Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Аналіз сучасних підходів до проектування архітектури веб застосунків на прикладі сервісу опитування

Текстова частина до курсової роботи
за спеціальністю „Комп'ютерні науки" 6.050103

Керівник курсової роботи
ст.викладач Вовк Н.Є
_____
(підпис)
"____" _____ 2020 р.

Виконав студент
Федюченко М.І
"____" _____ 2020 р.

Київ 2020

ЗМІСТ

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри мультімедійних систем,
Доцент., к. ф.-м. н. О.П.Жижерун
_____
(підпис)
«____»_____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Федюченку Михайлу Ігоровичу факультету інформатики 3-го курсу
ТЕМА Аналіз сучасних підходів до проектування архітектури веб застосунків

на прикладі компоненту користувача для сервісу опитувань

Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. Основні типи архітектур сучасних веб додатків
6. SPA архітектура та найкращі практики
7. Огляд створеного додатку та використаних практик
8. Висновки
9. Список використаної джерел

Дата видачі „___” _____ 2020 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Календарний план виконання роботи

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи	15.10.2019	
2.	Огляд технічної літератури за темою роботи	30.10.2019	
3.	Виконання аналізу архітектурних підходів побудови веб-додатків	15.02.2020	
4.	Написання першого розділу	20.02.2020	
5.	Аналіз найкращих практик при побудові односторінкового додатку	20.03.2020	
6.	Написання другого розділу	30.03.2020	
7.	Створення веб-застосунку для опитування студентів на основі SPA архітектури	01.04.2020	
8.	Написання останнього розділу	09.04.2020	
9.	Корегування роботи згідно із зауваженнями керівника	10.05.2020	

Студент Федюченко М.І.

Керівник Вовк Н.Є.

“_____”_____

Анотація

У роботі розглянуто сучасні підходи до побудови веб-застосунків, детально розглянуто SPA-підхід та його переваги. Розроблено веб застосунок на основі SPA-підходу та з використанням досліджених практик, що забезпечують гнучкість застосунку, масштабованість, ефективність та багаторазовість використання.

ВСТУП

З розвитком людства, об’єми створеної людиною інформації збільшуються і для отримання її виникає потреба створення нових джерел інформації, чи покращення-модифікації вже існуючих.

Веб-браузер – це найпоширеніша у світі програма, потрібність котрої є беззаперечною. Певне кожен користувався веб-ресурсами, щоб знайти потрібний матеріал, чи переглядав відео, фотографії та іншу інформацію представлену в мережі інтернет. У цьому нам допомагають веб застосунки, котрі не потребують встановлення, є доступними для будь-якої платформи через більшість браузерів, а також надають можливість обробки великих об’ємів інформації, що здійснюється за лічені секунди у хмарних сховищах з відображенням користувачу без використання потужностей використовуваного девайсу.

Це викликає попит на створення ефективних веб-застосунків, з можливістю їх багаторазового використання, масштабування при виникненні нових потреб та гнучкості використання. Усе це залежить від архітектурної складової та принципів їх побудови.

Актуальність правильної архітектури тільки зростає з бажанням людини ще більше автоматизувати процеси власної діяльності, тому зараз складається тенденція на використання практик, що призводять до загальної ефективності в розробці додатків, технічному обслуговуванні коду та часу розробки.

Метою курсової роботи є визначення основних сучасних підходів до побудови архітектури веб-додатків та з’ясування найкращих практик при створенні односторінкового застосунку.

Текстова частина курсової роботи складається з трьох розділів.

У першому розділі розглядаються основні сучасні архітектурні підходи при побудові веб застосунків, а також їх порівняння. Наведено огляд архітектурних моделей та технологій для їх створення.

У другому розділі окреслюються шляхи підвищення ефективності при побудові SPA веб-застосунку, обґрунтовуються найкращі з розглянутих практик.

Третій розділ присвячено опису архітектурної побудови SPA веб-додатку на основі сервісу опитування.

Постановка задачі.

1. Виконати аналіз архітектурних підходів при побудові веб-додатків, зокрема:
   - на MPA архітектурі
   - на SPA архітектурі
   - на Microservice архітектурі
   - на Serverless архітектурі
2. Виконати порівняльний аналіз архітектурних підходів.
3. Виконати детальне дослідження практик та архітектурних моделей при побудові клієнтського компоненту веб-додатку на основі SPA-архітектури.
4. Розробити аргументовану структуру клієнтського компоненту з урахуванням досліджених практик та моделей.
5. Зробити висновки щодо актуальності та доречності використаних архітектурних практик.

1 ОСНОВНІ АРХІТЕКТУРНІ ПІДХОДИ ПРИ ПОБУДОВІ ВЕБ-ЗАСТОСУНКІВ

Перш за все важливо зрозуміти, що таке веб-додаток і в чому його особливість. Веб-додатком називають тип програми побудованої на основі клієнт-серверної архітектури і його особливість в тому, що сам веб-додаток розташований і виконується на віддаленому сервері, тоді як клієнт отримує тільки результати власної роботи.

Робота такої програми заснована на опрацюванні запитів користувача та надсилання результатів клієнту через мережу інтернет, тому і не залежить від операційної системи певного користувача. Зазвичай, за відображення результатів запитів та отримання вхідних даних для подальшого відправлення на сервер відповідає браузер (*Google Chrome*, *Opera*, *Mazila*, тощо).

Практично будь-який веб-додаток складається з двох основних компонентів:

- клієнтська компонента – код, що знаходиться в браузері та реагує на певний ввід користувача, є представленням функціональності веб- застосунку ,користувач має повний доступ до цієї компоненти.
- серверна компонента – код , що знаходиться на сервері та відповідає на HTTP запити, відповідальний за збереження інформації, складається щонайменше з бази даних та логіки програми, дана компонента недоступна для користувача.

Архітектура веб-додатків описує взаємодію між компонентами що працюють одночасно, базами даних та системами проміжного програмного забезпечення в мережі Інтернеті. Тобто, від того як розподілена логіка програми між серверною та клієнтською компонентами залежить тип архітектури веб-додатку.

1.1 MPA та SPA архітектури

1.1.1 MPA архітектура

Багатосторінковий тип архітектури (*MPA*) – це традиційний тип архітектури для веб застосунків, при якому виконується завантаження усієї сторінки, та перезавантаження на абсолютно нову, якщо користувач взаємодіє із застосунком.

Багатосторінкові програми - це традиційні веб-програми, які завантажують всю сторінку та відображають нову, коли користувач взаємодіє з веб-додатком. Браузер виконує новий запит до сервера при переході на іншу сторінку і знову виконує завантаження всіх ресурсів і навіть тих компонентів, що можуть повторюватися на усіх сторінках.

Відповідно це відображається на продуктивності та швидкості, адже потрібно заново завантажувати ті самі елементи. Основними технологіями для розробки клієнтської компоненти для такого типу архітектури є *HTML*[1] та *CSS*[2].

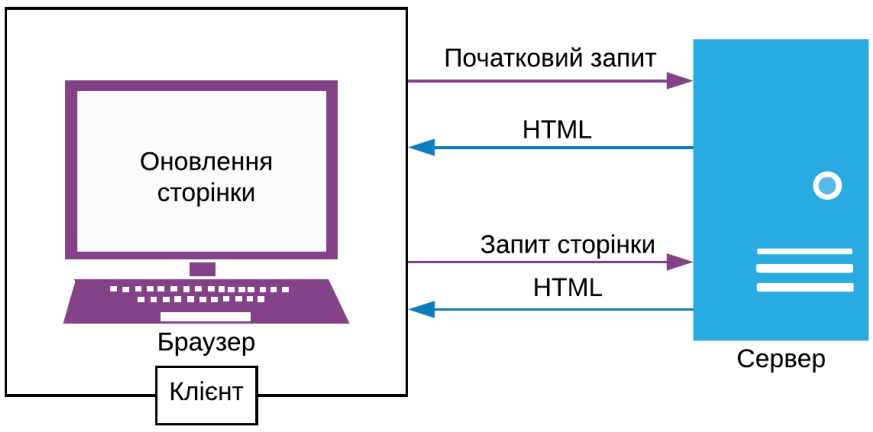


Рисунок 1.1. – Приклад MPA архітектури

З рисунку 1.1 можна помітити, що при доданні нових функціональних можливостей вартість змін буде досить високою. Розробка клієнтської та

[1] мова тегів, якою пишуться гіпертекстові документи для мережі Інтернет
[2] називається набір параметрів форматування, який застосовується до елементів документа, щоб змінити їх зовнішній вигляд.

серверної компоненти дуже пов’язані між собою, що й ускладнює саму розробку.

Перевагами використання даної традиційної архітектури є можливість легкої оптимізації кожної сторінки для пошукових систем, адже існує можливість додання метатегів[3] для будь-якої сторінки.

1.1.2 SPA архітектура

До двохтисячних років розробники шукали ефективний підхід для створення таких веб-застосунків, що були б схожими на програми для настільних комп’ютерів. Були випробовані різні технології, наприклад *Java applets*[4], *Adobe Flash*[5], and *Microsoft Silverlight*[6]. Хоч технології були й різними, однак вони мали одну спільну мету – перенести потужність й можливості стандартної настільної програми у крос-платформене середовище веб-браузерів.

Можливість реалізації даної ідеї почали існувати на початку двохтисячних років, з активної інтеграцією технології AJAX. Котра бере початок з контролю *ActiveX* у браузері *Microsoft’s Internet Explorer*, що використовувався для асинхронного надсилання й отримання даних. Це призвело до того що така контрольна функціональність була офіційно додана до більшості браузерів, як *XMLHttpRequest* (XHR) *API.*

Концепція *SPA* архітектури підняла технологію веб-розробки на новий рівень розширивши сторінкові методи маніпуляції *AJAX* на увесь застосунок.

---

[3] HTML-теги, призначені для надання структурованих метаданих про веб-сторінки.

[4] невеликі програми, написані мовою програмування Java, яка компілюється в байт-код Java та передається користувачам у вигляді байт-коду Java.

[5] мультимедійна та програмна платформа використовувана для авторської розробки векторної графіки, анімації, які можна переглядати в Adobe Flash Player.

[6] застаріла програмою для написання та запуску Інтернет-додатків, подібна до Adobe Flash.

При використанні *SPA* архітектури існує одна повна HTML-сторінка, що складається з багатьох відображень, котрі є частиною *DOM*[7]. Після початкового завантаження сторінки усі необхідні інструменти для створення відображень завантажуються та готові до використання.

За потреби, нове відображення генерується локально у браузері і динамічно додається в DOM за допомогою *JavaScript*.

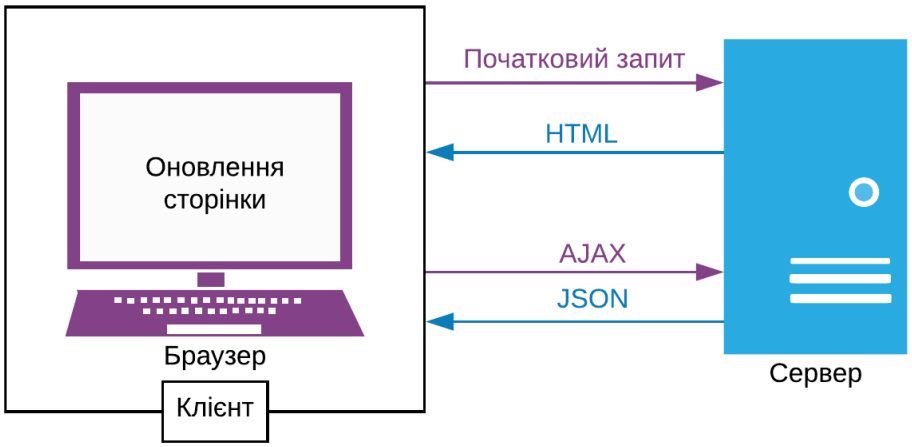


Рисунок 1.2. – приклад SPA архітектури

SPA архітектура передбачає використання декількох підходів для відображення даних із серверної компоненти, наприклад коли відображення генерується повністю на стороні клієнтської компоненти, а дані надсилаються та отримаються тільки під час бізнес-транзакцій, що виконуються асинхронно через *XHR API*.

Зазвичай формат обміну даних предсталяє собою *JSON*[8]. Як і у випадку генерації даних серверною компонентою, вихідний *HTML* має спеціальні блоки для заповнення даними. Але це не повний шаблон *HTML* для повної сторінки, а тільки частина відображення.

[7] специфікація прикладного програмного інтерфейсу для роботи зі структурованими документами.
[8] текстовий формат обміну даними між комп'ютерами.

1.1.3 Порівняння SPA та MPA архітектурних підходів

Проаналізуємо архітектурний підхід на основі MPA для традиційного веб-застосунку.

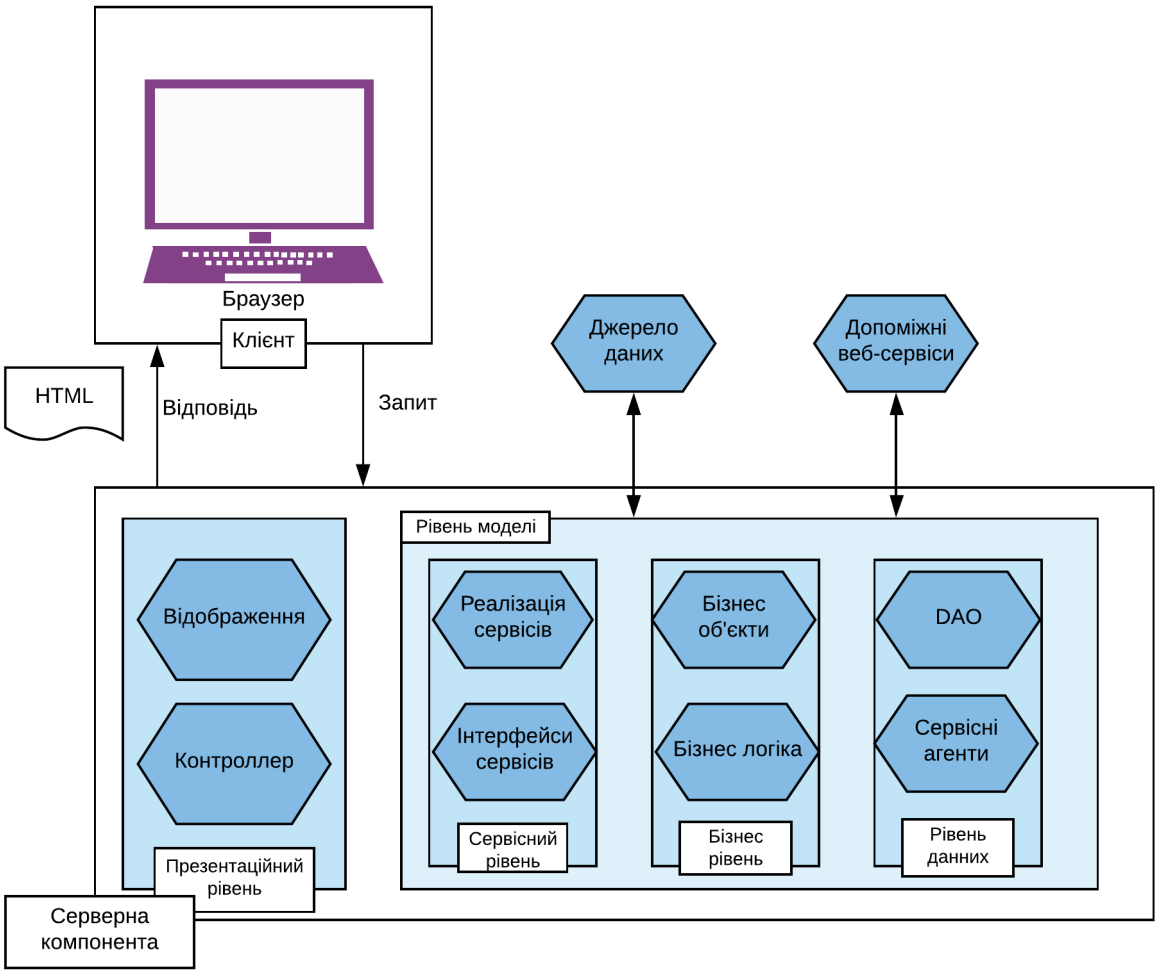


Рисунок 1.3. – приклад MPA для серверної компоненти

Бачимо, що кожний запит на нове відображення призводить до подорожі на сервер. Запит надсилається серверній компоненті, якщо потрібні нові дані.

На стороні серверної компоненти запит перехоплюється контролером, що взаємодіє з рівнем моделі за допомогою сервісного рівня, котрий визначає усі потрібні компоненти для виконання завдання рівня моделі. Дані вилучаються об’єктом доступу до даних (*DAO*[9]), або сервісним агентом. Необхідні зміни виконуються на бізнес рівні. Далі контроль передається назад презентаційному

[9] об'єкт що надає абстрактний інтерфейс до деяких видів баз даних без розкриття деталей бази даних.

рівню, де логіка відображення певним чином формує отримані дані для обраного відображення. Після об'єднання даних з відображенням, новостворене відображення повертається браузеру у якості *HTML* сторінки. Браузер оновлює сторінку для показу нових даних.

SPA підхід майже такий самий, як і традиційний *MPA*. Ключові зміни полягають у наступному: відсутність повного оновлення сторінки браузером, логіка відображення знаходиться в клієнті, а транзакції на серверній компоненті можуть бути представлені лише даними.

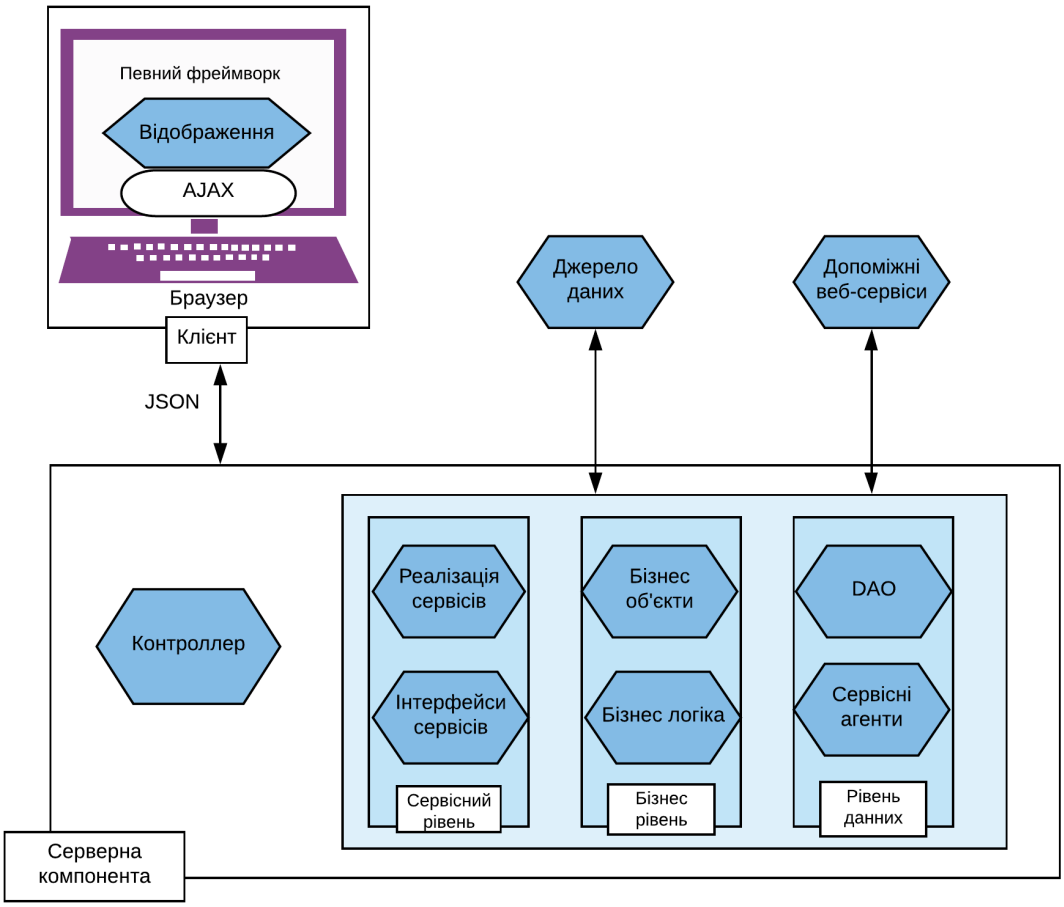


Рисунок 1.4. – приклад SPA для серверної компоненти

З рисунку 1.4. можна побачити, що серверна компонента більше не бере участі у відображенні даних, якщо не здійснюється часткова візуалізація.

Можна визначити такі переваги використання *SPA* архітектури над *MPA*:

1. Легше обслуговування програмного коду – принципи побудови *SPA* веб-застосунків допомагають розділити код на різні сфери, наприклад *JavaScript* код зберігається поза *HTML* окремими частинами. Також зменшується кількість зв’язків між клієнтською компонентою та серверною.
2. Динамічне перемальовування частин екрану – оскільки *SPA* завантажує структуру *HTML* сторінки заздалегідь, для отримання нових сторінки з сервера не потрібно виконувати запит на руйнування наявної.
3. Скорочений часу очікування – чим менше людина витрачає часу на очікування, тим більше шанс, що вона залишиться на сайті й повернеться у майбутньому.
4. Генерація відображення клієнтською компонентою – код , що керує логікою відображення інтерфейсу для користувача зберігається на стороні клієнта замість сервера, що дає перевагу в незалежній підтримці та оновленні обох компонент.
5. Швидкі та ненавантажені транзакції – виконання асинхронних транзакцій роблять тип цієї архітектури дуже швидким, адже після початкового запиту тільки дані надсилаються та отримаються серверною копмонентою. Вміст, що повертається при *MPA* підході включає в собі і *HTML* розмітку.

1.2. Мікро сервісний тип архітектури

Мікро сервісна архітектура – це форма сервісно-орієнтованої архітектури технологія розробки програмного забезпечення, яка структурує додаток як колекцію незалежних мікро-сервісів, що комунікують між собою використовуючи певні механізми, зазвичай *HTTP*[10] чи *REST*[11].

[10] протокол передачі даних
[11] підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів

Кожен мікро-сервіс є самодостатнім і повинен реалізувати конкретну бізнес-потребу, тому його можливо створити й запустити незалежно від інших , навіть на зовсім іншій мові програмування.

Усе більше постачальників послуг (Twitter, eBay, Netflix та Amazon) надають перевагу мікро сервісній архітектурі.

Наприклад, до 2012 року Walmart Canada використовували монолітний тип архітектури, що в піках активного використання сервісу призводило до обробки шести мільйонів сторінок на хвилину, що займало багато часу, тому й зменшувало кількість купівель через сервіс.

Після оновлення архітектурного підходу на мікро сервіси їм вдалося збільшити дохід на двадцять відсотків за рахунок того, що час простою було скорочено і компанія отримала змогу використовувати дешевші сервери, збільшити кількість покупок через мобільні пристрої на дев’яносто вісім відсотків.

1.2.1 Філософія мікро сервісної архітектури

Філософія мікро сервісів дуже близька до філософії сімейства відомих операційних систем *Unix* [12]: “Зробіть одну річ і зробіть її добре”[1].

Архітектура мікро сервісів використовує бібліотеки, однак її головний спосіб розбиття застосунку шляхом поділу його на сервіси.

Перевага незалежності розвертання кожного сервісу зумовлена тим, що при зміні логіки одного з них не потрібно буде виконувати повторне розвертання усіх сервісів програми, а тільки того, що зазнав змін. Даний архітектурний підхід допомагає не порушувати інкапсуляцію компонентів за рахунок використання віддалених викликів сервісами програми.

[12] сімейство багатозадачних комп'ютерних операційних систем

Архітектурний підхід починається з різних типів клієнтів, що можуть бути будь-якими пристроями, котрі намагаються виконувати певні можливості керування, наприклад пошук, налаштування тощо.

Оскільки клієнти не звертаються до сервісів напряму, шлюз API є вхідною точкою, що виконує переадресацію запитів клієнтів у відповідні мікро сервіси.

Першою перевагою використання API шлюзу є те, що ми можемо виконувати зміни над сервісом без попередження клієнта, а також що сервіси можуть використовувати протоколи спілкування не пристосовані для веб середовища.

Існують два основні типи повідомлень для комунікації сервісів між собою:

- синхронні повідомлення – у випадку, якщо клієнт очікує відповіді від сервісу, прийнято використовувати *REST*, адже це розподілене середовище у котрому кожна функціональність представлена відповідним ресурсом для проведення операцій;
- асинхронні повідомлення – у випадку, коли клієнт не очікує відповіді. Зазвичай використовуються такі протоколи, як *AMQP*, *STOMP*, *MQTT*, чия основа повідомлень є чітко визначеною і ці повідомлення мають бути сумісними між реалізаціями.

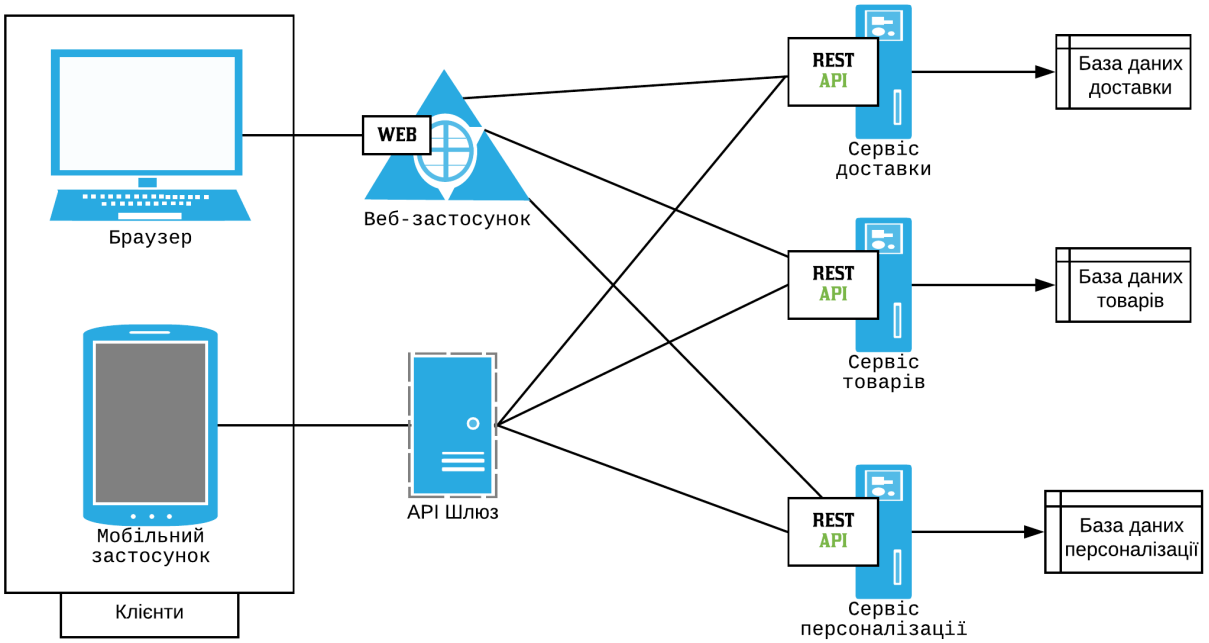


Рисунок 1.5 – приклад мікро сервісної архітектури

Особливість котру ми спостерігаємо на рисунку 1.5 полягає в тому, що кожній бізнес потребі було виділено окремий мікро сервіс, для обробки власних даних та виконання різних функціональних можливостей. Ці мікро сервіси мають власне середовище розгортання для виконання своїх функціональних потреб та власні бази даних для фіксації даних. Усі мікро сервіси спілкуються між собою через сервер, що не зберігаю стан, який є або шиною повідомлень, або *REST* клієнтом. Усі внутрішні точки з'єднані з шлюзом *API*. Отже, кожен, хто підключається до шлюзу *API*, автоматично підключається до повної системи.

1.2.2 Переваги та недоліки мікро сервісної архітектури

Тенденція на мікро сервісну архітектуру набуває популярності, адже в міру збільшення складності програмного забезпечення, можливість розбиття функціональних областей застосунку на колекції незалежних сервісів може принести багато переваг розробникам програмного забезпечення, що передбачають:

1. Масштабованість – мікро сервіси, що складаються з компонентів легко можуть бути інтегровані з іншими програмами чи сервісами через інтерфейси, наприклад *REST*.
2. Прискорена розробка програмного забезпечення – через можливе використання різних мов програмування стає ширшим вибір кола розробників. А також різні команди можуть працювати паралельно над окремими компонентами, без очікування результатів один-одного. Інші ж зацікавлені сторони можуть в цей час вдосконалювати вже реалізовані компоненти.
3. Можливість повторного використання – оскільки мікро сервіси не є одним проєктом і розташовані навколо певних бізнес потреб, то легко можуть бути повторно використаними в інших проєктах, що також і зменшує час розробки.
4. Краще розуміння кодової бази – оскільки кожен сервіс являє собою єдину функціональну область або бізнес потребу, то це покращує подальшу підтримку та налагодження.
5. Ефективна ізоляцій можливих проблем – немає потреби у постійному переході між структурними шарами застосунку, розробники знають, де саме потрібно шукати проблеми, котра потребує вирішення. Якщо зачіпається один сервіс, його можна легко видалити або вирішити без пошкодження інших частин програм

Мікро сервісний тип архітектури, як і будь який інший тип не є ідеальним, виявлено і певні недоліки:

1. Складність інтеграційного тестування – зумовлене можливістю існування компонентів що належать іншим системам, чи середовищам. Адже перед початком тестування потрібне підтвердження кожного залежного сервісу.

2. Комунікація – оскільки канали зв’язку мають бути визначені у певних інтерфейсах, можливе виникнення помилок між мікро сервісами при недотриманню визначеного інтерфейсу.
3. Потреба у належному моніторингу – кожний сервіс залежить від власного *API* та платформи розгортання й для належного моніторингу потрібно слідкувати за багатьма сутностями та мати контроль над усією інфраструктурою.

1.3. Serverless архітектура

Serverless архітектура – це підхід до створення та запуску веб-додатків та сервісів без необхідності керування усією інфраструктурою. Такий підхід включає у себе використання сторонніх послуг “BackEnd as Service ” (*BaaS*), або коду, що контролюється тимчасовими контейнерами на платформі “Function as a Service” (*FaaS*).

При використанні даної архітектури збільшується залежність від постачальників сторонніх послуг.

До 2006 року більшість розробників використовували власні фізичні сервери, розміщені у центрах обробки даних, для розгортання серверної компоненти власних веб-застосунків.

Але в жовтні 2006 року новий відділ Amazon Web Services (AWS) американської компанії Amazon оголосив про запуск нової технології Elastic Compute Cloud (EC2).

Це дозволило стороннім компаніям використовувати EC2, як інфраструктуру для сервісів (*IaaS*). Використання *IaaS* – можна визначити, як інфраструктурний аутсорсинг. EC2 дав змогу орендувати обчислювальну потужність, запускати свої власні серверні застосунки замість того, щоб купляти власне обладнання для розгортання цих застосунків.

1.3.1 Інфраструктурний аутсорсинг

Будь-яка форма використання *IaaS*, або ж інфраструктурний аутсорсинг принаймні частково передбачає ідею економії від масштабу, тобто що виконання тієї самої дії у сукупності дешевше, ніж сума самостійного виконання цих дій незалежно один від одного, завдяки ефективності, що можна використати.

Інфраструктурний аутсорсинг відображає п’ять переваг *IaaS*:

- підвищена гнучкість масштабування – адже стає доступним широкий вибір подібних ресурсів, котрі можна використати, а потім ефективно утилізувати без зайвих залишкових даних.
- швидкість розробки застосунку – скорочення часу на розробку починаючи від концепції до релізу.
- дешевша вартість ресурсів – менша вартість за ті ж самі можливості.
- зменшення ризиків – від суб’єкта використання вимагається менше знань та надається жива операційна підтримка.
- зниження вартості робочої сили – для виконання робіт над інфраструктурою потрібно менше людей та часу.

Однією із найсучасніших форм інфраструктурного аутсорсингу і став Serverless підхід.

1.3.2 Serverless технології

Як і будь-яку інший підхід – *Serverless* поділяються на дві паралельні сфери:

1. Поняття “Serverless” вперше було використано для опису програм, де за управління логікою та станом на сервері відповідають сторонні хмарні-застосунки та сервіси. Зазвичай це програми мобільні додатки, або веб-застосунки побудовані на SPA архітектурі, що використовують масштабну екосистему хмарних баз даних, служб аутентифікації та інших хмарних технологій. Такі типи сервісів і називають “BackEnd as a Service” (*BaaS*).

2. А також *Serverless* – може бути застосунок, де логіка серверної компоненти все ще написана розробником, але на відміну від традиційної архітектури, вона запускається в обчислювальних контейнерах, що не зберігають стан, викликаються при виникненні подій, а також є тимчасовими і керуються сторонніми сервісами. Такий тип називається “Function as a Service” (FaaS).

1.3.2.1 BackEnd as a Service

Сервіси BaaS – це сторонні не вбудовані компоненти, котрі можна додати до програми за допомогою API. Вони прибирають потребу у створені та використанні власного сервера посередника, а також керують компонентами, що зберігають дані від нашого імені. Також такі сервіси дозволяють покластися на сторонню, вже реалізовану логіку. Наприклад багато застосунків, в котрих наявна автентифікація мають схожу логіку, тому для збереження часу та ресурсів використовуються такі продукти, як “OAuth”, чи “Amazon`s Cognito”, що дозволяють веб-додаткам мати аутентифікацію, але без потреби створення коду для реалізації цієї функції.

BaaS дає змогу розробникам зосередитися на створенні клієнтської компоненти, що як результат дозволяє швидше створювати та запускати нові веб-додатки.

1.3.2.2 Function as a Service

FaaS - це новий спосіб створення та розгортання програмного забезпечення серверної компоненти, що орієнтований на розгортання окремих функцій, чи операцій.

При стандартному розгортанні серверної компоненти, зазвичай починають з хост-екзмепляром і розгортають додаток у хості, як на рисунку 1.6.

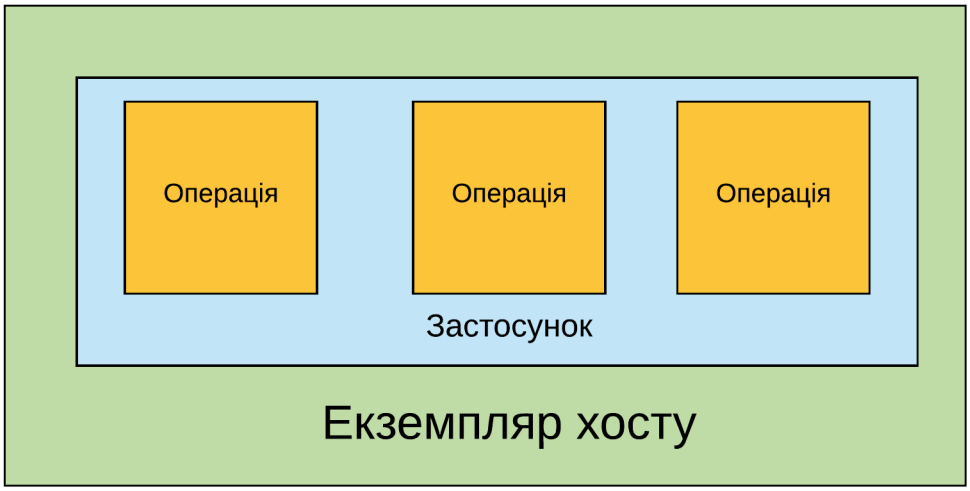


Рисунок 1.6 – приклад стандартного розгортання

У середовищі FaaS ми завантажуємо код для нашої функції провайдеру FaaS, а вже сам провайдер виконує усе інше для забезпечення ресурсів, керування процесами, тощо, як на рисунку 1.7.

Тобто йде зосередження на окремих операція, чи функціях, що відображають логіку програми. FaaS, як підхід не вимагає коду для певного фреймворку, чи бібліотеки.

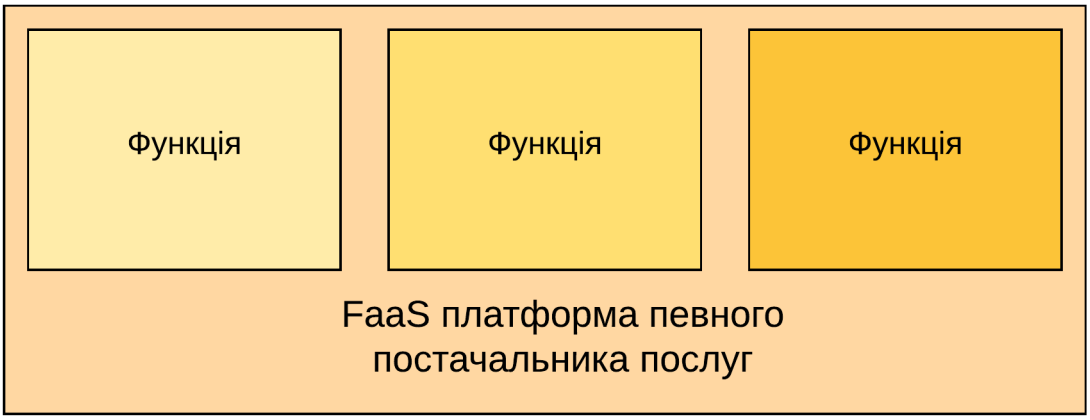


Рисунок 1.7. – приклад розгортання для FaaS

Функції FaaS - це звичайні програми, якщо йдеться про мову програмування та навколишнє середовище. Платформа FaaS налаштована для прослуховування конкретної події для кожної операції. Коли ця подія відбувається, платформа постачальника створює лямбда-функцію, а потім викликає її при запуску події.

Наприклад в Amazon AWS такими типами подій є оновлення файлів у “S3”, час запланованого завдання. Більшість провайдерів дозволяють запуск функцій у відповідь на вхідні HTTP-запити.

На прикладі того ж самого Amazon AWS за таку поведінку відповідає шлюз API. FaaS функції, як правило, обмежені в часі для реагування на відповідь. Тобто для певних класів довготривалих завдань виникає потреба створення кількох скоординованих функцій.

1.3.2.3 Приклад використання FaaS та BaaS

Ключовим моментом, що поєднує FaaS та BaaS та відносить їх до Serverless архітектури є те, що немає потреби керувати процесами у серверної компоненти. Уся логіка, котру ми можемо закодувати власноруч, або

інтегрувати у сервіс стороннього постачальника – працює в гнучкому середовищі, як і стан, що зберігається в цьому ж середовищі.

На прикладі типового веб-застосунку для замовлення товарів можна зазначити основні переваги використання Serverless підходу.

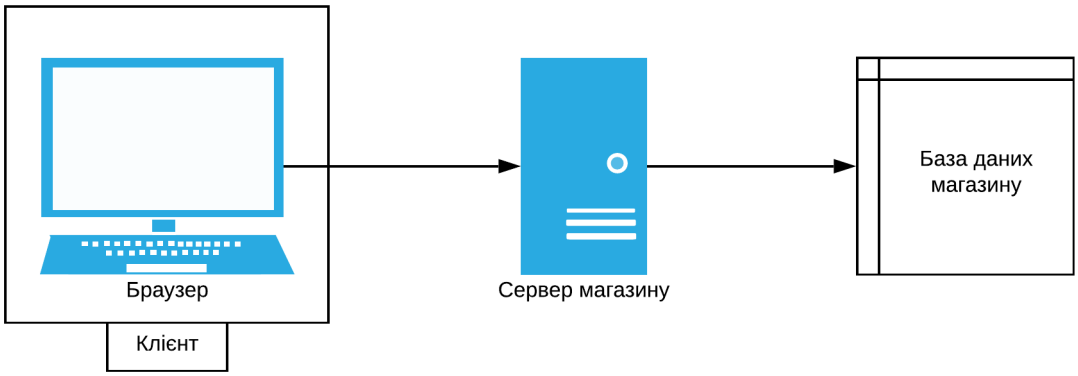


Рисунок 1.8. – приклад стандартної інфраструктури

Використовуючи стандартну трьох-рівневу архітектуру на рисунку 1.8 можна побачити, що велика частина логіки – пошук, навігація, транзакції, керування користувачами – реалізовано серверною компонентною, що робить клієнтську компоненту дуже несвідомою.

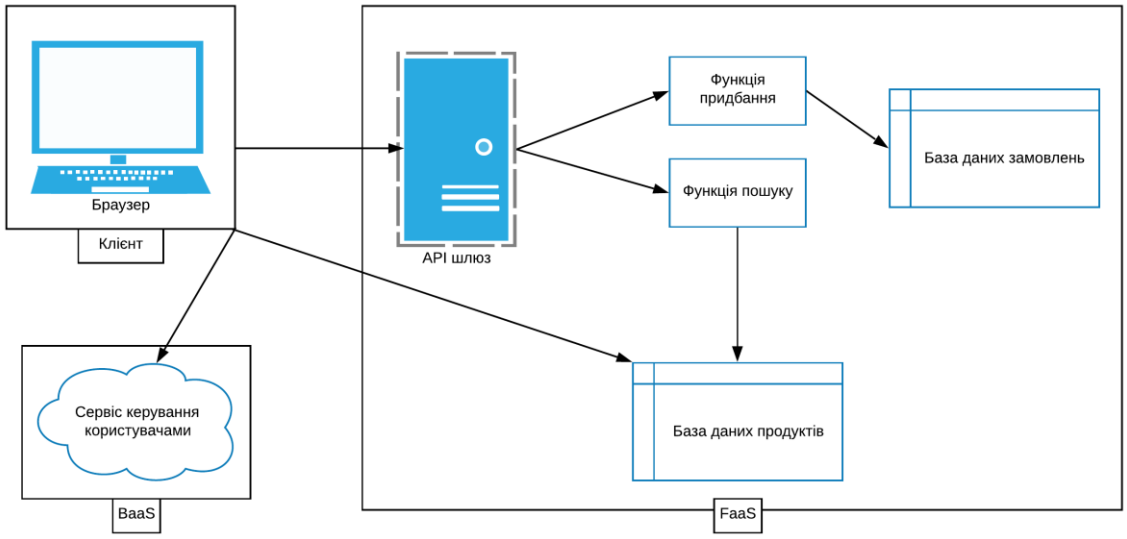


Рисунок 1.9. –приклад безсерверної інфрастурктури з використанням FaaS та BaaS

На рисунку 1.9 використовуючи *Serverless* архітектуру, ми замінили логіку керування користувачами на сторонній сервіс *BaaS*.

Через використання іншого сервісу *BaaS* ми дозволяємо клієнту прямий доступ до підмножини бази даних, що повністю розміщена на сторонньому сервісі. Можна помітити, що тепер певна логіка, котра знаходилася на сервері магазину тепер знаходиться в межах клієнта, що дозволяє відстежувати сеанси користувача, конвертувати отримані дані з бази даних для представлення клієнту.

Тепер клієнт може мати архітектуру односторінкового веб-застосунку. Оскільки ми хочемо зберегти деякий функціонал для пошуку товару на серверній компоненті, але без постійно працюючого сервера ми імплементуємо *FaaS* функцію, яка дає відповіді на *HTTP* запити через шлюз *API*.

Після цього новостворена *FaaS* функція як і клієнт може використовувати базу даних продуктів. *FaaS* функцію, що реалізує придбання товару ми теж інтегруємо на серверну компоненту, а не на клієнтську в міркуваннях безпеки.

Підкреслити те, що у початковій версії усім потоком керування, безпекою керував серверний застосунок. У *Serverless* версії уся логіка розбита між сервісами, що орієнтовані на власні бізнес потреби – можна помітити схожість ідеї з мікро сервісною архітектурою.

2 КЛІЄНТСЬКИЙ КОМПОНЕНТ ПРИ ВИКОРИСТАННІ SPA АРХІТЕКТУРИ

Клієнтська компонента побудована на основі *SPA* архітектури пропонує набагато більше можливостей, ніж традиційний інтерфейс користувача для веб-сайту. Адже якісно написаний *SPA*-клієнт – це практично настільна програма.

Розробники що мали бачення успішного переходу від традиційних *MPA* веб-застосунків до *SPA*, зрозуміли, що застарівші практики та структури повинні змінюватися.

Фокус завдань перейшов на архітектурний талант, дисципліну та тестування на стороні клієнтської компоненти.

2.1 Головна оболонка

Побудова клієнтської компоненти *SPA* починається зі створення початкового файлу *HTML* або оболонки.

Цей файл *HTML* завантажується тільки один раз і служить вихідною точкою для решти програми. Браузер виконує це завантаження.

Подальші частини програми завантажуються динамічно та незалежно від оболонки, без повного перезавантаження сторінки.

Модуль оболонки надає форму та структуру веб-додатку. Можливі модулі взаємодії додаються до оболонки за допомогою API. Вона ж погоджує модулі функцій з бізнес-логікою та універсальними інтерфейсами браузера, такими як *URI*[13] або *Cookies*[14]. Наприклад, коли користувач своїми діями змінює стан програми, оболонка координує внесені зміни.

[13] компактний рядок літер, який однозначно ідентифікує окремий абстрактний чи фізичний ресурс.
[14] поняття, яке використовується для опису інформації у вигляді текстових або бінарних даних, отриманих від веб-сайту на веб-сервері

Зазвичай, на початку розробки оболонка має мінімальну структуру і складається з одного пустого тегу *DIV*[15], що тримає у собі решту програми. Початковий контейнер *DIV* може містити дочірні контейнери, що візуально поділяють екран на логічні зони як це продемонстровано на рисунку 2.1.

Зони допомагають розділити видиму ділянку на керовані фрагменти вмісту. Але наприклад *React* виконує виправлення *DOM* замість заміни конкретних зон.

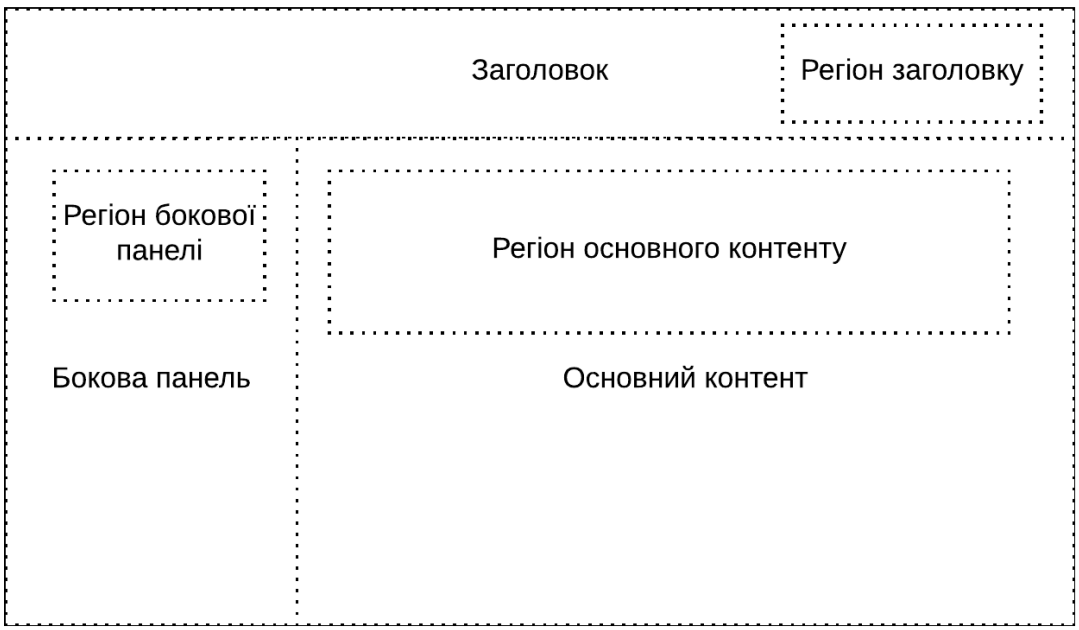


Рисунок 2.1 – приклад поділу на зони

Сторінками веб-додатку – є незалежні розділи, що ще називають відображеннями. Відображення – це фактично частина програми з котрою взаємодіє кінцевий користувач.

Для *MPA* архітектури сторінки є закінченими *HTML* структурами як можна побачити на рисунку 2.2.

[15] HTML-тег “ ”

Page 1

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Page 1</title>
</head>
<body>
    <h1>Page 1</h1>
</body>
</html>
```

Page 2

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Page 2</title>
</head>
<body>
    <h1>Page 2</h1>
</body>
</html>
```

Рисунок 2.2. – приклад архітектури сторінки для MPA

Для *SPA* архітектури та ж частина веб сайту матиме повний *HTML* файл із заповнювачами для фрагментів *HTML*, котрі зберігаються у файлах відображеннях як на рисунку 2.3.

Shell

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Page 2</title>
</head>
<body>
    <div id="container"></div>
</body>
</html>
```

View1

```
<div id="page1">
    <h1>Page 1</h1>
</div>
```

View2

```
<div id="page2">
    <h1>Page 2</h1>
</div>
```

Р

исунок 2.3. – приклад архітектури сторінки для SPA

Можна зазначити, що головна оболонка (*Shell*) відповідає за:

1. Координування модулів відображень
2. Управління станом клієнтської компоненти веб-додатку.
3. Відображення та управління контейнерами відображень.

2.2 DOM та Virtual DOM

DOM або ж *World Wide Web Consortium Document Object Model* – це платформа та мовно-нейтральний інтерфейс, який дозволяє програмам динамічно отримувати доступ та оновлювати вміст, стиль та структуру дукументів.

Тобто *DOM* представляє інтерфейс вашої програми. Кожного разу коли відбувається зміна стану інтерфейсу програми , *DOM* оновлюється, щоб представити цю зміну.

DOM представлений у вигляді такої структури даниї, як дерево. Тому зміни та оновлення відбуваються швидко. Але після зміни потрібно повторно перемалювати оновлений елемент та усі його дочірні елементи, що робить *DOM* повільним.

Тобто, чим більше в програмі компонентів інтерфейсу користувача, тип дорожчими будуть оновлення *DOM*.

Тому виникла концепція *Virtual DOM* , що працює значно ефективніше за справжній *DOM*. Віртуальний *DOM* - це лише віртуальне представлення *DOM*. Кожного разу, коли стан нашої програми змінюється, віртуальний *DOM* оновлюється замість реального.

Коли нові елементи додаються до інтерфейсу користувача, створюється Virtual *DOM*. Кожен елемент є вузлом на цьому дереві. Якщо стан будь-якого з

вузлів змінюється, то створюється новий *Virtual DOM*. Далі виконується порівняння з попереднім *Virtual DOM*.

Після цього *Virtual DOM* обчислює найкращий метод внесення наявних змін у справжній *DOM* як на рисунку 2.4, що забезпечує мінімальну кількість операцій над реальним *DOM*.

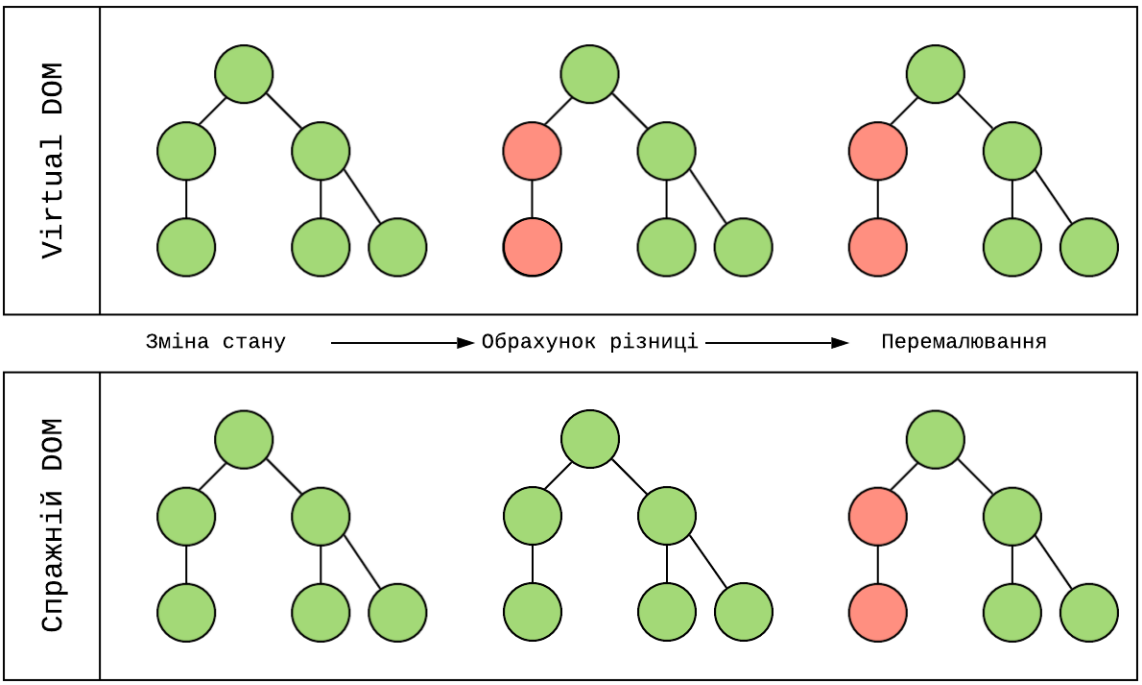


Рисунок 2.4. –алгоритм оновлення DOM

2.3 React для SPA веб-додатку

React – це бібліотека *JavaScript* для розробки інтерфейсів користувача. Компоненти інтерфейсу користувача створюється за допомогою *JavaScript*, а не спеціальної мови шаблонів, тому такий підхід називають створенням сумісних інтерфейсів, що і є основою філософії *React.*

Компоненти *React UI* – є самодостатніми , функціональними блоками для певної специфіку.

Такі компоненти мають як і динамічну логіку, так і візуальне предсталення. *React* використовує компонентну архітектуру – при використанні котрої легше підтримувати, розширяти та заново використовувати компоненти ніж при монолітній архітектурі.

2.3.1 Переваги React

React було створено для вирішення проблеми у розробці та керуванні складними інтерфейсами користувача для веб-додатків.

У дискусії на *React Podcast* [2] згадується, що розробник *React* вирішував проблему наявності кількох джерел даних для оновлення да автоматичного заповнення у *Facebook.*

Дані надходили асинхронно із серверної компоненти. Визначати, куди потрібно вставити нові рядки, щоб використовувати повторно елементи DOM, ставало все складніше.

Уолк вирішив щоразу генерувати елементи *DOM*. Це рішення було елегантним у своїй простоті: інтерфейси як функції. Пізніше з'ясувалося, що генерування елементів у пам'яті відбувається надзвичайно швидко.

Команда *React* придумала алгоритм, який дозволяє уникнути зайвих проблем з DOM. Це зробило *React* дуже швидким (та дешевим з точки зору продуктивності).

Висока ефективність та зручна архітектура на основі компонентів – стало виграшної комбінацією.

Можна зазначити такі переваги використання *React*:

1. Швидкість інтерфейсу користувача - *React* забезпечує надзвичайну продуктивність завдяки своєму *Virtual DOM* та алгоритму розумного узгодження.

2. Простіші програми – компонентна архітектура, що є в *React* в *комбінаціїї* з простим *JavaScript*, зручні водночас потужні й прості абстракції *DOM*.

3. Невелика кількість коду для створення – завдяки розвинутій екосистемі компонентів розробники отримують доступ до великої кількості різноманітних бібліотек.

2.3.2 Використання DOM React-ом

У *React* кожен фрагмент інтерфейсу користувача є компонентом зі станом. *React* використовує схему спостереження і слідкує за змінами стану. Коли стан компоненту змінюється, *React* оновлює дерево *Virtual* DOM.

*Я*к тільки було виконано оновлення *React* порівнює поточну версію *Virtual* DOM з попередньою версією. Такий процес називається “*diffing*”.

Коли *React* дізнається, які *Virtual* DOM-об’єкти змінилися – він оновлює їх у справжньому DOM. Це робить ефективність набагато кращою в порівнянні з маніпулюванням реальним DOM на пряму.

Тобто розробникам не потрібно турбуватися про те, як маніпулювати атрибутами, щоб змінити відображення, бо обробка подій, або оновлення DOM відбувається за кадром. *Virtual* DOM існує тільки у пам’яті *JavaScript.*

На рисунку 2.5 показано на високому рівні, як працює *Virtual* DOM, коли відбувається зміна даних.

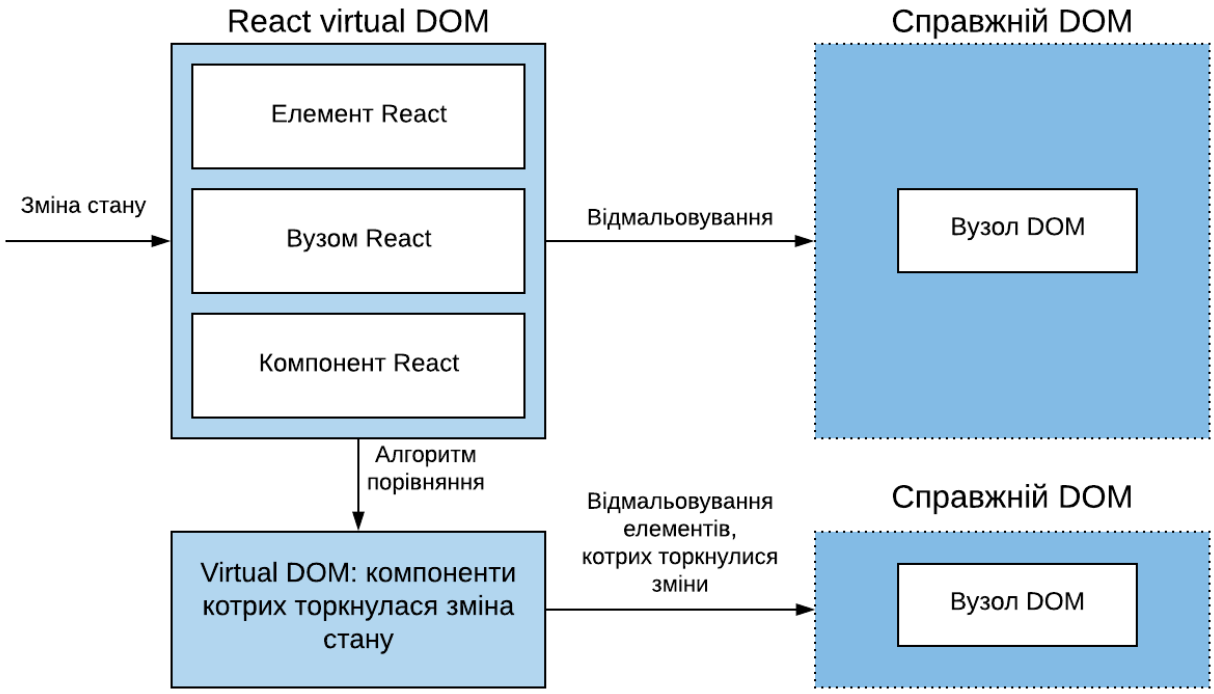


Рисунок 2.5. – приклад оновлення DOM з використанням React

React виконує оновлення тільки потрібних елементі, тобто *Virtual DOM* та справжній DOM – однакові.

Наприклад якщо у нас є елемент тег-*span* і розробник доповнює його через стан компонента, то виконається оновлення тільки тексту елемента, а не самого елементу. Це призводить до підвищення продуктивності порівняно з наданням цілих наборів елементів або, тим більше, цілих сторінок.

2.3.3 Побудова React застосунку з MVC підходом

Концепція *Model-View-Controller* (*MVC*) дозволяє розділити данні, відображення та обробку дій користувача на окремі три компоненти:

1. Модель, що відповідає за:
    a. зміну власного стану у відповідь на запити;
    b. данні та методи роботи з ними;
    c. не знає як візуалізувати власні дані;
2. Відображення – відповідає за візуалізацію інформації.
3. Контролер – контролює ввід даних користувачем і за допомогою моделі та відображення виконує потрібну реакцію.

Така структура забезпечує гнучкість за рахунок чіткого розділення. Стає простіше обслуговувати та використовувати код, створюється підтримка асинхронних реалізацій.

React не вимагає використання *MVC* архітектури для власної реалізації, але у даній курсовій роботі мною було використано саме такий концепт.

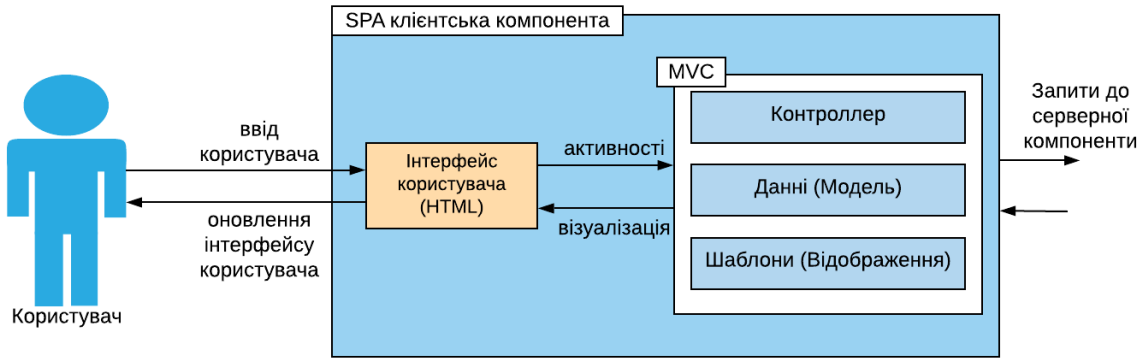


Рисунок 2.6. – архітектура MVC для SPA

На рисунку 2.6 контролер диктує, які потрібно використовувати шаблони (відображення) та обирає дані котрі необхідно отримати.

Контролер робить запит на отримання даних, а потім заповнює шаблони цими даними для візуалізації інтерфейсу користувача у формі *HTML*. Усі дії користувача передаються назад у код.

Сам *React* розміщується в блоці “Шаблони” і є рівнем відображення. Тому його можна використовувати для візуалізації *HTML*.

3 ОГЛЯД СТВОРЕНОЇ КЛІЄНТСЬКОЇ КОМПОНЕНТИ ВЕБ-ДОДАТКУ ПОБУДОВАНОГО НА SPA АРХІТЕКТУРІ

Клієнтська компонента *SPA* веб-додатку була побудована з використанням таких технологій як *React, Redux, React-Redux* для застосування концепції *MVC*. Та використовуючи *JavaScript*, CSS, HTML та JSX.

На рисунку 3.1 визначено поділ клієнтської компоненти для *MVC* концепту.

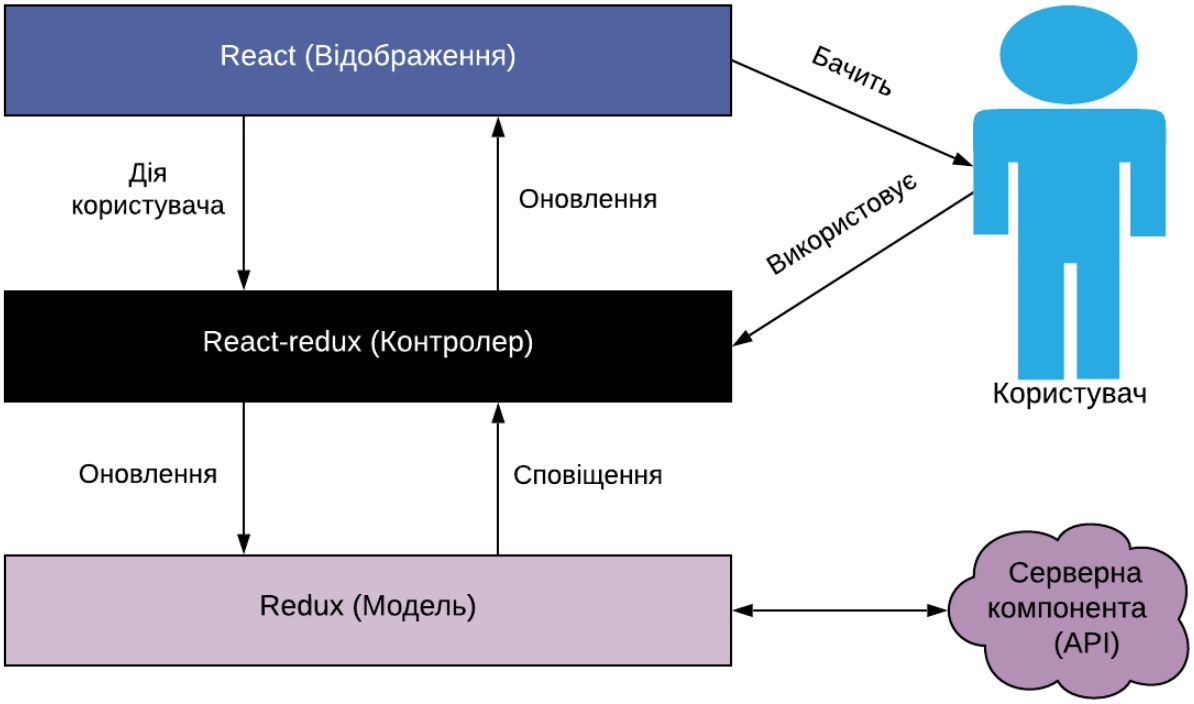


Рисунок 3.1. – відтворення NVC з використанням React

3.1 React в ролі відображення

За відображення у розробленій програмі відповідає *React*. Оскільки за рахунок декларативності він дозволяє створити різні відображення для кожного стану, що будуть ефективно оновлюватися та відтворюватися. Також є

можливість будувати компоненти, що мають власний стан та можуть його структурувати разом у більш складний інтерфейс користувача.

Для того щоб не заплутатися в коді *SPA* проєкту, було вирішено створити структуру папок, відображену на додатку А.

Дана структура підпорядковується такій логіці – в папці "*src*" знаходяться усі файли веб-додатку, компоненти, що можуть бути використані для побудови інших складніших компонентів розміщені у папці "*components*" для ефективного імпорту як зазначено на додатку Б.

Усі константні значення, файли налаштувань, адреси для запитів на серверну компоненту та локалізація розташована у папці "*constants*".

Найскладніші компоненти, що наприклад є сторінками веб-додатку, згруповані залежно від стану, коли вони візуалізуються. Тобто вони є сценами, котрі складаються з головної компоненти та допоміжних.

Кожна сцена може мати свій власний файл стилів та додаткову папку з допоміжними компонентами для побудови – мікро-сценами. За потреби мікро-сцени можуть бути згруповані у папки, якщо вони розміщуються навколо певної функціональної потреби, як це відображено на додатку В, де у сцені що відповідає за завантаження кадрів є допоміжні компоненти відповідальні за форму завантаження та таблицю результатів завантаження.

Точкою входу для клієнтської компоненти є файл index.js. тут вказується кореневий *DOM*-елемент, в котрий я кладу головний компонент *App*, що і є *React*-додатком, завчасно обгорнутим в провайдер.

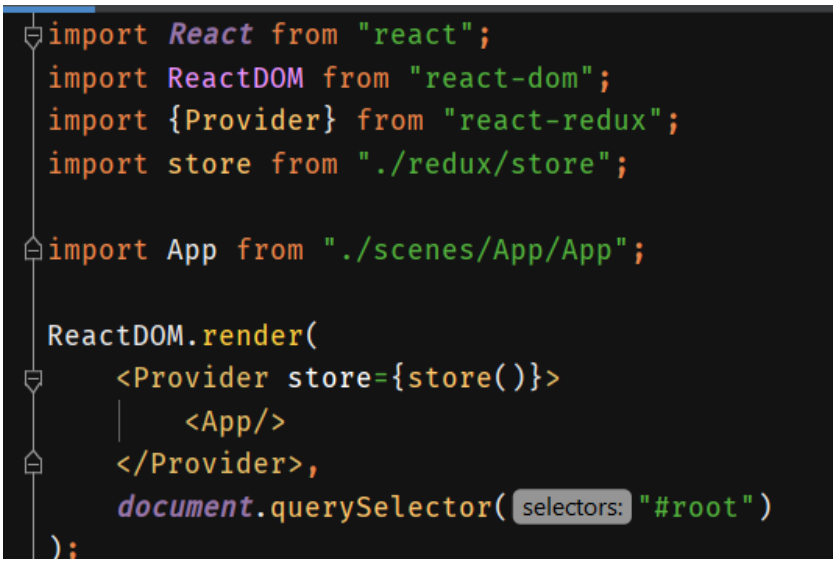


Рисунок 3.2. – додання App в DOM - елемент

Для зручності використовується *JSX*-синтаксис (*JavaScript XML*) – це синтаксичний цукор для визначення компонентів та їх позиціонування всередині розмітки. *JSX* дозволяє використання будь-якого *JavaScript* коду всередині нього, що поліпшує створення компонентів, що відображають комплексні дані.

Але оскільки веб-браузер не розуміє *JavaScript* файли, що містять *JSX* код, то спочатку він має бути трансформований у простий та зрозумілий *JavaScript*, що можна виконати наприклад за допомогою *Babel* [16].

Наведу для прикладу компоненту “*LoginForm*” котра відповідальна за відображення форми авторизації.

[16] транскопілятор JavaScript з відкритим кодом, який в основному використовується для перетворення коду ECMAScript 2015+ (ES6 +) у сумісний JavaScript

```
render() {
    return (
        <Fragment>
            <h3>Форма входу</h3>
            <form onSubmit={this.props.handleSubmit(this.onSubmitPressed)}>
                <Field name="username" component={this.renderInputField} type="text" label="Логін"/>
                <Field name="password" component={this.renderInputField} type="password" label="Пароль"/>
                <button className="btn btn-secondary" type="submit">
                    Надіслати
                </button>
            </form>
        </Fragment>
    );
}
```

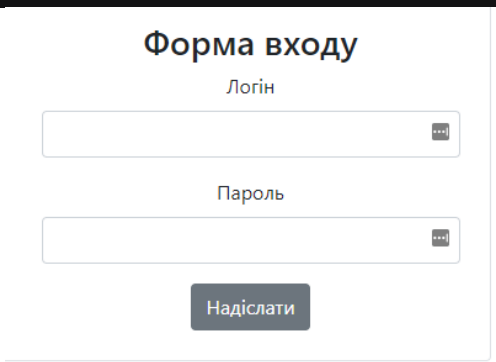


Рисунок 3.3. – компонент, що відображає форму

Сама по собі ця компонента нічого не здатна зробити, оскільки ніякої логіки обробки даних в ній немає. При створенні вона намагається використовувати функції з *Props* [17], про котрі нічого не знає.

3.2 Redux в ролі Моделі

Перед тим як зануритися в *Redux*, важливо зрозуміти архітектурну ідею на котрій ця технологія працює. У попередніх, досліджених мною проєктах була можливість керувати станом компонентів починаючи з верхнього рівня. Тобто компонент *React* верхнього рівня керував основним станом, передаючи дані в низ у дочірні компоненти.

[17] довільні дані, котрі приймають компоненти React

Для зміни стану, дочірні компоненти надсилали події до батьківських через допоміжні функції, котрі приймали як *Props*. Будь які мутації стану відбувалися вгорі, а потім знову падали вниз.

Для складних веб-додатків, коли одна дія користувача впливала на велику кількість дискретних частин стану то викликала проблеми, через складність керування усіма оновленнями, що мали передаватися по складній ієрархії компонентів. Коли *Facebook* зіткнулася з цією проблемою – це їх мотивувало винайти *Flux* архітектуру.

3.2.1 Flux

Flux складається з 4 ключових елементів:

- *Action* – об’єкти із властивісю та данними;
- *Stores* – містять логіку та стан застосунку;
- *The Dispatcher* – процес, що реєстрував *action* та зворотній виклик (*callback*) ;
- *Controller Views* – *React* компоненти, що брали стан зі *Stores* й передавали дочірнім компонентам;

Увесь функціонал, як правило, знаходиться в *Store*. У *Store* виконується уся робота та повідомляється *Dispatcher* які типи подій він має очікувати. Коли відбувається подія *Dispatcher* надсилає “корисне навантаження ” в *Store*, котрий зареєстрований саме для цієї події.

Тепер в *Store* виконується оновлення *View*, що викликає *action*. *View* поширює *action* через центральний *dispatcher*, що в свою чергу надсилається у різні *Store*, що відображають бізнес логіку програми та інші дані. Store оновлює всі *View*.

Тобто завжди дані проходять через *Dispatcher*, котрий керує всіма даними.

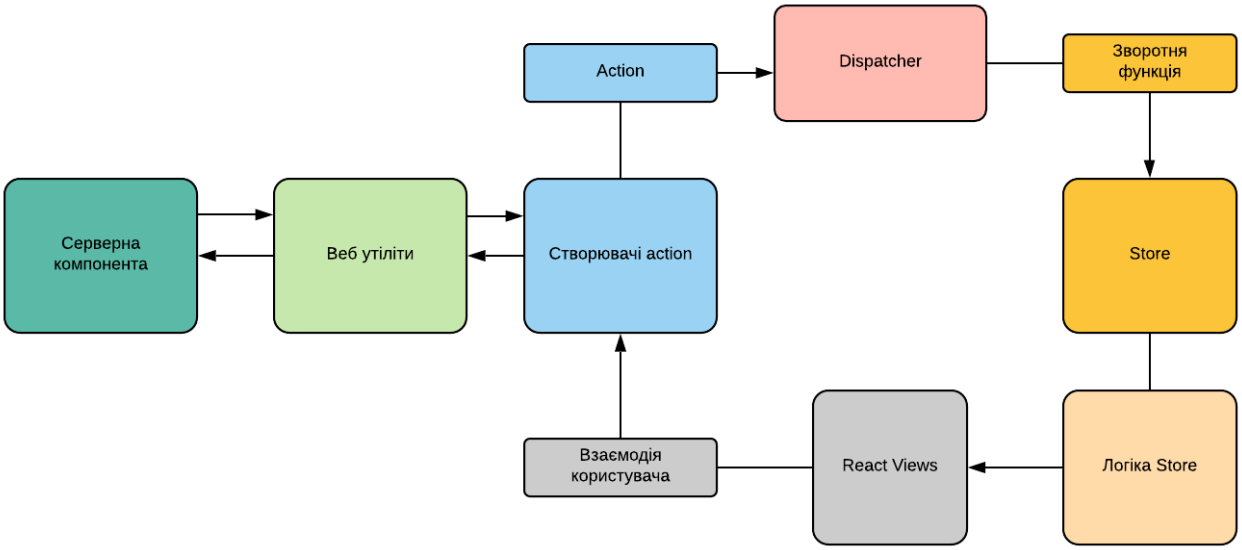


Рисунок 3.4. – життєвий цикл Flux

Використовуючи таку архітектуру можна позбавити головну компоненту відповідальності за увесь стан, спростити *React* компоненти, що стануть легшими для розуміння. А також *Flux* дозволяє винести усю обчислювальну логіку для отримання даних поза компоненту.

3.2.2 Redux

Redux побудований на основі *Flux* і може бути описаний в трьох основних принципах:

- одне єдине джерело істини - *стан* усього застосунку зберігається в одному *store*.
- стан тільки для *считування* – єдиний спосіб змінити стан – це передати *action*.
- змінити вносяться тільки чистими функціями – перетворення виконується за допомогою *action* з *reducer*, що дозволяє переходити по станах.

Redux відмінний від *Flux* тим, що не має поняття *dispatcher*, тому що покладається на чисті функції замість активаторів подій.

А також *Redux* передбачає, що дані не змінюються, а завжди повертається новий об’єкт. Приблизну архітектуру *Redux* можна побачити на додатку Г.

3.2.1.1 Структурна організація Redux

Для побудови масштабного веб-додатку, щоб можна було у майбутньому легко його розширювати виникає потреба у правильній структурній організації *Redux*.

Я вирішив скористатися підходом що фокусуються на окремих особливостях.

Тобто, що каталоги верхнього рівня називаються за основними функціями програми, як це видно да рисунку 3.5.

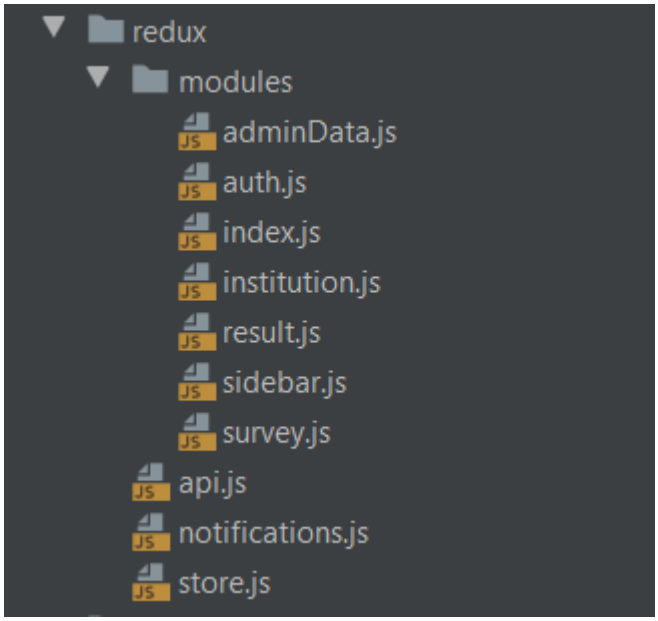


Рисунок 3.5. – структура папки Redux

За допомогою цього, масштабування виконується набагато краще. Кожна функція програми має відображати окремі дії та *Reducers*. Такий підхід назвали re-*duck*.

Тобто вирішенням було розбити кожну функцію на папку *duck*, у файловій структурі це відображаться модулями.

Кожен такий модуль має містити всю логіку для обробки лише одного концепту веб-застосунку. Він містить декілька основних частин:

1. Типи - містить назви дій, які надсилаються у програмі. Хорошою практикою є розширення імен, що складаються з назв функцій, що допомагає при налагодженні складних програм. Приклад можна побачити на додатку Д.
2. *Action* – блок, що має усі функції для створення дій, відображений на додатках Е та Ж.
3. *Reducers* – якщо особливість (функція) має декілька можливих типів, тоді обов'язково використовувати кілька *reducers* для обробки різних частин стану. Це дає гарну гнучкість при роботі зі складними формами станів, приклад відображено на додатку К.

Сам глобальний *Store* для застосунку створюється у корені папки "*redux*" у файлі "store.js" рисунок 3.6.

```
const composeEnhacers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

export default function store() {
  return createStore(
    rootReducer,
    composeEnhacers(applyMiddleware(thunkMiddleware))
  );
}
```

Рисунок 3.6. – створення Store

Для використання асинхронних *action* використовується спеціальна бібліотека "*Redux-Thunk*".

Вона дозволяє нам викликати створювача подій, повертаючи при цьому функцію, а не об'єкт.

Сама функція приймає метод *dispatch* у якості аргументу, щоб після закінчення асинхронної операції, використовувати даний *dispatch* для простого синхронного *action* в середині тіла функції.

3.3. React-redux в ролі Контролера

Усі компоненти *React* отримують свій стан та зворотні функції для його зміни тільки через *Props*. Ні одна компонента не знає про існування *Redux*, чи *action*, а *reducer* чи створювач подій не знає про *React*-компоненти.

Данні та логіка для їх обробки повністю розмежовані від їх відображення. Використання *react-redux* відображено на додатку Л.

3.4. Переваги використання

Розглянуті архітектурні підходи дозволяють контролювати кожну частину клієнтської компоненти, покращують надійність за рахунок жорстких правил, збільшують ефективність кожної компоненти, дозволяють використовувати створені компоненти повторно за рахунок того, що вони спеціалізуються на відображенні конкретної інформації та конкретних взаємодій з користувачем.

Також застосовані архітектури спрощують тестування, адже логіка та відображення не пов’язані між собою.

Перевагу використання *MVC* архітектури для *React* компонентів реалізовану за допомогою *Redux* продемонстровано на рисунку 3.7.

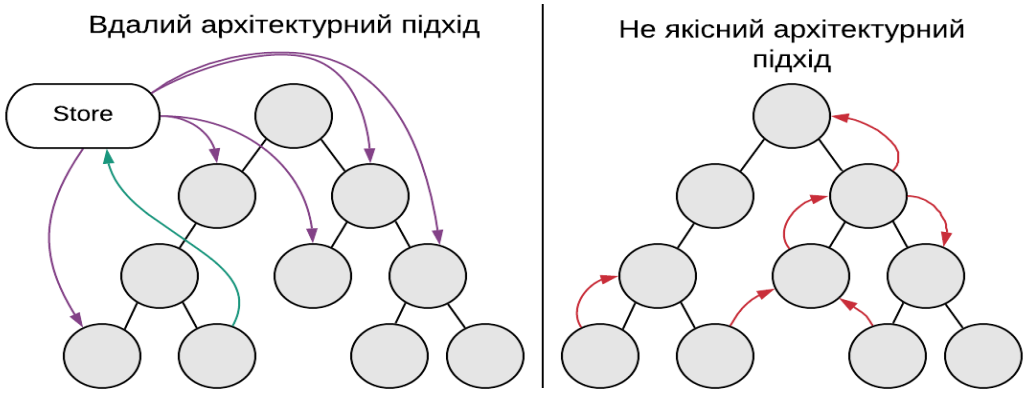


Рисунок 3.7. – порівняння Redux підходу та підходу передачі стану через Props

Можна помітити, що при вдалому архітектурному підході увесь контроль стану передається *Store*, що зменшує можливість виникнення помилок у розробці веб-додатку, фокусує завдання *Redux*-компонент на виконанні відображення та полегшує ведення процесу оновлень.

ВИСНОВКИ

Обрання вірного архітектурного підходу для побудови сучасного веб-застосунку є чи не найважливішим завданням будь-якого розробника. Адже від цього залежить подальший процес розробки усього веб-додатку, можливості його покращення, розширення та підтримки.

Ефективна та швидка робота так само залежить від правильних архітектурних підходів.

Можна визначити, що гарно збудована та продумана архітектура має вирішувати можливі проблеми з масштабуванням, швидкістю виконання, часом розробки та ціною для створення веб-застосунку.

Із розвитком технологій продовжує розвиватися й сама архітектура веб-додатків.

На зміну застарілому *MPA* прийшов *SPA* підхід, що мотивував розвиток цілої сфери фреймоврків[18]. Він зробив досвід кінцевого користувача з додатком більш природнім та ефективнішим.

Також використання такої архітектури спростило роботу розробникам, котрі замість того щоб витрачати ресурси та час на другорядні завдання, можуть сфокусуватися на виконанні чітких потреб кінцевого користувача.

Створення складних веб-застосунків стало ефективнішим та швидшим.

Було розглянуто й інші архітектурні підходи, що активно розвиваються – Мікро сервіси та Serverless, що більше орієнтовані на особливості серверної компоненти.

Використання таких архітектур дало можливість компаніям впроваджувати власні продукти набагато швидше за рахунок аутсорсингових послуг, а також спрощене керування інфраструктурою потрібною для веб-додатку зменшило кількість проблем, що можуть виникати при недосвідченому керуванні та моніторингу.

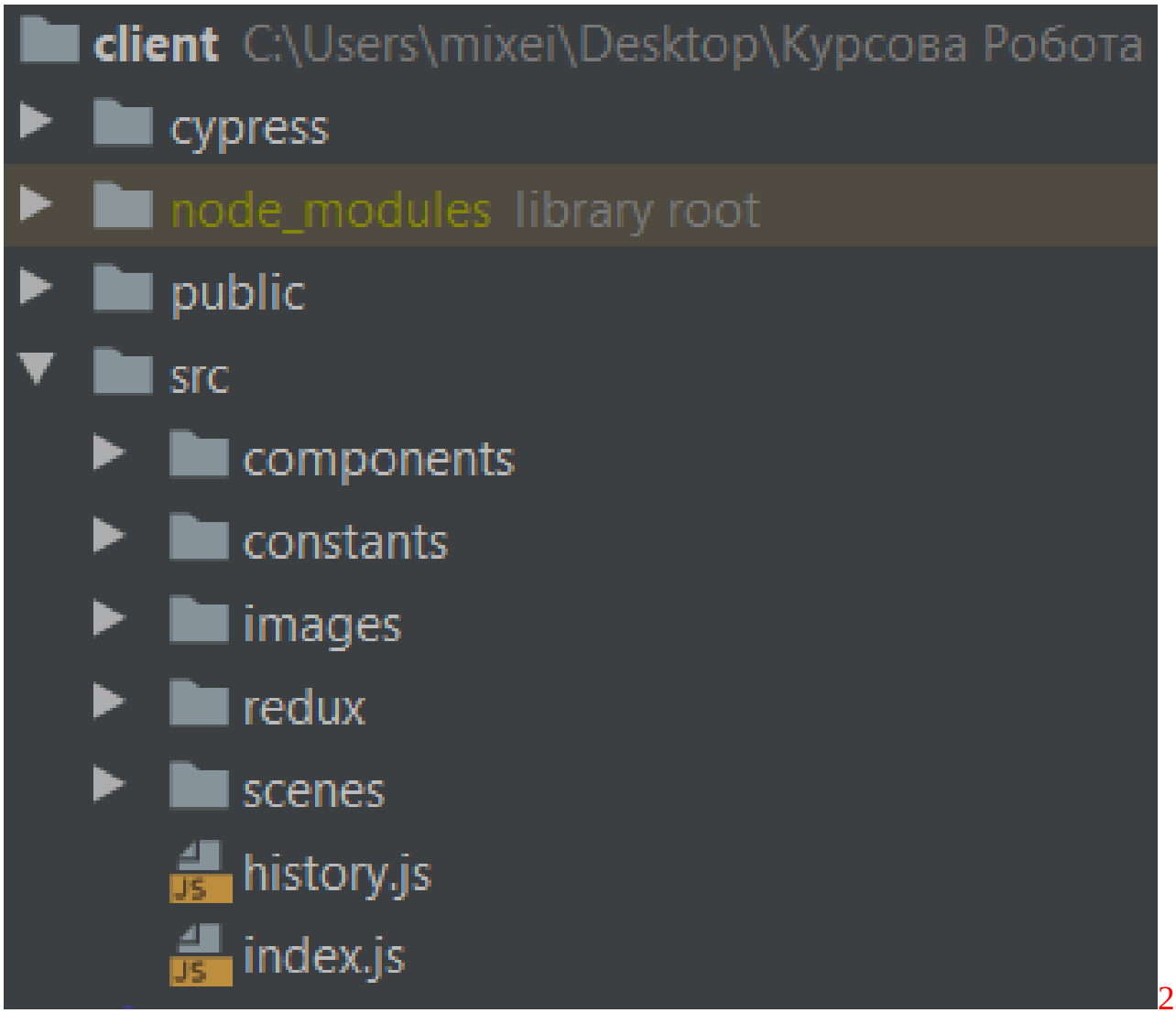
[18] інфраструктура програмних рішень, що полегшує розробку складних систем.

На основі *SPA* архітектури було розроблено клієнтську компоненту веб-додатку для “Сервісу опитування студентів про якість викладання дисциплін”.

Також було застосовано практику *MVC* реалізовану через *React* та *Redux*, для того щоб додаток був гнучким, масштабованим та ефективним. Розроблений додаток відповідає концепту *SPA* і за потреби може легко бути перебудований для використання з *Serverless* підходом.

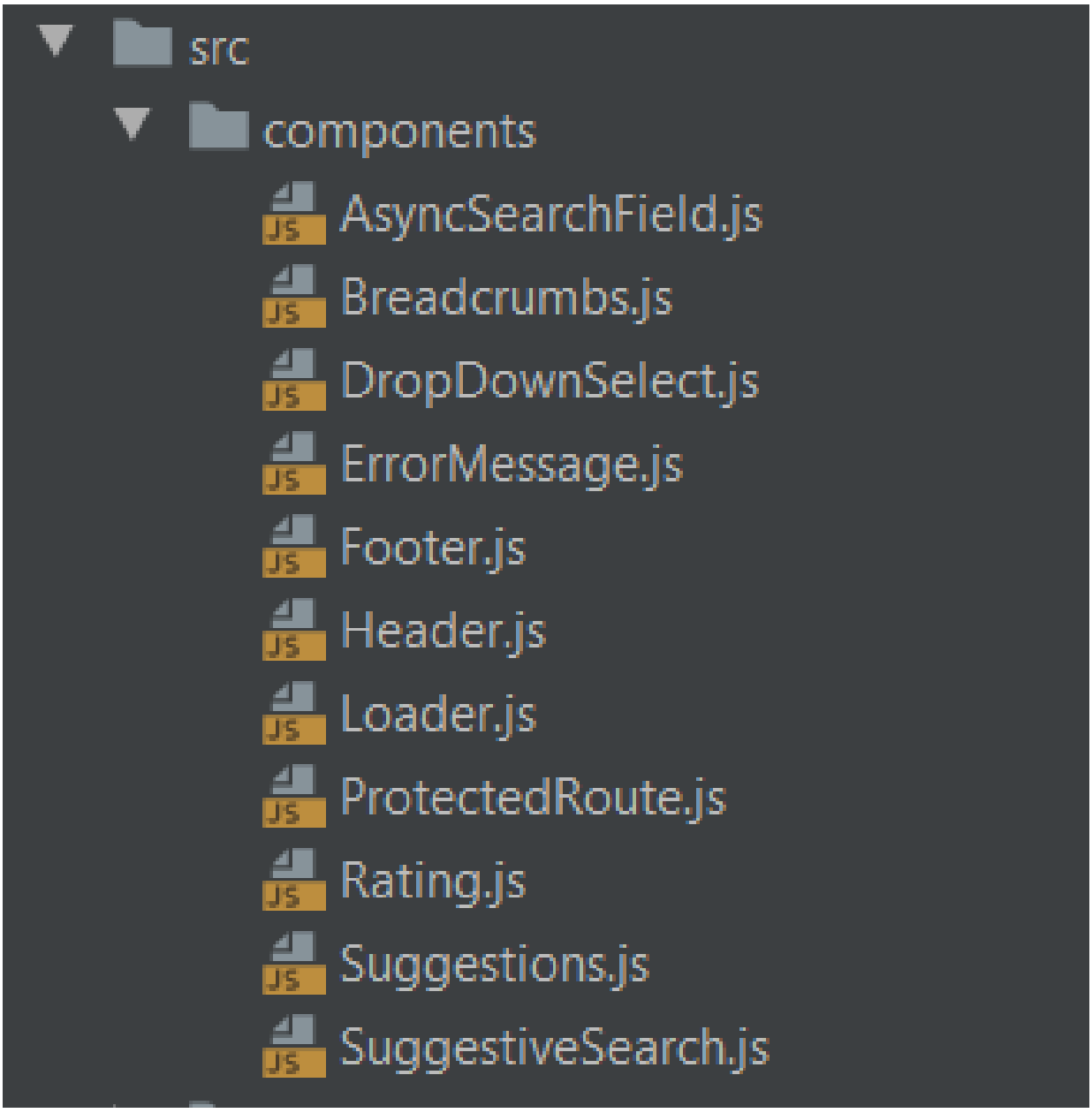
Було продемонстровано ефективність використання *React* для побудови сучасних веб-застосунків, а також показаний процес, що аргументує доречність використання даної технології.

Додаток А

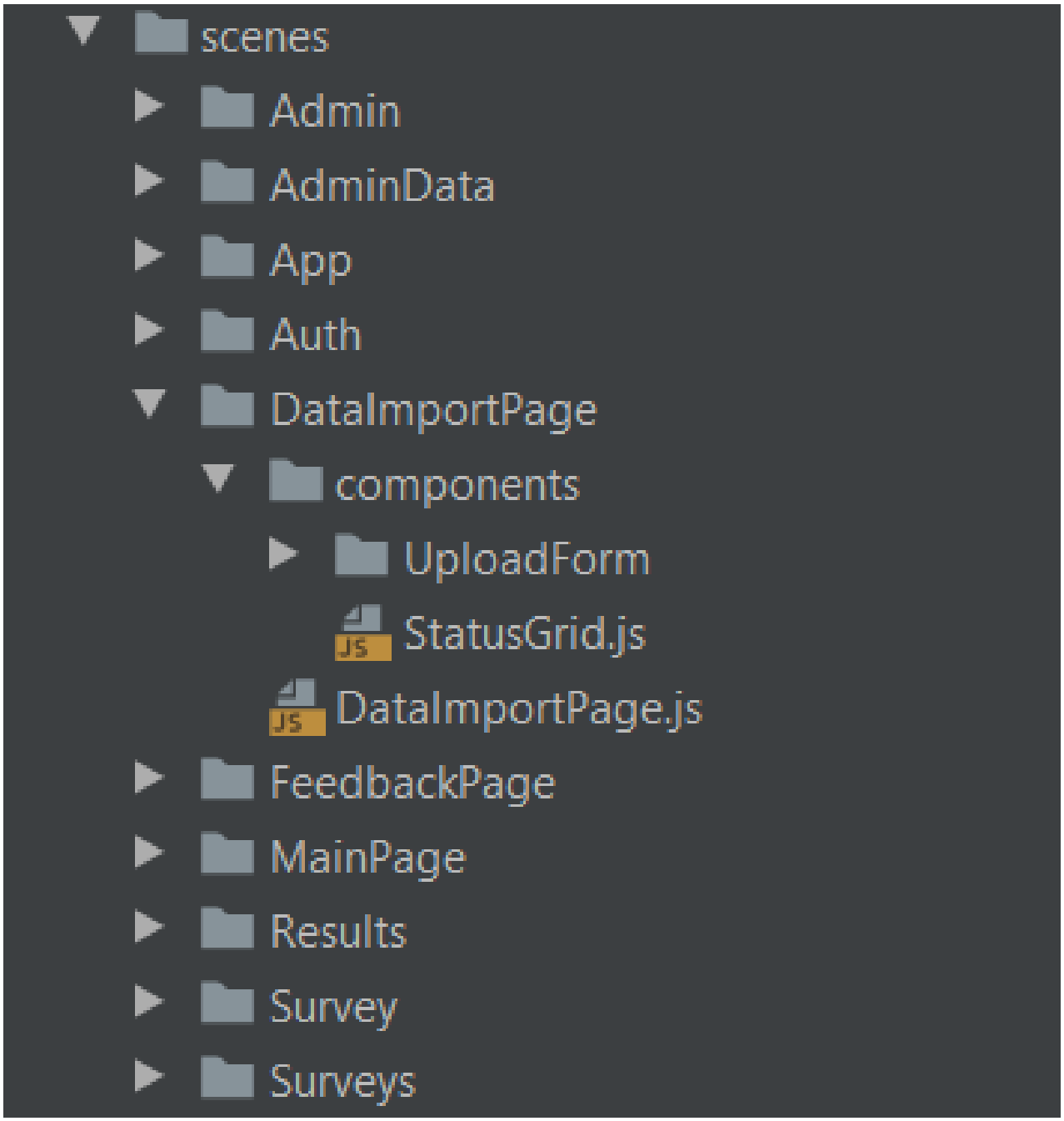

client C:\Users\mixei\Desktop\Курсова Робота
cypress
node_modules library root
public
src
components
constants
images
redux
scenes
history.js
index.js

2

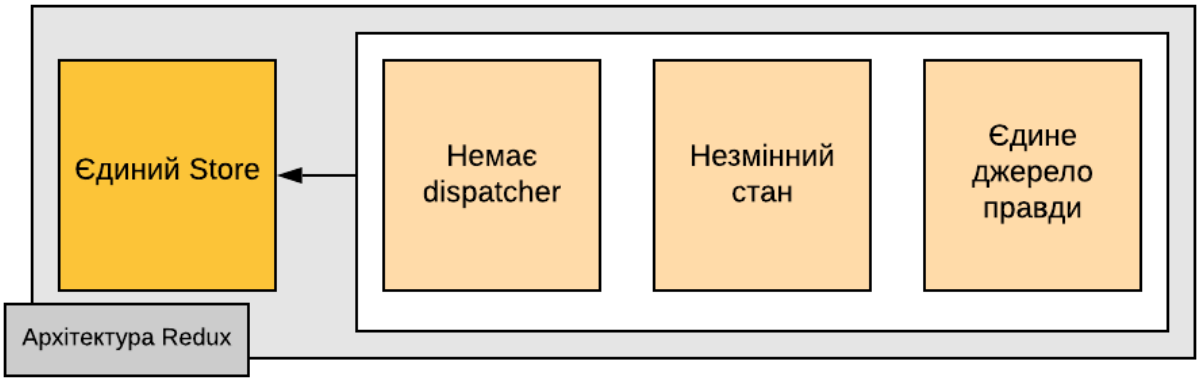
Додаток Б


src
components
AsyncSearchField.js
Breadcrumbs.js
DropDownSelect.js
ErrorMessage.js
Footer.js
Header.js
Loader.js
ProtectedRoute.js
Rating.js
Suggestions.js
SuggestiveSearch.js

Додаток В


scenes
Admin
AdminData
App
Auth
DataImportPage
components
UploadForm
StatusGrid.js
DataImportPage.js
FeedbackPage
MainPage
Results
Survey
Surveys

Додаток Г


Єдиний Store
Немає dispatcher
Незмінний стан
Єдине джерело правди
Архітектура Redux

Додаток Д

```
const ACTION_TYPES = {
  OAUTH: "auth/OAUTH",
  SIGN_IN: "auth/SIGN_IN",
  SIGN_OUT: "auth/SIGN_OUT",
  SET_USER: "auth/SET_USER",
  SIGN_IN_ERROR: "auth/SIGN_IN_ERROR",
  DELETE_ERROR: "auth/DELETE_ERROR",
  GET_USER_INFO: "auth/GET_USER_INFO",
  GET_USER_ACCESS: "auth/GET_USER_ACCESS"
};
```

Додаток Е

```
const doGetUserInfo = () => async dispatch => {
  await authApi
    .get("/auth/me")
    .then(resp => {
      dispatch({ type: ACTION_TYPES.GET_USER_INFO, payload: resp.data });
      dispatch(doGetUserAccess());
    })
    .catch(e => {
      if (window.location.pathname !== '/')
        addNotification( message: "Авторизуйтесь у системі для подальшої роботи.");
        dispatch({ type: ACTION_TYPES.SIGN_IN_ERROR, payload: e.response });
      history.push("/auth");
    });
};

const doGetUserAccess = () => async dispatch =>{
  await authApi.get("/auth/access").then(resp=>{
    dispatch({type: ACTION_TYPES.GET_USER_ACCESS,payload: resp.data});
  });
};
```

Додаток Ж

```
const actionCreators = {
  doAuth,
  doSignIn,
  doSignOut,
  doDeleteError,
  doGetUserInfo
};
```

Додаток К

```
import results from "./result"
import admin_data from "./adminData"
import sidebar from "./sidebar"
import {reducer as formReducer} from "redux-form";

const appReducer = combineReducers( reducers: {
  auth: auth,
  institutions: institutions,
  surveys: surveys,
  results: results,
  admin_data: admin_data,
  sidebar: sidebar,
  form: formReducer
});

const rootReducer =  (state, action) => {
  if (action.type === ACTION_TYPES.SIGN_OUT) {
      state = undefined;
  }
  return appReducer(state, action)
};
```

Додаток Л

```
function mapDispatchToProps(dispatch) {
    return {
        signIn: bindActionCreators(actionCreators.doSignIn, dispatch),
        closeModal : bindActionCreators(actionCreators.doDeleteError,dispatch)
    };
}

const LoginReduxForm = connect(
    null,
    mapDispatchToProps
)(LoginForm);

export default reduxForm({
    form: 'login',
    validate: validate
})(LoginReduxForm);
```

Список літератури

1. A Quarter-Century of Unix (1994).
2. React Podcast, “8. React, GraphQL, Immutable & Bow-Ties with Special Guest Lee Byron,” December 31, 2015, http://mng.bz/W1X6.
3. “Single Page Web Applications: JavaScript end-to-end” by Michael S. Mikowski and Josh C. Powell September 2013
4. “Learning React” by Alex Banks, Eve Porcello May 2017
5. “Learn React Hooks” by Daniel Bugl October 2019