. – Режим доступу: https://flexiple.com/frontend/deep-dive.
7. Sinisalo A. Web frontend component quality model : дис. канд. техн. наук. – Turku, 2017. – 83 с.
8. 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. – Stuttgart, 2018. – 11 с.
9. Олійник Р. Проектування структури генератора сіткок для підготовки веб-інтерфейсу // Комп’ютерні технології друкарства. – 2017. – №2. – С. 77–83
10. Шинкарук О., Яшина О., Онишко О. Управління якістю програмних веб-систем засобами розробки // Вісник Хмельницького національного університету. Технічні науки. – 2020. – №6. – С. 39–44.

11. Scott E. SPA Design and Architecture: Understanding Single Page Web Applications. – Shelter Island : Manning Publications, 2015. – 304 с.
12. Flexiple. Intro to Frontend Development [Електронний ресурс]. – Режим доступу: https://flexiple.com/frontend/deep-dive.
13. Sinisalo A. Web frontend component quality model : дис. канд. техн. наук. – Turku, 2017. – 83 с.
14. Mozilla Developer Network. Web Components [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/Web_components.
15. Goldberg J. Learning TypeScript. – Sebastopol : O'Reilly Media, 2022. – 324 с.
16. HTML – Wikipedia [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/HTML.
17. CSS – Wikipedia [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/CSS.
18. CSS-Tricks. A Complete Guide to Grid [Електронний ресурс]. – Режим доступу: https://css-tricks.com/snippets/css/complete-guide-grid/.
19. Mozilla Developer Network. CSS Grid Layout [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_grid_layout.
20. Sass. Documentation [Електронний ресурс]. – Режим доступу: https://sass-lang.com/documentation/.
21. Klauzinski P., Moore J. Mastering JavaScript Single Page Application Development. – Birmingham : Packt Publishing, 2016. – 452 с.
22. Gavrila V., Bajenaru L., Dobre C. Modern Single Page Application Architecture: A Case Study // ResearchGate. – 2019. – Режим доступу: https://www.researchgate.net/publication/334150777_Modern_Single_Page_Application_Architecture_A_Case_Study.
23. Wikipedia. Single-page application [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Single-page_application.

ДОДАТКИ

Додаток А Фрагмент Конфігурації Gulp

```
const settings = {
    build: {
        html: 'build/',
        js: 'build/js/',
        css: 'build/css/',
        cleanCss: 'build/css/',
        img: 'build/images/',
        fonts: 'build/fonts/',
        json: 'build/json/',
        assets: 'build/assets/',
        favicons: 'build/favicons/',
        ico: 'build/',
    },
    src: {
        html: ['app/**/*.html'],
        js: 'app/js/*.js',
        style: ['app/scss/*.scss'],
        cleanCss: ['app/css/*.css'],
        img: 'app/images/*.*',
        sprite_png: 'app/images/sprites/**/*.{png,jpg}',
        sprite_svg: 'app/images/sprites/**/*.svg',
        fonts: 'app/fonts/**/*.*',
        json: 'app/json/*.json',
        assets: 'app/assets/**/*.*',
        responsive: 'app/responsive/**/*.{jpg,png}',
        favicons: 'app/favicons/**/*.*',
        ico: 'app/*.ico',
    },
    responsiveImage: {
        sizes: [
            { width: 320, quality: 40 },
        ]
    },
    htmlInlcudeSettings: {
        prefix: '@@',
        basepath: 'app/template'
    },
    clean: '/build',
    //build raster or/and vectro sprites, in main scss should be
uncommented sprite css
    isSprite_RASTER: false,
    isSprite_VECTOR: false,
    //scss compilation settings
    compress_Css: 'expanded', //'compressed', 'nested', 'expanded',
    legacyGrid: true,
    //browser sync settings
    browser_sync: 'app/**/*.*',
    isProxy: false,//used when have local server instead browsersunc
server
    isProxy_path: 'http://your full URL', //when local server used
instead browsersync
    port: 3001}
```

Додаток Б Фрагмент коду Gulp

```
/ Responsive images generator
const responsiveImages = (done ) => {
  gulp
    .src(settings.src.responsive)
    .pipe(
      sharpResponsive({
        formats: [
          // jpeg
          { width: 640, format: "jpeg", rename: { suffix: "-sm" } },
          { width: 768, format: "jpeg", rename: { suffix: "-md" } },
          { width: 1024, format: "jpeg", rename: { suffix: "-lg" } },
          // webp
          { width: 640, format: "webp", rename: { suffix: "-sm" } },
          { width: 768, format: "webp", rename: { suffix: "-md" } },
          { width: 1024, format: "webp", rename: { suffix: "-lg" } },
        ],
      })
    )
    .pipe(gulp.dest(settings.build.assets));
    done();
};
exports.responsiveImages = responsiveImages;

const scss = () => {
  return gulp
    .src(settings.src.style)
    .pipe(plumber())
    .pipe(sass({ outputStyle: settings.compress_Css }))
    .pipe(sourcemaps.write({ includeContent: false }))
    .pipe(sourcemaps.init({ loadMaps: true }))
    .pipe(autoprefixer({ grid: settings.legacyGrid }))
    .pipe(sourcemaps.write("."))
    .pipe(
      rename({
        suffix: ".min",
      })
    )
    .pipe(gulp.dest(settings.build.css))
    .pipe(browserSync.stream());
};

exports.scss = scss;

//move css form folder css (normalize plugin required css)
const cleanCss = () => {
  return gulp
    .src(settings.src.cleanCss)
    .pipe(plumber())
    .pipe(gulp.dest(settings.build.cleanCss))
    .pipe(browserSync.stream());
};
exports.cleanCss = cleanCss;

//move html and integrate SVG
```

```
const html = () => {
  return (
    gulp
      .src(settings.src.html)
      .pipe(plumber())
      // .pipe(rigger())
      .pipe(fileinclude(settings.htmlInlcudeSettings))
      .pipe(injectSvg(injectSvgOptions))
      .pipe(gulp.dest(settings.build.html))
      .pipe(browserSync.stream())
  );
};
exports.html = html;

//js
const jsMinify = () => {
  return gulp
    .src(settings.src.js)
    .pipe(plumber())
    .pipe(uglify())
    .pipe(
      rename({
        suffix: ".min",
      })
    )
    .pipe(gulp.dest(settings.build.js))
    .pipe(browserSync.stream());
};
exports.jsMinify = jsMinify;

//json
const jsonMinify = () => {
  return gulp
    .src(settings.src.json)
    .pipe(plumber())
    .pipe(jsonminify())
    .pipe(
      rename({
        suffix: ".min",
      })
    )
    .pipe(gulp.dest(settings.build.json));
};
exports.jsonMinify = jsonMinify;

// minify images
const imageMinify = () => {
  return gulp
    .src(settings.src.img)
    .pipe(plumber())
    .pipe(
      imagemin([
        imagemin.gifsicle({ interlaced: true }),
        imagemin.jpegtran({ progressive: true }),
        imagemin.optipng({ optimizationLevel: 5 }),
      ])
    )
    .pipe(gulp.dest(settings.build.img));
```

```
};
exports.imageMinify = imageMinify;

// move addititional files
const assets = () => {
  return gulp
    .src(settings.src.assets)
    .pipe(plumber())
    .pipe(
      imagemin([
        imagemin.gifsicle({ interlaced: true }),
        imagemin.jpegtran({ progressive: true }),
        imagemin.optipng({ optimizationLevel: 5 }),
      ])
    )
    .pipe(gulp.dest(settings.build.assets));
};
exports.assets = assets;

// favicons folder
const favicons = () => {
  return gulp
    .src(settings.src.favicons)
    .pipe(plumber())
    .pipe(gulp.dest(settings.build.favicons))
    .pipe(browserSync.stream());
};
exports.favicons = favicons;

// ico move
const ico = () => {
  return gulp
    .src(settings.src.ico)
    .pipe(plumber())
    .pipe(gulp.dest(settings.build.ico))
    .pipe(browserSync.stream());
};
exports.ico = ico;

// folder fonts move
const fonts = () => {
  return gulp
    .src(settings.src.fonts)
    .pipe(plumber())
    .pipe(gulp.dest(settings.build.fonts))
    .pipe(browserSync.stream());
};
exports.fonts = fonts;

const watch = () => {
  gulp.watch(settings.src.style, scss);
  gulp.watch(settings.src.cleanCss, cleanCss);
  gulp.watch(settings.src.html, html);
  gulp.watch(settings.src.js, jsMinify);
  gulp.watch(settings.src.json, jsonMinify);
  gulp.watch(settings.src.responsive, responsiveImages);
  gulp.watch(settings.src.img, imageMinify);
  gulp.watch(settings.src.assets, assets);
```

```
  gulp.watch(settings.src.favicons, favicons);
  gulp.watch(settings.src.ico, ico);
  gulp.watch(settings.src.fonts, fonts);
  if (!settings.isProxy) {
    return browserSync.init(settings.browser_sync, {
      server: {
        baseDir: settings.build.html,
      },
      port: settings.port,
    });
  } else {
    // if proxy already have server
    return browserSync.init({
      proxy: {
        target: settings.isProxy_path,
      },
      port: settings.port,
    });
  }
};
exports.watch = watch;

exports.default = gulp.series(
  gulp.parallel(scss, cleanCss, html, jsMinify, jsonMinify),
  gulp.parallel(responsiveImages, imageMinify, assets, favicons, ico,
fonts),
  gulp.parallel(watch)
);
```

Додаток В

CSS.SCSS

```
//import mixins
@import "../_scss-vars/vars";
@import "../_scss-vars/animate";
@import "../_scss-vars/box";
@import "../_scss-vars/grid";
//@import "../_scss-vars/typo";
//@import "../_scss-vars/bars";
@import "../_scss-vars/flex";
//if used sprites uncomment this
//@import "../_scss-vars/sprite";
@include cssVars();
// @include exampleGrid();
@include responsiveGrid(4,
  false,
  280px,
  10px,
  10px repeat(4, minmax(8px, 25%)) 10px,
  10px repeat(4, 10px minmax(8px, 25%)) 10px,
  "(10px minmax(8px, 10%))[4] 10px"
);
@include responsiveGrid(8,
  false,
  768px,
  15px,
  1fr repeat(8, minmax(8px, 60px)) 1fr,
  1fr repeat(8, 15px minmax(8px, 60px)) 15px 1fr,
  "1fr (15px minmax(8px, 60px))[4] 15px 1fr");
@include responsiveGrid(12,
  true,
  1023px,
  20px,
  1fr repeat(12, minmax(0, 60px)) 1fr,
  1fr repeat(12, 20px minmax(0, 60px)) 20px 1fr,
  "1fr (20px minmax(0px, 60px))[12] 20px 1fr");
body {
  margin: 0;
}
h2 {
  margin: 70px 0 20px 0;
}
.grid {
  >* {
    padding: 20px;
  }
}
.header {
  @include grid-column(1, 15, 1, 28);
  @include grid-row(1, 2);
  background-color: violet;
}
.menu {
  @include grid-column(10, 14, 19, 26);
  @include grid-row(2, 7);
  background: red;
```

```
}
.banner {
  @include grid-column(2, 8, 5, 14);
  @include grid-row(2, 3, 3, 4);
  background-color: green;
}
.news {
  @include grid-column(8, 10, 15, 18);
  @include grid-row(2, 3, 3, 4);
  @include var(background-color, red);
}
.content {
  @include grid-column(2, 10, 5, 18);
  @include grid-row(4, 6, 5, 7);
  background-color: violet;
}
.image {
  @include grid-column(2, 13, 5, 26);
  @include grid-row(6, 7, 7, 8);
  padding: 0;
  img {
    width     : 100%;
    height    : auto;
    object-fit: cover;
  }
  picture {
    width  : 300px;
    display: inline-block;
  }
}
.footer {
  @include grid-column(1, 15, 1, 28);
  @include grid-row(7, 8, 8, 9);
  background-color: black;
  color           : white;
}
.flex {
  border: 1px dashed gray;
}
.content-container {
  border: 1px dotted lightgrey;
}
.content-body {
  border          : 1px dotted lightskyblue;
  height          : 100px;
  background-color: lightslategray;
}
```