МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет комп'ютерних наук, фізики та математики
Кафедра комп'ютерних наук та програмної інженерії

ОСОБЛИВОСТІ ПРОЕКТУВАННЯ SPA З ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ.

Кваліфікаційна робота (проєкт)
на здобуття ступеня вищої освіти "бакалавр"

Виконав здобувач 4 курсу 441 групи
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійної програми «Інженерія програмного забезпечення»
Авраменко Петро Сергійович
Керівник доктор пед.н., професор кафедри комп'ютерних наук та програмної інженерії Круглик В.С., кандидат фізико-математичних наук, доцент Ермолаев В.А.
Рецензент: керівник центру компанії DataArt, Щедролосьєв Д.Є.

Херсон – 2022

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА ТЕРМІНІВ

1. **SPA** — це тип веб-застосунків, у яких завантаження необхідного коду відбувається на одну сторінку.
2. **API** — програмний інтерфейс додатку (application programming interface).
3. **AMD** — Асинхронне визначення модуля (asynchronous module definition, AMD).
4. **MPA** — багатосторінковий веб-додаток.
5. **HTML** — мова розмітки документів (Hypertext Markup Language)
6. **JSON** — текстовий формат обміну даними на основі синтаксису ECMAScript (Javascript object notation).
7. **JWT** — коротке повідомлення з криптографічними перетвореннями, що використовується зокрема для аутентифікації користувачів (JSON Web Token).
8. **ORM** — відображення моделей даних на структури бази даних (ObjectRelational Mapping)
9. **REST** — передача репрезентативного стану системи, підхід до організації API (Representational state transfe).
10. **SQL** — мова запитів до реляційних баз даних (Structured Query Language).
11. **Авторизація** — підтвердження користувачем права на виконання тих чи інших дій.
12. **Аутентифакація** — підтвердження користувачем своєї особи.

ВСТУП

Актуальність теми дослідження обумовлена великим попитом та популярністю використання SPA з мікросервісної архітектури у сучасній розробці.

В умовах сучасності існують багато великих веб-додатків, які були б дуже ресурсомісткими, як би не мікросервіси. Вони допомогають розбити важкі задачі на окремі сервіси, що полегшує роботу сервера та кінцеву взаємодію з користувачем.

Метою кваліфікаційної роботи є створення SPA за допомогою використовування мікросервісної архітектури замість класичних підходів до розробки та порівняти їх зручності, складності та кінцевого результату.

Згідно з поставленою метою в роботі вирішуються наступні **завдання**:

1. провести теоретичний та порівняльний аналіз архітектур розробки серверної частини додатків.

2. Дослідити можливості SPA додатків та їх взаємодії з мікросервісною архітектурою.

3. провести порівняльний аналіз середовищ для тестування SPA додатків та обрати оптимальний;

4. на основі проаналізованих наукових джерел проектувати та розробити повноцінний SPA, який взаємодіє з мікросервісною архітектурою.

Об’єктом дослідження є особливості проектування SPA з використанням мікросервісної архітектури.

Предметом дослідження є проектування та розробка SPA додатку з мікросервісною серверною частиною.

У процесі виконання кваліфікаційної роботи були використані наступні **методи дослідження**:

- теоретичні: аналіз науково-теоретичних джерел, що забезпечує ознайомлення з сучасним станом об’єкта дослідження з метою встановлення

вихідної концепції предмета дослідження й визначення його понятєвого апарату;

- емпіричні: порівняльний аналіз, з метою визначення оптимального програмного забезпечення для вирішення проблеми дослідження.

Структура роботи. Робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків загальним обсягом 42 сторінки, 1 з яких – додатки.

РОЗДІЛ 1
ТЕОРЕТИЧНИЙ ОГЛЯД ПІДСТАВ ЗАСТОСУВАННЯ SINGLE-PAGE APPLICATION

1.1 Про single-page application

SPA – це веб-додаток, який розміщується на одній веб-сторінці, яка для забезпечення роботи завантажує весь необхідний код разом з завантаженням цієї сторінки. Додатки такого типу з’явились нещодавно, з появою нового стандарту HTML5.

Розглянемо приклад коли додаток достатньо великий і містить багатий функціонал. Тоді кількість файлів та скриптів може складати декілька сот або і тисяч, що може сповільнити перше завантаження додатку. Для вирішення цієї проблеми при розробці SPA використовують зовнішнє API під назвою AMD (рис. 1.1).

AMD – це асинхронне визначення модуля, де модулі та їх залежності можуть бути завантажені асинхронно. Асинхронна загрузка модулів дозволяє покращити швидкість завантаження веб-сторінки в цілому, так як модулі завантажуються одночасно з іншим контентом сайту.

Тобто ця технологія реалізує можливість завантаження контенту або скриптів за запитом. Наприклад, якщо для головної сторінки SPA потрібно 4 скрипта, то вони будуть завантажені перед стартом додатку, а якщо користувач перейшов на іншу сторінку, яка потребує додаткового контенту або скриптів, то за принципом AMD буде завантажений відповідний модуль перед переходом на нову сторінку.

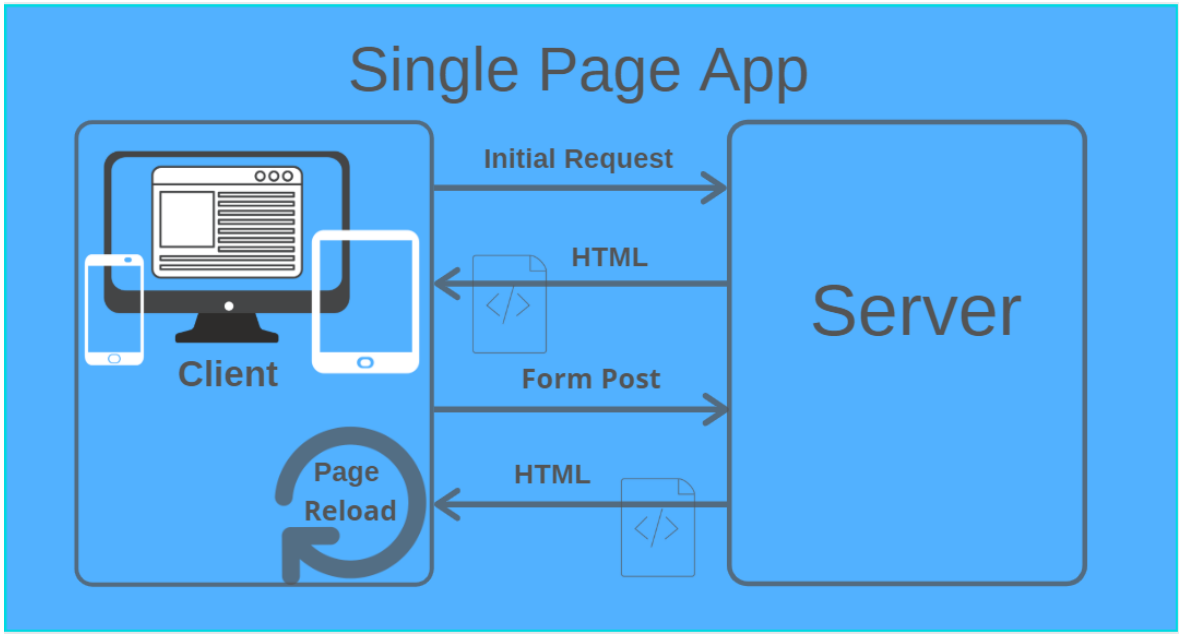


Рисунок 1.1 робота SPA.[1]

Принципи будь якого фреймворку, який реалізує парадигму розробки SPA повинні притримуватись наступних понять та визначень:

- SPA підтримує клієнтську навігацію. Всі переходи користувача по веб-сторінкам фіксуються в історії навігації, при чому навігаціє є глибокою. Це коли користувач може скопіювати і відкрити посилання на внутрішню сторінку в іншому браузері чи вікні, та потрапити на відповідну сторінку.
- SPA розміщується на одній сторінці, тобто всі необхідні для роботі додатку файли повинні бути в одному місці проекту – на єдиній веб-сторінці.
- SPA зберігає постійний стан роботи клієнта в кеші браузера або Web Storage.
- SPA завантажує всі скрипти, що потрібні для старту додатку при ініціалізації веб-сторінки.
- SPA поступово завантажує модулі за вимогою.

SPA взаємодіє с користувачем шляхом динамічного перезапису поточної сторінки, а не завантаження цілих нових сторінок з сервера.

Цей підхід усуває переривання взаємодії з користувачем між послідовними сторінками, роблячи додаток більш схожим на настільний додаток.

На більшості веб-сайтів багато повторюваного контенту. Деякі з них залишаються незмінними незалежно від того, куди переходить користувач (заголовки, нижні колонтитули, логотипи, панель навігації і т. д.), деякі з них постійні тільки в певному розділі (панелі фільтрів, банери), і існує безліч макетів, що повторюються, і шаблонів.

Односторінкові додатки використовують це повторення.

Поряд із більш швидкою продуктивністю, описаною вище, SPA також дозволяють розробникам набагато швидше створювати зовнішній інтерфейс. Це пов'язано з непов'язаною архітектурою SPA або розподілом серверних служб та зовнішнього дисплея.

Багато критично важливих для бізнесу функцій не сильно змінюються на серверній частині. Хоча те, як ваші клієнти входять в систему, реєструються, купують і відстежують замовлення, може час від часу змінювати «зовнішній вигляд» або уявлення, логіка та оркестрування даних, що стоять за цим, досить постійні – і ви не хочете ризикувати зіпсувати їх.

Так само необроблений контент і дані можуть залишитися незмінними, але те, як ви хочете їх відобразити, відрізняється. Відокремлюючи цю внутрішню логіку та дані від того, як вони представлені, ви перетворюєте їх на "сервіс", і розробники можуть створювати безліч різних способів зовнішнього інтерфейсу для відображення та використання цього сервісу.

Завдяки роздільному настроюванню розробники можуть створювати, розгортати та експериментувати із зовнішнім інтерфейсом повністю незалежно від базової серверної технології. Вони проектують те, як вони хочуть, щоб інтерфейс користувача виглядав і відчувався, а потім завантажують контент, дані і функціональні можливості через ці служби.

Це робиться за допомогою API, які є стандартним набором правил між додатками про те, як вони будуть структурувати, обмінюватися і збирати дані (рис. 1.2).

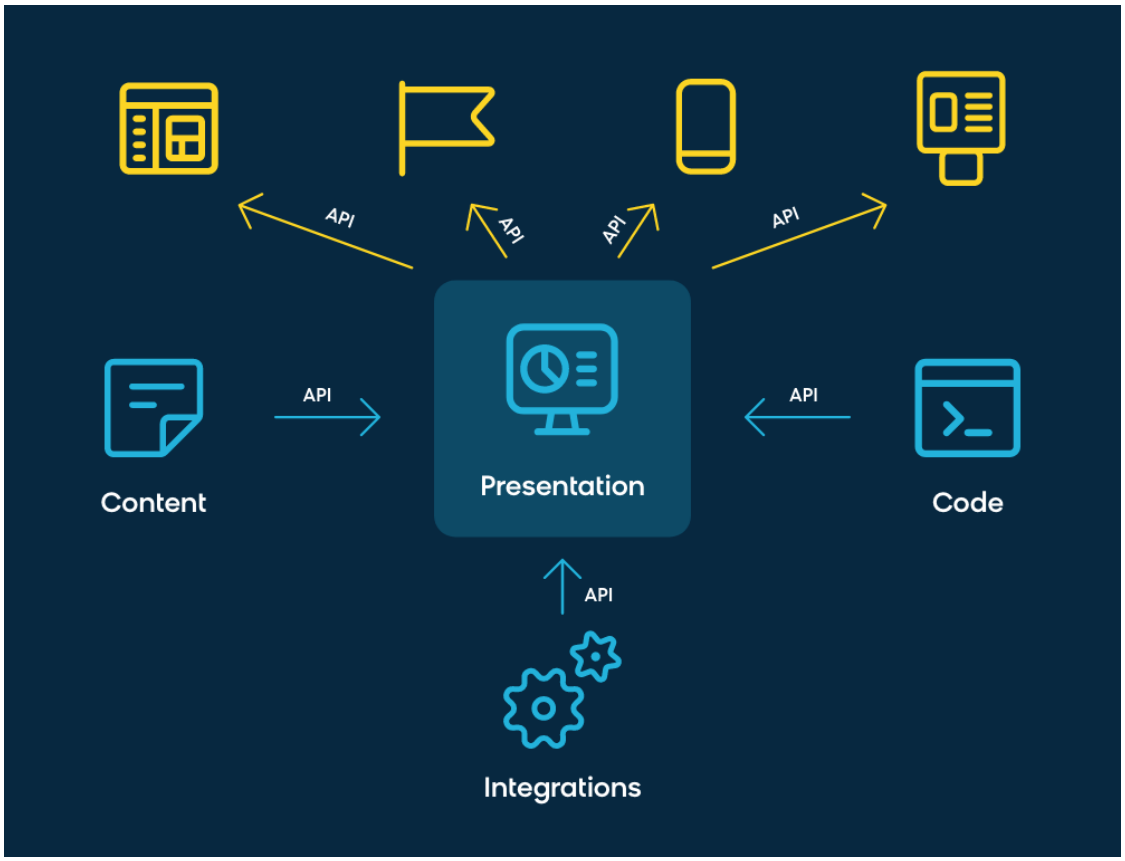


Рисунок 1.2 взаємодія SPA з API. [2]

Це налаштування API дозволяє розробникам швидко працювати з інтерфейсом користувача без ризику для важливих для бізнесу серверних технологій.

На сьогоднішній день найпопулярнішими фреймворками для розробки SPA є:

- React js
- Angular js
- Vue js

Для розробки кваліфікаційного проекту було вирішено використовувати React js, як фронтенд фреймворк для розробки SPA.

1.2 Порівняння SPA з класичними архітектурами

Можливо, у майбутньому всі будуть використовувати модель одно сторінкового додатка, оскільки вона дає багато переваг. Багато програм на

ринку переходять на цю модель. Однак, оскільки деякі проекти просто не вписуються в SPA, модель MPA, як і раніше, актуальна.

MPA – це багатосторінкові додатки (рис. 1.3). Вони працюють більш “традиційним” шляхом. Кожна зміна, наприклад. відображати дані або надсилати дані на запити сервера, відображаючи нову сторінку з сервера в браузері. Ці програми більші, більше, ніж SPA, тому що вони повинні бути такими.

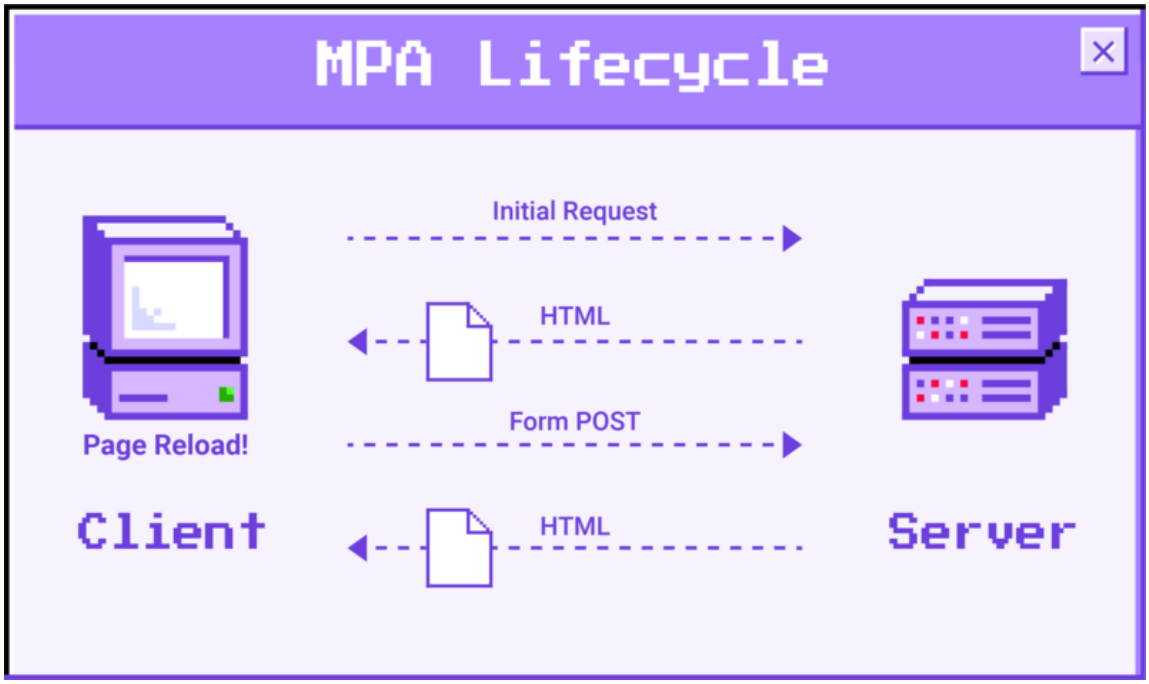


Рисунок 1.3 робота MPA. [3]

У MPA вся сторінка перезавантажується за запитом, тоді як у SPA вилучаються лише необхідні дані у форматі JSON, ініційовані JavaScript

Незважаючи на те, що SPA зараз поширені, існують проблеми з пошуковою оптимізацією (SEO), управлінням історією браузера та вирішенням проблем безпеки, пов'язаних з виконанням JavaScript. В результаті є випадки, коли MPA стає найкращим вибором.

Далі розглянемо переваги та недоліки кожної з архітектур.

У рішень SPA є багато переваг, таких як покращена продуктивність додатків, узгодженість, скорочення часу розробки та витрат на інфраструктуру.

Відокремлюючи презентацію від вмісту та даних, команди розробників можуть працювати з різною швидкістю, зберігаючи цілісність загального рішення. SPA хороший для створення адаптивного дизайну для мобільних пристроїв, комп'ютерів та планшетів.

Перевага 1. Одноразове завантаження файлів HTML, CSS, JS.

Односторінкова програма, після початкового завантаження сторінки сервер більше не надсилає вам HTML-код - ви завантажуєте все відразу на самому початку.

Сервер надсилає вам сторінку оболонки, а ваш браузер відображає інтерфейс користувача (UI).

Потім, коли ви клацаєте мишею, SPA відправляє назад запити на дані та розмітку, сервер відправляє необхідні вихідні матеріали, а ваш браузер бере їх і відображає оновлений інтерфейс - взаємозамінні частини без необхідності оновлення всієї сторінки. Ця швидка взаємозамінність робить SPA неймовірно корисними на сторінках з високою навігацією та використанням повторюваних шаблонів.

Перевага 2. Відсутність додаткових запитів до сервера.

Оскільки серверу не потрібно витрачати час та енергію на повне відображення, SPA в цілому знижують навантаження на ваші сервери: це означає, що ви можете заощадити гроші, використовуючи менше серверів для того ж обсягу трафіку.

Перевага 3. Швидкий та чуйний зовнішній інтерфейс.

Поряд із більш швидкою продуктивністю, описаною вище, SPA також дозволяють розробникам набагато швидше створювати зовнішній інтерфейс. Це пов'язано з непов'язаною архітектурою SPA або розподілом серверних служб та зовнішнього дисплея.

Перевага 4. Розширений інтерфейс користувача.

У міру того, як все більше і більше функціональних можливостей створюються як модульні сервіси (мікросервісна архітектура), які можна

оновлювати незалежно, стає простіше експериментувати з тим, як вони відображаються і використовуються.

Фреймворки SPA відмінно підходять для експериментів з цими сервісами для створення привабливих, динамічних і навіть анімованих інтерфейсів користувача.

Недолік 1. Видалено інструменти редагування, до яких звикли маркетологи.

При використанні SPA доставка визначається SPA (рис 1.4), а контент просто зберігається CMS стандартним способом, який можуть читати API. Оскільки SPA обробляється в зовнішньому інтерфейсі, серверна CMS не має жодного уявлення про те, як він має виглядати, і тому не може запустити попередній перегляд.

Рисунок 1.4 взаємодія SPA та CMS.[4]

Недолік 2. Утруднено повторне використання контенту.

Ця проблема виникає з двох основних причин: одна з них пов'язана із застарілими системами керування контентом, а інша – з дизайном SPA.

Існують певні системи управління контентом, у яких просто немає поділу між тим, як ваш контент виглядає і як він зберігається. Оскільки вміст зберігається над стандартному, нейтральному форматі представлення, SPA неспроможна використовувати його способом, заснованим на API, а хоче.

Недолік 3. Проблеми з персоналізацією/релевантністю.

SPA захоплюють контент «сервісним» способом, тому це невеликий фрагмент контенту без особливого контексту — не велика допомога в релевантній доставці.

Незважаючи на те, що SPA зараз поширені, існують проблеми з пошуковою оптимізацією (SEO), управлінням історією браузера та вирішенням проблем безпеки, пов'язаних з виконанням JavaScript. В результаті є випадки, коли MPA стає найкращим вибором.

Далі розглянемо переваги та недоліки MPA.

Перевага 1. SEO оптимізація.

MPA мають кращі можливості пошукової оптимізації ніж SPA, тому що ми можемо оптимізувати кожну сторінку за певним ключовим словом, вставляючи метатеги та відображаючи контент.

В результаті пошукові роботи зазвичай MPA краще.

Перевага 2. Більше інформації про веб-сайт.

MPA можуть отримати більше інформації від аналітичних сервісів, таких як Google Analytics порівняно з SPA.

У SPA складніше визначити, яка частина програми не працює, і ми можемо отримати лише загальні дані, такі як відвідувачі та як довго вони залишалися на веб-сайті.

Перевага 3. Легко масштабується.

Висока масштабованість MPA важлива, коли йдеться про складні та великі веб-сайти. Оскільки обмежень за кількістю сторінок немає, ви можете розробляти та випускати нові сторінки контенту з найменшим впливом на існуючий код.

Недолік 1. Фронтенд і бекенд технології тісно пов'язані.

При розробці MPA фронтент і бекенд тісно зв’язані між собою, через що стає складно розподіляти роботу між декількома людьми.

Недолік 2. Проблема мобільної версії.

Багатосторінкові сайти також складніше адаптувати, тому що немає коду, який міг би просто адаптувати їх для мобільних пристроїв. Через це мобільна версія для них має бути побудована з нуля.

SPA ідеально підходять, якщо потрібно створити динамічне рішення з обмеженим обсягом даних. Незважаючи на те, що вони працюють у браузері, SPA виглядають як настільні або мобільні програми, і тому вони такі популярні.

У свою чергу, MPA будуть безпрограшним варіантом для великих компаній, яким необхідно представити широкий спектр послуг та продуктів, і тому програмі потрібно досить багато функцій, сторінок та меню. Наприклад, якщо вам потрібно створити інтернет-магазин.

РОЗДІЛ 2
МІКРОСЕРВІСНА АРХІТЕКТУРА

2.1 Про мікросервісну архітектуру

Мікросервіси – це архітектурний стиль, де єдиний додаток розробляється як сукупність невеликих сервісів. В цей час сервіс – це найпростіша одиниця, який приймає вхідні запити для здійснення дій. Кожен з них працює окремо не залежно від інших сервісів але спілкується з іншими, використовуючи прості та швидкі протоколи даних, такі як HTTP.

Ці сервіси будуються навколо бізнес-потреб і розгортаються незалежно один від одного з використанням зазвичай повністю автоматизованого середовища. Існує абсолютний мінімум централізованого керування цими сервісами. Самі по собі вони можуть бути написані з використанням різних мов програмування і технологій зберігання даних[5].

Отже, це бекенд служба, розгорнута на сервері.

У якомусь сенсі це той самий монолітний додаток, однак він не несе в собі всю функціональність системи, а лише меншу частину логіки.

Зазвичай мікросервіси поділяють та складають з ярусів. Не існує правил щодо того, скільки і яких ярусів має бути, однак, є найкращі практики. На рисунку нижче (рис. 2.1) можна побачити найпоширеніші яруси, що використовуються в подібних архітектурах. Назви можуть відрізнятися, але структура дуже схожа на класичну багатоярусну архітектуру.

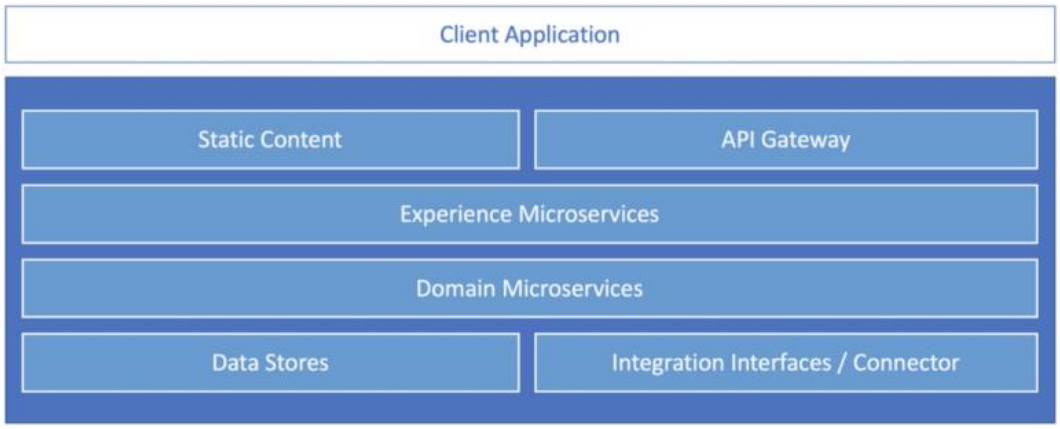


Рисунок 2.1 типова архітектура мікросервісного веб-додатку [5].

Ми можемо побачити наступні логічні яруси:

- Front-End — Client Application, Static Content
- Back-End — API Gateway, Experience Microservices, Domain Microservices
- Data & Integration — Data Stores, Integration Interfaces / Connectors

Далі розглянемо кожен ярус окремо.

1. Client Application

   При використанні веб-додатку, клієнтом, як правило є браузер. Браузер – це клієнт, що завантажує статичний контент із видаленого сервера і малює UI контент всередині себе. У той же час браузер є клієнтом, що обробляє фізичні HTTP запити, які надходять до сервера (рис. 2.2).

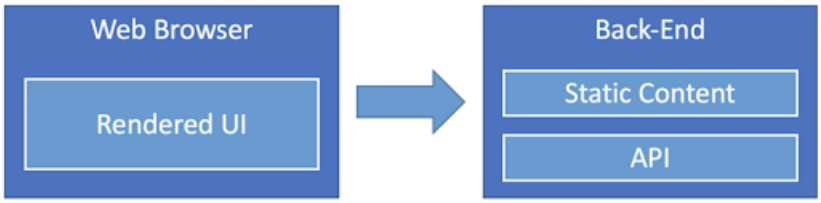


Рисунок 2.2 взаємодія браузера з сервером [5].

2. Static Content

   Найкраща практика – розділити відповідальність шляхом роз’єднання логічних компонентів, щоб жоден компонент не відповідав за все. З цієї причини рекомендується подавати статичний контент через окремий серверний компонент. Це дозволяє відокремлювати статичний контент від API.

3. API Gateway

   API Gateway — один із основних патернів, навколо якого будується мікросервісна архітектура. Це єдина точка входу для так званих Experience APIs, доступних для клієнтських програм, і є важливою складовою найкращих практик управління API (API Management). По

суті, він передає запити реальним бекенд сервісам та здатен приймати рішення. Частіше за все, API Gateway відповідає за такий функціонал, як:

- Request handling
- Upstream data transfer
- Access control
- Security policies
- Throttling
- Rate Limiting
- Analytics
- Caching

4. Experience Microservices

   При використанні неструктурованого розподіленого середовища стає складно утримувати контроль над множиною мікросервісів. З цієї причина треба контролювати ці мікросервіси, структуруючи їх за ярусами. Існує купа різних архітектурних шаблонів, які можуть допомогти. Загальним для цих шаблонів є те, що всі вони зосереджені на experience (рис. 2.3).

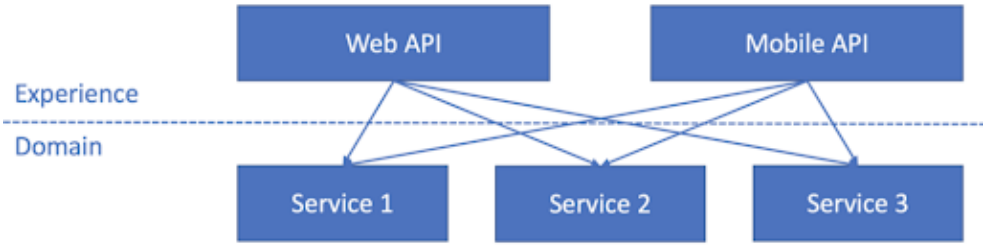


Рисунок 2.3 Experience сервіси покладаються на доменні сервіси [5].

   Ці мікросервіси утворюють experience ярус, який логічно відокремлений від інших ярусів. Він може бути керований та налаштований (масштабований, налаштований, захищений тощо) незалежно.

5. Domain Microservices

Доменні мікросервіси – це нижчий ярус, ніж ти що ми вже розглянули. Він зосереджується на бізнес-логіці, також від нього залежать experience сервіси.

Таким чином experience сервіси сфокусовані на користувачі та продукті, в той час, коли доменні мікросервіси відповідають за дані та ресурси, що знаходяться під їх управлінням.

В результаті маємо ситуацію, що додавання кожного нового experience сервісу не потребує додаткових витрат часу на повторну розробку доменної логіки.

6. Integration Interfaces / Connectors

   Окрім внутрішніх джерел даних, програмі може знадобитися доступ до інших підсистем та сторонніх API. Корисна практика – розділяти інтеграційний ярус, запровадивши окремі сервіси-конектори.

Мікросервісна архітектура по суті є патерном який вчить застосовувати різні (чи інші) архітектурні патерни, стилі та підходи в рамках єдиного додатка, розподіленого в мережі.

2.2 Порівняння з монолітною архітектурою

Далі розглянемо різницю між мікросервісною та монолітною архітектурою. На даний момент саме мікросервісна архітектура є однією з найбільш обговорюваних у галузі програмного забезпечення. Це вже зробило величезний вплив на підприємства та підприємства інформаційних технологій. Це також призвело до цифрової революції у всьому бізнесі прикладних програм, де за монолітною архітектурою широко дотримувалися всі підприємства інформаційних технологій. Більшість величезних технологічних гігантів, таких як Google, Netflix, Amazon тощо, дотримуються архітектури мікросервісів для всіх своїх застосувань. А малі підприємства здебільшого дотримуються монолітної архітектури через її простоту.

Монолітна архітектура розглядається як звичайний метод розробки додатків. Додаток в монолітній архітектурі розробляється як єдиний пакет. Розробка звичайного додатку починається з модульної шаруватої або шестикутної архітектури. Ця архітектура складається з різних типів шарів наступним чином:

- Презентаційний шар: Саме рівень графічного інтерфейсу користувача обробляє запити протоколу передачі HyperText (HTTP) за допомогою HTML або XML / JSON.
- Бізнес-логічний шар: Ділова логіка програми присутня в цьому шарі.
- Шар доступу до бази даних : У цьому шарі відбувається доступ до всіх баз даних, включаючи SQL і NoSQL додатків.
- Інтеграційний шар додатків : на цьому рівні відбувається вся інтеграція програмного забезпечення з іншими системами.

Навіть незважаючи на те, що монолітна архітектура має логічну шарувату архітектуру, остаточні програми будуть упаковані в єдиний моноліт і потім будуть розгорнуті таким чином. Монолітні програми не мають належної модульності, і він має лише єдину базу коду.

З іншого боку, архітектура мікросервісів дотримується модульного підходу до розробки різних програм. Мікросервісна архітектура містить набір невеликих, незалежних та автономних модулів, які надають різні послуги. Кожна служба повинна мати можливість незалежного впровадження відповідних бізнес-підрозділів. Монолітна архітектура - це єдине ціле. Але архітектура мікросервісів має групу невеликих незалежних підрозділів, яка спільно працює як єдине додаток. Вся функціональність програми розбита на окремі та незалежні розгорнуті модулі, які спілкуються один з одним методами під назвою інтерфейси програмування прикладних програм (API). Кожну з послуг в архітектурі мікросервісів можна легко масштабувати, розгортати та оновлювати легко.

Це архітектура, що не пов'язана між собою, кожен компонент є незалежним відносно один одного. Для їх кодування можна використовувати

кілька мов програмування. Крім того, вони можуть використовувати інший тип зберігання для зберігання даних.

Далі розглянемо деякі ключові відмінності між мікросервісом та монолітом.

По-перше, це прихильність до технології, де у мікросервісах:

- Розробники мають широкий спектр варіантів для різних технологій, таких як операційні системи, рамки, мови програмування тощо для створення програми.
- Постійне зобов'язання та залежність можуть бути усунені за допомогою одного стека технологій.
- Кращий і новий стек технологій може бути прийнятий при створенні нових служб або оновленні існуючих служб.
- Залежності команди розробників від ресурсів для створення або оновлення послуг усуваються.

У той же час у монолітній архітектурі:

- Розробники змушені використовувати лише одну технологію, незалежно від її обмежень.
- Якщо рамки вашої програми застаріли, перехід до нового, кращого складу буде дуже складним і складним.
- У таких ситуаціях розробники повинні переписати всю програму на іншій мові програмування та на новій основі, зробивши це більш ризикованим та трудомістким.

По-друге, це ізоляція несправностей, де:

Навіть якщо в будь-якому з процесів мікросервісу сталася помилка, решта процесів не впливатиме і їх можна буде запустити, оскільки всі служби незалежні та ізольовані один від одного.

Навпаки, в монолітній архітектурі будь-який вид неправильної поведінки в будь-якому з компонентів може сильно вплинути на роботу всієї програми.

По-третє, це масштабування додатку, де у мікросервісах:

- Усі послуги додатків для мікросервісів побудовані у вигляді різних модулів.
- Це призводить до розподілу команди на різні роботи, що надалі допомагає їм легко змінювати та оновлювати виробництво.
- Це полегшує масштабування програми.

У той же час у монолітній архітектурі:

- Масштабування програм - це завдання для розробників, оскільки це єдиний пакетний пакет.
- Розробникам не можна працювати над окремими модулями.
- Якщо це можливо, буде потрібна важка координація під час розгортання та розробки.

Монолітна архітектура є кращою для розробки дуже малих, простих та легких програм. Оскільки монолітна архітектура розглядається як традиційний спосіб розробки додатків, завжди краще мати хороші знання про те саме. Мікросервісна архітектура добре підходить для розробки більш складних та великих додатків[6].

РОЗДІЛ 3
ПРОЕКТУВАННЯ ДОДАТКУ

3.1 Вибір технологій

Для розробки інтерфейсу користувача було використано JavaScript бібліотеку React.

React — це декларативна, ефективна і гнучка JavaScript-бібліотека, призначена для створення інтерфейсів користувача. Вона дозволяє компонувати складні інтерфейси з невеликих окремих частин коду — “компонентів”.

React спрощує створення інтерактивних інтерфейсів. Потрібно лише описати, як різні частини інтерфейсу виглядають у кожному стані вашого додатку і React ефективно оновить та відрендерить лише потрібні компоненти, коли ваші дані зміняться.

Декларативні інтерфейси роблять код більш передбачуваним і його набагато легше налагоджувати.

React дає можливість створювати інкапсульовані компоненти, які керують власним станом, а з них будуйте складні інтерфейси.

Оскільки логіка компонентів написана на JavaScript, замість шаблонів, можна з легкістю передавати складні дані у вашому додатку і зберігати стан окремо від DOM[7].

Не дивлячись на те, що React призначений для створювання інтерфейсів для користувача, стилі потрібно прописувати самостійно або використовувати для цього спеціальні бібліотеки чи фреймворки.

У процесі розробки додатку було вирішено використовувати бібліотеку компонентів MUI.

MUI пропонує повний набір інструментів інтерфейсу користувача, які допомагають швидше передавати нові функції.

Створений та розроблений Google, Material Design – це мова дизайну, яка поєднує у собі класичні принципи успішного дизайну з інноваціями та

технологіями. Мета Google - розробити систему дизайну, що забезпечує уніфікований інтерфейс користувача для всіх їх продуктів на будь-якій платформі.

До принципів Material Design можна віднести наступні:

1. Метафора матеріалу визначає відносини між простором та рухом. Ідея полягає в тому, що технологія натхнена папером та чорнилом і використовується для сприяння творчості та інноваціям. Поверхні та краї забезпечують знайомі візуальні підказки, що дозволяють користувачам швидко зрозуміти технологію за межами фізичного світу.

2. Елементи та компоненти, такі як сітки, типографіка, колір та зображення, не лише візуально приємні, але й створюють відчуття ієрархії, значення та фокусу. Акцент на різних діях та компонентах створює візуальний посібник для користувачів.

3. Рух дозволяє користувачеві провести паралель між тим, що він бачить на екрані та в реальному житті. Забезпечуючи як зворотний зв'язок, так і знайомство, це дозволяє користувачеві повністю поринути у незнайому технологію. Рух містить узгодженість та безперервність на додаток до надання користувачам додаткової підсвідомої інформації про об'єкти та перетворення[8].

Надалі було вибрано технологію для написання бекенду, а саме мікросервісів.

Зростаюча популярність JavaScript принесла з собою безліч змін, і обличчя веб-розробки сьогодні кардинально змінилося. Речі, які ми можемо робити в Інтернеті в даний час за допомогою JavaScript, що працює на сервері, а також у браузері, було важко уявити лише кілька років тому.

Node.js (або просто Node) - це серверна платформа для роботи з JavaScript через двигун V8 (рис. 3.1). JavaScript виконує дію на стороні клієнта, а Node – на сервері. За допомогою Node можна писати повноцінні програми. Node вміє працювати із зовнішніми бібліотеками, викликати команди з коду на JavaScript та виконувати роль веб-сервера.

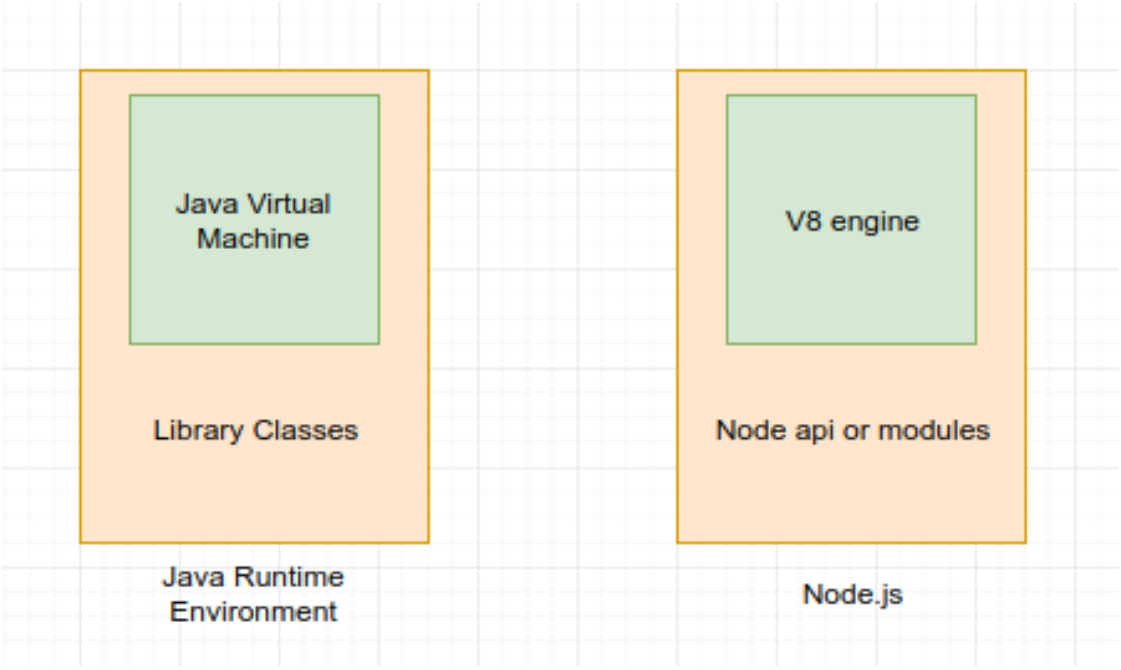


Рисунок 3.1 Node js.

Обидва – браузерний JavaScript та Node.js запускаються у середовищі виконання V8. Цей двигун використовує ваш JS код, і перетворює його на швидший машинний код. Машинний – низькорівневий код, який може запускати комп'ютер без необхідності спочатку його інтерпретувати.

Основна ідея Node.js: використовувати не блокуючий введення-висновок, керований подіями, щоб залишатися легким і ефективним перед додатками реального часу, що інтенсивно використовують дані, які працюють на розподілених пристроях.

Порівняно з традиційними методами веб-обслуговування, коли кожне з'єднання (запит) породжує новий потік, займаючи системну оперативну пам'ять і, зрештою, максимально використовуючи обсяг доступної оперативної пам'яті, Node.js працює в одному потоці, використовуючи введення/виведення, що не блокує (рис. 3.1). Що дозволяє йому підтримувати десятки тисяч одночасних з'єднань, що утримуються у циклі подій.

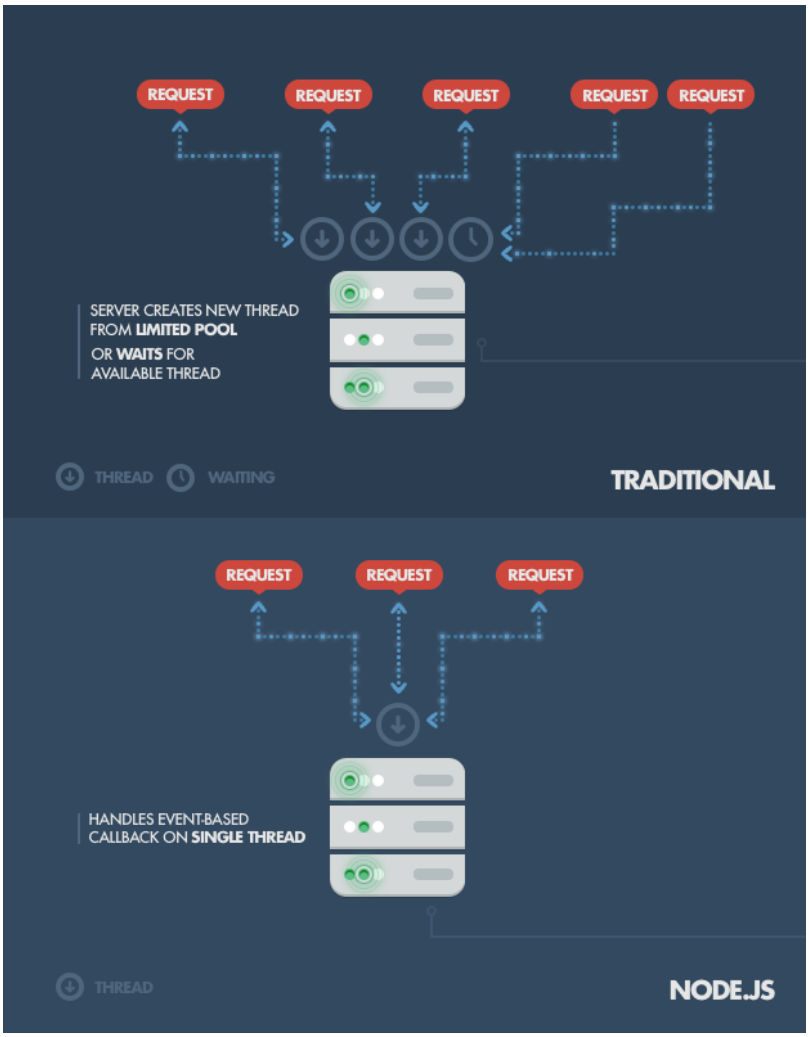


Рисунок 3.2 Робота з потоками Node js.

Хоча Node.js дійсно блискуче працює з програмами реального часу, він цілком природно підходить для надання даних з об'єктних баз даних.

У якості ORM для бази даних була обрана бібліотека Sequelize.

Sequelize – це ORM-бібліотека для додатків на Node.js, яка здійснює зіставлення таблиць у базі даних та відносин між ними з класами. При використанні Sequelize ми можемо не писати SQL-запити, а працювати з даними як із звичайними об'єктами. Причому Sequelize може працювати з СУБД - MySQL, Postgres, MariaDB, SQLite, MS SQL Server (рис. 3.3).

Чудова особливість Sequelize полягає в тому, що він не піклується про вашу базову базу даних. Ви можете легко перемикати бази даних,

налаштувавши файл конфігурації, і ваш код здебільшого залишиться незмінним.

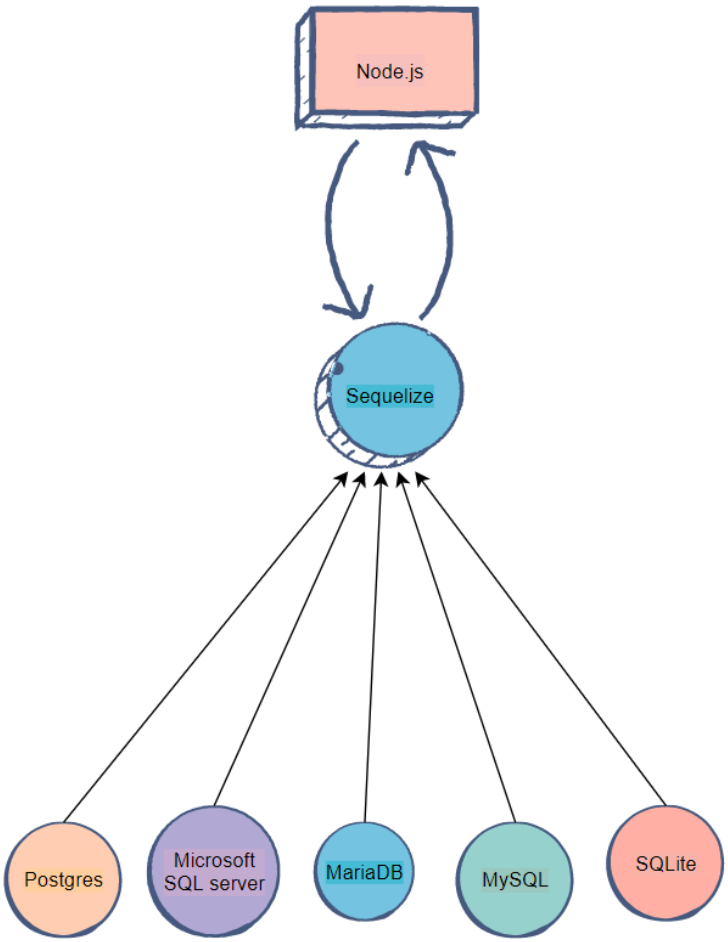


Рисунок 3.3 Взаємодія Sequelize з Node.js та базами даних.

Моделювати свою базу даних за допомогою Sequelize дуже зручно. Модель у Sequelize — це в основному представлення таблиці у вашій базі даних. Примірник моделі містить відомості про сутність, яку він представляє.

Sequelize – це все про моделі. У базі даних це наші схеми – форма, яку набувають наші дані. Ваші моделі — це як об’єкти, з якими ви будете взаємодіяти у своїй програмі, так і основні таблиці, які ви будете створювати та керувати ними у своїй базі даних.

Бази даних мають стандартні асоціації, такі як «багато до багатьох», «належить багатьом» або «багато до одного».

У кожному випадку Sequelize дає нам чіткі методи для налаштування асоціацій.

Якби ми керували нашою базою даних традиційним способом, нам потрібно було б створити окремі таблиці для зберігання наших зовнішніх ключів. Ці таблиці, які часто називають таблицями зовнішніх ключів або таблицями з'єднання, містять посилання, які з'єднують дві інші таблиці.

Наприклад, якщо ми хочемо пов'язати користувачів і облікові записи, нам може знадобитися таблиця для з'єднання UserID з accountID. Замість того, щоб зберігати їх в одній таблиці, ми створили б історію із зовнішнім ключем, яка містила б лише ідентифікатор користувача та пов'язаний з ним ідентифікатор облікового запису. Це дозволить легко отримати доступ до профілю та ідентифікатора користувача з інформацією облікового запису цього користувача.

У якості базі даних було використано PostgeSQL.

PostgreSQL – це потужна система об'єктно-реляційних баз даних з відкритим вихідним кодом, яка використовує та розширює мову SQL у поєднанні з багатьма функціями, які безпечно зберігають і масштабують найскладніші робочі навантаження даних. Витоки PostgreSQL сягають 1986 року в рамках проекту POSTGRES в Каліфорнійському університеті в Берклі і має понад 30 років активної розробки на базовій платформі.

PostgreSQL постачається з багатьма функціями, які допомагають розробникам створювати програми, адміністраторам захищати цілісність даних і створювати відмовостійкі середовища, а також допомагають вам керувати даними незалежно від того, наскільки великий чи маленький набір даних. Окрім того, що PostgreSQL є безкоштовним і відкритим вихідним кодом, він дуже розширюваний. Наприклад, ви можете визначати власні типи даних, створювати власні функції, навіть писати код з різних мов програмування без перекомпіляції бази даних.

3.2 Проектування

Зв'язок між мікросервісами має бути ефективним і надійним. Оскільки багато невеликих сервісів взаємодіють для завершення однієї ділової діяльності.

Мікросервіс може вийти з ладу з будь-якої кількості причин. Це може бути збій на рівні вузла, наприклад, апаратний збій або перезавантаження віртуальної машини. Він може вийти з ладу або бути перевантажений запитами та не зможе опрацювати нові запити. Будь-яка з цих подій може призвести до збою дзвінка. Існує два шаблони проектування, які можуть допомогти зробити мережеві дзвінки між службами більш стійкими:

- Мережевий виклик може завершитися помилкою через тимчасову помилку, яка зникне сама по собі. Замість прямої відмови сторона зазвичай повинна повторювати операцію певну кількість разів або до тих пір, поки не закінчиться налаштований період очікування. Однак, якщо операція не є ідемпотентною, повторні спроби можуть спричинити ненавмисні побічні ефекти. Початковий виклик може завершитися успішно, але абонент ніколи не отримає відповіді. Якщо об'єкт, що викликає, повторює спробу, операція може бути викликана двічі.
- Занадто велика кількість невдалих запитів може стати вузьким місцем, тому що очікувані запити накопичуються в черзі. Ці заблоковані запити можуть містити критично важливі системні ресурси, такі як пам'ять, потоки, з'єднання з базою даних тощо, що може призвести до каскадних збоїв. Шаблон переривника ланцюга може запобігти повторній спробі служби виконати операцію, яка, швидше за все, завершиться помилкою.

Також існує два основних шаблони обміну повідомленнями, які можуть використовуватися для взаємодії з іншими мікросервісами.

- Синхронний зв'язок. У цьому шаблоні служба викликає API, який надається іншою службою, використовуючи такий протокол, як HTTP або gRPC. Цей параметр є шаблоном синхронного обміну повідомленнями, оскільки сторона, що викликає, очікує відповіді від одержувача.

- Асинхронне надсилання повідомлень. У цьому шаблоні служба надсилає повідомлення, не чекаючи відповіді, і одна або кілька служб обробляють повідомлення асинхронно.

Кожен шаблон має компроміси. Запит/відповідь — добре зрозуміла парадигма, тому розробка API може бути більш природною, ніж розробка системи обміну повідомленнями. Однак асинхронний обмін повідомленнями має деякі переваги, які можуть бути корисними в архітектурі мікросервісів:

- Відправнику не потрібно знати про споживача.
- Використовуючи модель pub/sub, кілька споживачів можуть передплатити отримання подій.
- Якщо споживач виходить з ладу, відправник все ще може надсилати повідомлення. Повідомлення будуть отримані, коли споживач одужає. Ця можливість особливо корисна в архітектурі мікросервісів, оскільки кожен сервіс має свій життєвий цикл. Служба може бути недоступною або замінити нову версію в будь-який час. Асинхронний обмін повідомленнями може опрацьовувати періодичні простої.
- Вища служба може відповідати швидше, якщо вона не очікує підлеглих служб. Це особливо корисно в архітектурі мікросервісів. Якщо існує ланцюжок залежностей служби (служба A викликає службу B, яка викликає службу C тощо), очікування синхронних викликів може призвести до неприйнятних затримок.
- Черга може діяти як буфер для вирівнювання робочого навантаження, щоб одержувачі могли обробляти повідомлення зі своєю швидкістю.
- Черги можна використовувати для керування робочим процесом, позначаючи повідомлення після кожного кроку робочого процесу.

Однак існують деякі проблеми з ефективним використанням асинхронного обміну повідомленнями:

- Використання певної інфраструктури обміну повідомленнями може призвести до жорсткого зв'язку з цією інфраструктурою. Пізніше буде складно перейти на іншу інфраструктуру обміну повідомленнями.
- Наскрізна затримка операції може стати високою, якщо черги повідомлень переповняться.
- За високої пропускної спроможності грошові витрати на інфраструктуру обміну повідомленнями може бути значними.

Наступним кроком було створення візуального представлення бази даних.

РОЗДІЛ 4
РЕАЛІЗАЦІЯ ПРОЕКТУ

4.1 Поділення проекту на віхи і задачі

Для планування та управління розробки проекту, його було поділено на віхи та завдання. Це дозволяє більш ефективно реалізовувати стратегічні ініціативи. Віхи спрощують вашу роль як лідер проекту, тому що вони показують позначки, на які йому потрібно орієнтуватися, а зацікавленим сторонам — прогрес у важливих для них областях.

Віхи позначають певні точки на хронології проекту. Ці контрольні точки показують, коли закінчуються окремі дії (групи дій) чи починається нова стадія. Ви можете відокремлювати віхи від інших елементів хронології, оскільки самі по собі вони не забирають час - це, швидше, дорожні покажчики, що допомагають не збитися зі шляху.

Віхи - це потужний інструмент, що дозволяє відстежувати хід реалізації проекту. Вони допомагають мотивувати та тримати в курсі подій, дозволяючи кожному бачити шкалу прогресу та оцінювати пріоритети. Крім того, за віхами також можна контролювати терміни, визначати важливі дати та виявляти перешкоди на шляху до виконання проекту. Навіть якщо прибрати всі завдання з хронології проекту, віхи, як і раніше, будуть давати загальне уявлення про його ключові етапи або стадії.

Вони часто збігаються з початком та закінченням стадій проекту, таких як початкове обговорення, планування, виконання та завершення. Одна така стадія може тривати тижнями і навіть місяцями, включати безліч завдань, над якими працює великий колектив.

В той самий час задачі - це будівельні блоки вашого проекту, і на їхнє завершення потрібен час. У віх немає тривалості. Вони є лінії на піску, які вказують на те, що було завершено групу задач.

Віхи можна додати до будь-якого проектного плану для полегшення його виконання. Вони можуть бути особливо корисними, коли йдеться про

хронологічний графік, тому що віхи виставляються поруч із відповідними завданнями або стадіями.

Так як віхи - це моменти часу, які не призначені для відстеження процесів, що ведуть до них, слід пов'язувати терміни досягнення віх із запуском або реалізацією конкретних ініціатив.

Відстеження віх дозволяє аналізувати найважливішу роботу та бачити справжній стан проектів, надаючи при цьому інформацію про перебіг робіт за проектом, якою можна впевнено поділитися.

Виходячи з цього було розроблено план робот (рис. 4.1).

Опис віхи	Дата початку	Статус
Ініціатива проекту	15.08.2021	Виконано
Пошук вже існуючих аналогів	02.09.2021	Виконано
Порівняння та виділення найкращого	22.09.2021	Виконано
Визначення основних ідей та концепцій додатку	05.10.2021	Виконано
Визначення технологій розробки	26.10.2021	Виконано
Визначення структури бази даних	10.11.2021	Виконано
Розробка серверної частини проекту	14.12.2021	У процесі
Розробка клієнтської частини проекту	08.02.2022	У процесі

Рисунок 4.1 План проекту.

У кожному проекті можна виділити основні ознаки:

Наявність цілі. Кожен проект має конкретну мету, досягнення якою є однією з умов успішного завершення проекту.

Наявність змін. Реалізація проекту завжди пов'язана із змінами деякої системи та є її цілеспрямованим перекладом з існуючого в деякий бажаний стан. Таким чином, проект є процесом перекладу системи з існуючого в бажаний стан певної вибраної траєкторії (концепції здійснення проекту).

Обмеженість у часі, або тимчасовість. Ознака «Обмеженість у часі» означає, що будь-який проект має чіткий початок та чітке завершення. Завершення настає, коли або досягнуто мети проекту, або керівництво проекту приходить до висновку, що цілі проекту не будуть або не можуть бути досягнуті, або зникла потреба у проекті, і він припиняється.

Неповторність або унікальність. У результаті реалізації проекту створюються унікальні продукти, послуги чи результати:

- товар, тобто. кінцевий виріб або частина іншого виробу;
- послуга або здатність надавати послугу;
- результат, такий як наслідки, до яких привів проект або документи, розроблені під час реалізації проекту;

Унікальність є важливою характеристикою результатів проекту, при цьому рівень унікальності може бути різним.

Неповторність відноситься не до окремих складових частин проекту, а до проекту загалом. Однак навіть у проектах з високим ступенем новизни, безсумнівно, є процеси, які не характерні тільки для цього проекту, але й використовуються в багатьох інших проектах.

Комплексність та розмежування. Комплексність проекту означає обов'язковий облік усіх внутрішніх та зовнішніх факторів, прямо або опосередковано впливають на процес виконання та результати проекту. У те водночас кожен проект має чітко визначені рамки своєї предметної області і має бути відокремлений від інших проектів чи підприємств.

Специфічна організація проекту. Будь-який проект виконується в рамках організаційної структури, що створюється лише на час виконання проекту. Після завершення проекту ця структура розформовується. Більшість великих проектів не може бути виконано в рамках існуючих організаційних структур та вимагає на час реалізації проекту створення проекту деякої організаційної структури. Водночас для окремих дрібних або щодо простих проектів створення спеціальної організації не потрібне та/або не виправдане та проект виконується тимчасовим колективом, який називається командою проекту[9].

4.2 Розробка бекенду

Як вже було описано для серверної частини додатку у якості інструмента розробки вибрано Node.js з Express.

Додаток поділено на мікросервіси, першим з яких було розроблено AuthService(сервіс автентифікація та авторизації). Його реалізовано за допомогою JWT токенів.

Додаток використовує JWT для перевірки аутентифікації користувача в такий спосіб:

1. Спершу користувач заходить на сервер автентифікації за допомогою автентифікаційного ключа.
2. Потім сервер автентифікація створює JWT і відправляє його користувачеві(рис. 4.2).
3. Коли користувач робить запит до API додатка, він додає до нього отриманий раніше JWT.
4. Коли користувач робить API запит, додаток може перевірити за поданою із запитом JWT є користувач тим, за кого себе видає. У цій схемі сервер додатку налаштований так, що зможе перевірити, чи є вхідний JWT саме тим, що був створений сервером автентифікації.

```
class UseToken {
    generateJWT(payload) {
        const accessToken = jwt.sign(
            payload,
            process.env.JWT_ACCESS_SECRET_KEY,
            {
                expiresIn: process.env.JWT_ACCESS_EXPIRES_IN
            });

        const refreshToken = jwt.sign(
            payload,
            process.env.JWT_REFRESH_SECRET_KEY,
            {
                expiresIn: process.env.JWT_REFRESH_EXPIRES_IN
            });

        return {
            accessToken,
            refreshToken
        }
    }
```

Рисунок 4.2 Створення JWT сервером.

JWT складається з трьох частин: заголовок header, корисні дані payload та підпис signature.

Було використано JWT, який підписаний за допомогою HS256 алгоритму і тільки сервер автентифікації і сервер додатка знають секретний ключ. Сервер програми отримує секретний ключ від сервера автентифікації під час установки автентифікаційних процесів. Оскільки додаток знає секретний ключ, коли користувач робить API-запит з доданим до нього токеном, додаток може виконати той же алгоритм підписування до JWT. Додаток може потім перевірити цей підпис, порівнюючи її зі своєю власною. Якщо підписи збігаються, то JWT валідний, тобто прийшов від перевіреного джерела. Якщо підписи не збігаються, значить щось пішло не так - можливо, це є ознакою потенційної атаки. Таким чином, перевіряючи JWT, додаток додає довірчий шар між собою і користувачем.

Після реєстрації користувачеві треба підтвердити свій акаунт за допомогою електронної пошти.

Для цього сервіс після вдалої реєстрації відправляє користувачеві лист на вказану електрону адресу з посиланням для активації акаунту (рис. 4.3).

```
async sendActivationMail(email, link) {
    await this.transporter.sendMail({
        from: process.env.SMTP_USER,
        to: email,
        subject: 'Account activation ' + process.env.CLIENT_URL,
        text: '',
        html:
            `
                <div>
                    <h1>To activate your profile, follow the link: </h1>
                    <a href="${link}">${link}</a>
                </div>
            `
    });
}
```

Рисунок 4.3 Відправлення листа для активації акаунта сервером.

Якщо акаунт користувача не активований, то він не може користуватися майже всіма функціями додатку. Це зроблено щоб захистити додаток від потенціальних “ботів”, які б могли навантажувати сервер.

Для цього було використано бібліотеку “nodemailer”. Це модуль для програм Node.js, що дозволяє легко надсилати електронні листи.

Другий сервіс, що було розроблено це TaskService (сервіс для управлінням завданнями).

На даний момент він надає такі можливості, як додати, редагувати, видалити завдання та пошук їх по ключу або повернення всіх, що належать до поточного користувача.

Вже за допомогою цих двох сервісів на стороні фронтенду є можливість створення планувальника задач та ToDo списків.

Третій реалізований сервіс розширює ці можливості, це GoalService (сервіс для управління цілями). Він надає такі можливості, як додати,

редагувати, видалити цілі, пошук їх по ключу або повернення всіх, що належать до поточного користувача та додати задачу до цілі або видалити.

4.3 Розробка фронтенду

Як вже було описано для клієнтської частини додатку у якості інструмента розробки вибрано React.js та бібліотеку компонентів MUI.

Відповідно до серверної частини розробку було розпочато з автентифікації та авторизації. Для цього користувач має дві форми, перша для реєстрації, а друга для авторизації.

Для валідації форм було вирішено використовувати бібліотеку Formik. Це найпопулярніша у світі бібліотека форм із відкритим вихідним кодом для валідації форм у React та React Native. Також, для більш зручної взаємодії між формами та бібліотекою був написаний власний хук.

Хуки – це нововведення в React 16.8, яке дозволяє використовувати стан та інші можливості React без написання класів.

Власний хук має початкові значення, правила валідації та функцію, яка визивається при подачі форми (рис. 4.4).

```
const loginValidationSchema = yup.object({
    email: yup
        .string()
        .trim()
        .email('Enter a valid email')
        .required('Email is required'),
    password: yup
        .string()
        .trim()
        .min(4, 'Password must be at least 4 and no more than 16 characters')
        .max(16, 'Password must be at least 4 and no more than 16 characters')
        .required('Password is required')
});

const onSubmit = async (values: ILoginFormFields, onSubmitProps: any): Promise<void> => {
    await dispatch(login({ email: values.email, password: values.password }))
        .unwrap()
        .then((originalPromiseResult) => {
            navigate('/');
        })
        .catch((error) => {
            if (error.param) onSubmitProps.setErrors({ [error.param]: error.msg });
        });
}
```

Рисунок 4.4 Хук useLoginForm.

Це дозволяє динамічно вказувати користувачеві на помилки при заповненні форми до відправлення її на сервер (рис. 4.5).

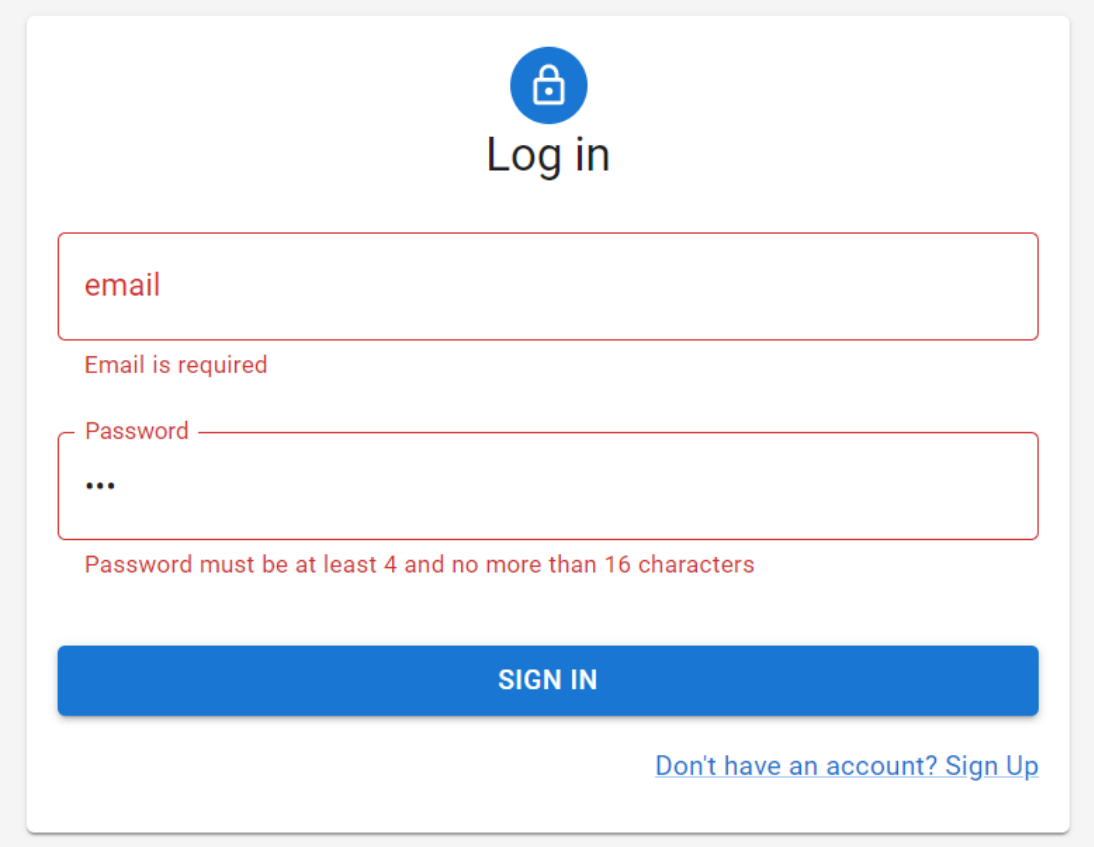


Рисунок 4.5 Динамічна валідації форми логіна.

Після того, як користувач авторизувався, його дані зберігаються у cookie та localstorage відповідно. Також для управління станом додатку застосовується Redux Toolkit.

Redux — бібліотека JavaScript з відкритим вихідним кодом, призначена для управління станом програми. Найчастіше використовується у зв'язку з React або Angular для розробки клієнтської частини. Містить ряд інструментів, що дозволяють спростити передачу даних сховища через контекст. Він працює за тим самим принципом, як і функція reduce, один із концептів функціонального програмування.

Зараз проект має два стана Redux, для особистих даних користувача, які він надає та списку задач відповідно. Це потрібно, щоб кожен раз коли

користувач переходить по сторінкам додатку не робити зайвих запитів до сервера, що тим самим додатково навантажує його. Також це дозволяє підвищити швидкість та плавність при користуванні додатку.

На даному етапі розробки після авторизації користувач має можливість зайти до свого профілю, та подивитись список завдань.

Список завдань виглядає як список блоків, які надають можливість управління завданнями тим самим користуватися всіма функціями TaskService.

ВИСНОВОК

Для виконання поставлених завдань було проведено аналіз актуальних методів та інструментів проектування SPA з використанням мікросервісною архітектури. При підготовці до розробки проекту було приділено увагу окремим особливостям роботи з фреймворками мови програмування JavaScript та Typescript.

Було проведено порівняльний аналіз вже існуючих аналогів додатку, зокрема обсягу їх можливостей.

На основі проведеного аналізу розроблено базові вимоги щодо можливостей додатку та його інтерфейсу.

При розробці проекту використовувалася система контролю версій git з публічними репозиторіями на сервісах BitBucket, що дозволяє слідувати сучасним методам розробки програмного забезпечення.

Надалі планується підтримка додатку, його наповнення контентом, так само планується реалізація інших сервісів, що залишилися та написання нових, додавання модуля багатомовності та темізації для сайту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ivan Palii. What Is Single Page App SEO? [Електронний ресурс] / Ivan Palii – Режим доступу до ресурсу: https://sitechecker.pro/single-page-application-seo/.

2. What are Single Page Applications and Why Do People Like Them so Much? [Електронний ресурс] – Режим доступу до ресурсу: https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application .

3. Single-Page Apps vs Multiple-Page Web Apps the prior written consent of owners [Електронний ресурс] – Режим доступу до ресурсу: https://agilie.com/blog/single-page-apps-vs-multiple-page-web-apps .

4. Headless CMS Is the Future of Content Management [Електронний ресурс] – Режим доступу до ресурсу: https://www.sam-solutions.com/blog/headless-cms-vs-traditional-cms/ .

5. Мікросервісна архітектура для початківців. Частина I [Електронний ресурс] – Режим доступу до ресурсу: https://www.globallogic.com/ua/insights/blogs/microservices-architecture-for-beginners-part-one/ .

6. Мікросервіс проти моноліту [Електронний ресурс] – Режим доступу до ресурсу: https://uk.education-wiki.com/1235570-microservice-vs-monolithic .

7. Посібник: знайомство з React [Електронний ресурс] – Режим доступу до ресурсу: https://uk.reactjs.org/tutorial/tutorial.html#what-is-react .

8. Learn about Material Design [Електронний ресурс] – Режим доступу до ресурсу: https://materializecss.com/about.html .

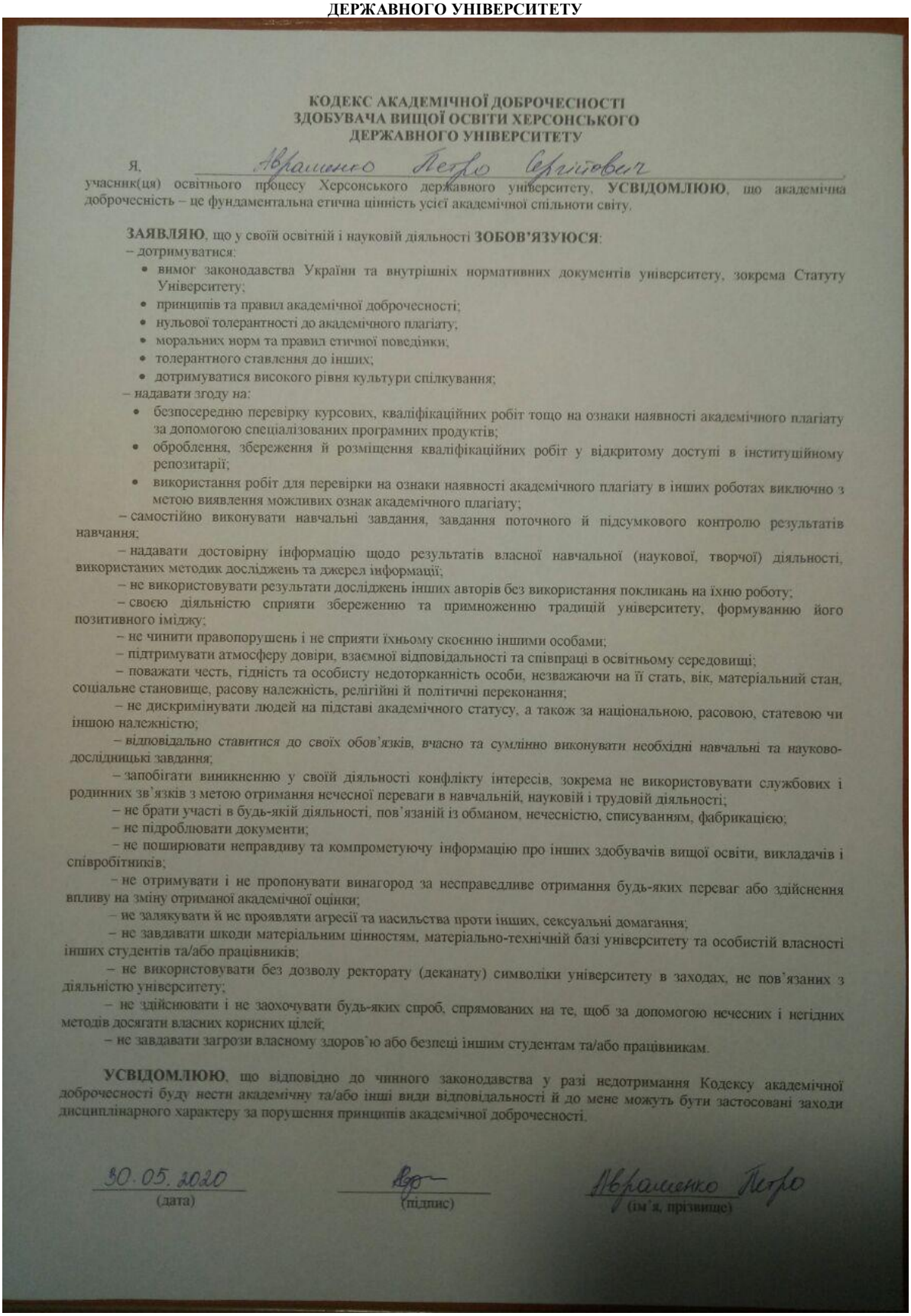
9. М. В. Грачева. УПРАВЛЕНИЕ ПРОЕКТАМИ / М. В. Грачева, С. Я. Бабаскин., 2017.

10. Redux A Predictable State Container for JS Apps [Електронний ресурс] – Режим доступу до ресурсу: https://redux.js.org/ .

ДОДАТКИ

Додаток А

КОДЕКС АКАДЕМІЧНОЇ ДОБРОЧЕСНОСТІ ЗДОБУВАЧА ВИЩОЇ ОСВІТИ ХЕРСОНСЬКОГО ДЕРЖАВНОГО УНІВЕРСИТЕТУ



КОДЕКС АКАДЕМІЧНОЇ ДОБРОЧЕСНОСТІ
ЗДОБУВАЧА ВИЩОЇ ОСВІТИ ХЕРСОНСЬКОГО
ДЕРЖАВНОГО УНІВЕРСИТЕТУ

Я, Іваненко Петро Сергійович
учасник(ця) освітнього процесу Херсонського державного університету, **УСВІДОМЛЮЮ**, що академічна доброчесність – це фундаментальна етична цінність усієї академічної спільноти світу.

ЗАЯВЛЯЮ, що у своїй освітній і науковій діяльності **ЗОБОВ'ЯЗУЮСЯ**:

– дотримуватися:
- вимог законодавства України та внутрішніх нормативних документів університету, зокрема Статуту Університету;
- принципів та правил академічної доброчесності;
- нульової толерантності до академічного плагіату;
- моральних норм та правил етичної поведінки;
- толерантного ставлення до інших;
- дотримуватися високого рівня культури спілкування;

– надавати згоду на:
- безпосередню перевірку курсових, кваліфікаційних робіт тощо на ознаки наявності академічного плагіату за допомогою спеціалізованих програмних продуктів;
- оброблення, збереження й розміщення кваліфікаційних робіт у відкритому доступі в інституційному репозитарії;
- використання робіт для перевірки на ознаки наявності академічного плагіату в інших роботах виключно з метою виявлення можливих ознак академічного плагіату;

– самостійно виконувати навчальні завдання, завдання поточного й підсумкового контролю результатів навчання;

– надавати достовірну інформацію щодо результатів власної навчальної (наукової, творчої) діяльності, використаних методик досліджень та джерел інформації;

– не використовувати результати досліджень інших авторів без використання покликань на їхню роботу;

– своєю діяльністю сприяти збереженню та примноженню традицій університету, формуванню його позитивного іміджу;

– не чинити правопорушень і не сприяти їхньому скоєнню іншими особами;

– підтримувати атмосферу довіри, взаємної відповідальності та співпраці в освітньому середовищі;

– поважати честь, гідність та особисту недоторканність особи, незважаючи на її стать, вік, матеріальний стан, соціальне становище, расову належність, релігійні й політичні переконання;

– не дискримінувати людей на підставі академічного статусу, а також за національною, расовою, статевою чи іншою належністю;

– відповідально ставитися до своїх обов'язків, вчасно та сумлінно виконувати необхідні навчальні та науково-дослідницькі завдання;

– запобігати виникненню у своїй діяльності конфлікту інтересів, зокрема не використовувати службових і родинних зв'язків з метою отримання нечесної переваги в навчальній, науковій і трудовій діяльності;

– не брати участі в будь-якій діяльності, пов'язаній із обманом, нечесністю, списуванням, фабрикацією;

– не підроблювати документи;

– не поширювати неправдиву та компрометуючу інформацію про інших здобувачів вищої освіти, викладачів і співробітників;

– не отримувати і не пропонувати винагород за несправедливе отримання будь-яких переваг або здійснення впливу на зміну отриманої академічної оцінки;

– не залякувати й не проявляти агресії та насильства проти інших, сексуальні домагання;

– не завдавати шкоди матеріальним цінностям, матеріально-технічній базі університету та особистій власності інших студентів та/або працівників;

– не використовувати без дозволу ректорату (деканату) символіки університету в заходах, не пов'язаних з діяльністю університету;

– не здійснювати і не заохочувати будь-яких спроб, спрямованих на те, щоб за допомогою нечесних і негідних методів досягати власних корисних цілей;

– не завдавати загрози власному здоров'ю або безпеці іншим студентам та/або працівникам.

УСВІДОМЛЮЮ, що відповідно до чинного законодавства у разі недотримання Кодексу академічної доброчесності буду нести академічну та/або інші види відповідальності й до мене можуть бути застосовані заходи дисциплінарного характеру за порушення принципів академічної доброчесності.

30.05.2020 (дата) _____ (підпис) Іваненко Петро (ім'я, прізвище)