# Архітектура проєкту

## 1. Про застосунок SipBiteUnite

SipBiteUnite є веб-застосунком, призначеним для допомоги користувачам у виборі ідеальних комбінацій пива та страв. Назва проєкту утворена від англійських слів "sip" (ковток), "bite" (укус) та "unite" (об'єднувати), що символізує основну місію платформи – об'єднання світу пива та кулінарії для створення гармонійних смакових композицій.

Крафтова революція принесла на ринок сотні унікальних сортів пива з різноманітними смаковими профілями, проте саме це різноманіття часто створює складнощі для споживачів у пошуку оптимального поєднання напою та їжі. SipBiteUnite вирішує цю проблему шляхом надання системи рекомендацій, що аналізує характеристики різних сортів пива та страв, враховуючи такі параметри як гіркота, міцність, стиль пива, ароматичний профіль та смакові нюанси.

Реалізація проєкту як односторінкового застосунку забезпечує користувачам плавний та інтуїтивно зрозумілий досвід взаємодії з платформою, дозволяючи швидко переглядати каталоги, отримувати рекомендації та зберігати улюблені поєднання.

## 2. Вибір архітектури

### 2.1 Вибір високорівневої архітектури

Для реалізації веб-застосунку SipBiteUnite ми обрали **клієнт-серверну архітектуру з трирівневою організацією**. Цей вибір обумовлений специфікою веб-додатків, де клієнтська частина виконується в браузері користувача, а серверна частина обробляє запити та працює з базою даних.

Трирівнева організація дозволяє чітко розділити відповідальності між компонентами системи. Клієнтський рівень відповідає за інтерфейс користувача, серверний рівень обробляє бізнес-логіку та рекомендації, а рівень бази даних зберігає всю інформацію про пиво, страви та користувачів. Таке розділення дає нам можливість працювати над різними частинами проєкту незалежно, що особливо зручно для команди студентів.

Ми також врахували, що в майбутньому може знадобитися створити мобільний застосунок або надати доступ до нашого API іншим розробникам. Завдяки клієнт-серверній архітектурі бекенд можна буде використовувати з різними клієнтськими додатками без необхідності переписування логіки підбору поєднань.

### 2.2 Вибір бекенд архітектури

Для організації серверної частини застосунку SipBiteUnite ми обрали багатошарову архітектуру (layered architecture) з використанням патернів MVC та Repository. Це рішення обумовлене необхідністю чіткого розділення відповідальностей між різними компонентами бекенду та забезпечення зручності підтримки коду.

Багатошарова архітектура дозволяє нам структурувати код таким чином, щоб кожен шар відповідав за конкретну функціональність і не залежав від деталей реалізації інших шарів. Це особливо важливо для нашого проєкту, оскільки рекомендаційна система може потребувати частих оновлень та вдосконалень алгоритмів, а чітке розділення дозволить вносити зміни без ризику порушити роботу всієї системи. Крім того, така структура полегшує тестування окремих компонентів та розподіл роботи між членами команди.

#### Структура шарів бекенду

Шар представлення (Presentation Layer) містить контролери та роутери, що обробляють HTTP-запити від клієнта. Цей шар відповідає за прийом запитів, первинну валідацію даних та формування

відповідей у форматі JSON. Тут також розміщується middleware для автентифікації користувачів через JWT-токени, обробки помилок та логування запитів. Контролери делегують виконання бізнес-логіки сервісам і не містять складних обчислень чи прямих звернень до бази даних.

Шар бізнес-логіки (Business Logic Layer) є серцевиною нашого застосунку, де реалізовано всі основні функції системи. Сервіс рекомендацій аналізує характеристики пива та страв, порівнює їхні смакові профілі та формує оптимальні поєднання. Сервіс користувачів управляє реєстрацією, автентифікацією та персоналізацією рекомендацій на основі історії вподобань. Сервіс каталогів забезпечує пошук, фільтрацію та сортування інформації про пиво та страви за різними параметрами. Всі сервіси працюють з даними через репозиторії, не знаючи деталей їхнього зберігання.

Шар доступу до даних (Data Access Layer) реалізований через патерн Repository, що надає уніфікований інтерфейс для операцій з базою даних. Репозиторії інкапсулюють всі SQL-запити та деталі взаємодії з PostgreSQL, надаючи сервісам простий об'єктно-орієнтований API для роботи з даними. Ми використовуємо ORM-бібліотеку для автоматичної генерації запитів та управління міграціями схеми бази даних, що спрощує розробку та зменшує кількість помилок.

Шар моделей (Model Layer) визначає структуру даних та відносини між сутностями предметної області. Моделі описують пиво, страви, користувачів, їхні характеристики та зв'язки між ними. Цей шар використовується всіма іншими шарами для роботи з даними в типізованому вигляді.

Таке розділення дозволяє нам легко тестувати кожен шар окремо, швидко вносити зміни в одну частину системи без впливу на інші, та розподіляти завдання між учасниками команди за їхньою спеціалізацією. Наприклад, один розробник може працювати над вдосконаленням алгоритму рекомендацій у сервісному шарі, поки інший оптимізує запити до бази даних у шарі репозиторіїв.

## 2.3 Архітектура клієнтської частини

Для реалізації клієнтської частини застосунку SipBiteUnite ми обрали **компонентну архітектуру** на основі сучасного JavaScript-фреймворку з використанням патерну **MVVM** (Model-View-ViewModel). Це рішення обумовлене вибором SPA-підходу та необхідністю створення інтерактивного, відгукливого інтерфейсу з ефективним управлінням станом застосунку.

Компонентна архітектура дозволяє розбити інтерфейс на невеликі, незалежні частини, кожна з яких відповідає за конкретну функціональність. Це спрощує розробку, оскільки різні члени команди можуть працювати над різними компонентами паралельно, а також полегшує повторне використання коду. Наприклад, компонент картки пива може використовуватися як у каталозі, так і в результатах рекомендацій. Крім того, така структура робить додаток більш підтримуваним, оскільки зміни в одному компоненті не впливають на інші.

### Структура клієнтської архітектури

**Шар представлення (View Layer)** містить React або Vue компоненти, що відповідають за візуалізацію даних та обробку користувацьких подій. Компоненти поділяються на презентаційні (dumb components), які лише відображають дані та генерують події, та контейнерні (smart components), які управляють станом та логікою. Презентаційні компоненти включають картки пива, форми фільтрів, кнопки та інші UI-елементи. Контейнерні компоненти координують роботу презентаційних компонентів, отримують дані зі store та викликають необхідні дії.

**Шар управління станом (State Management Layer)** реалізований через централізоване сховище стану (Redux для React або Pinia для Vue), що зберігає всі дані застосунку в одному місці. Це включає інформацію про поточного користувача, каталог пива та страв, результати рекомендацій, стан фільтрів та завантажень.

Централізоване управління станом дозволяє уникнути проблем з синхронізацією даних між різними компонентами та полегшує відладку, оскільки весь стан застосунку можна легко інспектувати.

**Шар бізнес-логіки (Business Logic Layer)** містить actions та thunks (для Redux) або actions (для Pinia), що інкапсулюють логіку взаємодії з бекендом та обробку складних операцій. Тут реалізовані функції для відправки запитів на отримання рекомендацій, збереження улюблених поєднань, фільтрації каталогів та автентифікації користувачів. Цей шар також обробляє помилки та керує станом завантаження даних.

**Шар комунікації з API (API Layer)** забезпечує взаємодію з бекендом через HTTP-запити. Тут знаходяться модулі, що інкапсулюють всі виклики до REST API, обробку токенів автентифікації, перехоплення помилок та retry-логіку. Використання окремого шару для API дозволяє централізовано керувати конфігурацією запитів, headers та базовими URL, а також легко замінити реалізацію комунікації при необхідності.

**Шар маршрутизації (Routing Layer)** управляє навігацією між різними сторінками SPA без перезавантаження застосунку. React Router або Vue Router відповідають за відображення потрібних компонентів залежно від URL, захист приватних роутів (наприклад, сторінки профілю доступні лише авторизованим користувачам) та передачу параметрів між сторінками.

**Шар утиліт та хелперів (Utils Layer)** містить допоміжні функції для форматування даних, валідації форм, роботи з датами та іншу загальну логіку, що використовується в різних частинах застосунку. Винесення такої логіки в окремий шар дозволяє уникнути дублювання коду та спрощує тестування.

Така архітектура клієнтської частини забезпечує чітке розділення відповідальностей, де кожен шар має конкретне призначення. Компоненти зосереджені на візуалізації, store керує станом, бізнес-логіка обробляє складні операції, а API-шар відповідає за комунікацію з сервером. Це робить код зрозумілим, тестованим та

легким у підтримці, що критично важливо для успішної розробки студентського проєкту.