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

ПРОЄКТУВАННЯ ТА РОЗРОБЛЕННЯ СЕРВІСНОЇ АРХІТЕКТУРИ УПРАВЛІННЯ БІЗНЕС-ПРОЦЕСАМИ УНІВЕРСИТЕТУ. ВІЗУАЛІЗАЦІЯ ТА ПОКРАЩЕННЯ ФУНКЦІОНАЛЬНИХ ІНСТРУМЕНТІВ ВЕБ-ПЛАТФОРМИ

Кваліфікаційна робота (проєкт)
на здобуття ступеня вищої освіти «магістр»

Виконав: здобувач
спеціальності: 121 Інженерія програмного забезпечення
Освітньо-професійної програми:
Інженерія програмного забезпечення
Валяєв Кирило Віталійович
Керівник: доктор пед. н., професор
Співаковський Олександр Володимирович
Рецензент:
Senior Software Engineer DataArt
Арутюнян Аветик Рач'яевич

Івано-Франківськ – 2022

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ЗВО	Заклад вищої освіти
COVID	Corona Virus Desease
CMS	Content Management System
API	Application Programming Interface
URL	Uniform Resource Locator
UX	User Experience
ОС	Операційна система
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
IT	Інформаційні технології
HTTP	HyperText Transfer Protocol
VLSI	Very Large Scale Integration
VLIW	Very Long Instruction Word
МРТ	Магнітно-Резонансна Томографія
GPU	Graphics Processing Unit
CSS	Cascading Style Sheets
MVP	Minimum Viable Product
ЦП	Центральний процесор
SVG	Scalable Vector Graphics
DOM	Document Object Model
SPA	Single Page Application
XHR	XML Http Request
YAML	YAML Ain't Markup Language
JSON	JavaScript Object Notation

ВСТУП

На сьогоднішній день більшість компаній та установ прагнуть частково або повністю цифровізувати свої послуги, оскільки це підвищує спроможність, стабільність та гнучкість у їх наданні. Такий спосіб не тільки пришвидшує та спрощує бізнес-процеси, але і додає їм можливості функціонування в повній мірі, особливо під час нестабільності, такої як наприклад глобальна пандемія на «COVID-19» або війна. Такі явища ставлять під загрозу не тільки певні аспекти у наданні послуг але і у спроможності в цілому, що є неприпустимим. Варто зауважити, що всесвітня тенденція на цифровізацію існує на усіх рівнях складності бізнес-процесів, від локального, місцевого рівня до більш глобального, державного.

Гарним прикладом такої цифровізації є «Дія» - електронний сервіс з надання державних послуг, розроблений Міністерством цифрової трансформації України [1], мета якого цифривізувати 100% процесів, що надають державні органи влади. Це має оптимізувати функціонування державних установ, полегшити та прискорити отримання послуг особами та зменшити рівень корупції, що еквівалентно підвищенню прибутковості держави, як керуючої бізнес-системи.

Так само і заклади вищої освіти, що є установами з надання освітніх послуг, ґрунтуються на бізнес-процесах, що задовільняють інтереси зацікавлених осіб. Війна, як дестабілізаційне явище, особливо сильно вплинула на організаційні процеси університету та стала каталізатором розвитку власної системи-агрегатора у мережі веб ХДУ24, що реалізує у собі послуги та організаційні процеси, задля забезпечення якісної та ефективної освітньої діяльності установи на усіх її рівнях.

Під час розробки таких систем особливо важливо приділяти увагу клієнтській частині додатку, оскільки саме з нею взаємодіє користувач. Неякісний та інтуїтивно незрозумілий інтерфейс може тільки нашкодити

досвіду користування системою та призведе до безсенсовної втрати як організаційних, так і користувацьких ресурсів. Важливу роль потрібно приділяти користувацькій частину – відстежувати помилки у інтерфейсі, оптимізовувати функціонал де це є необхідно, стежити за розподіленням ресурсів та мінімізувати витратність системи.

З огляду на вищезазначене, **актуальність** роботи підтверджується необхідністю швидкого управління бізнес-процесами ЗВО Херсонський державний університет з використанням єдиної централізованої системи-агрегатора, яка реалізує в собі можливість повної тристороньої взаємодії між студентами, професорсько-викладацьким складом та адміністрацією та полегшує її шляхом цифровізації та оптимізації процесів. Враховуючи постійне зростання попиту на користування інтернет-сервісами та їх зручність, доступність та гнучкість, розробка подібної системи є напрочуд актуальною.

Метою роботи є підвищення ефективності роботи системи ХДУ24 шляхом оптимізації існуючих процесів та покращення інтерфейсу користувача, що дозволило б зменшити час на взаємодію з платформою та зменшило споживання серверних та користувацьких ресурсів, а також розширення функціональності системи шляхом створення нових інструментів.

Об’єктом роботи є клієнтська частина системи ХДУ24 та шляхи її оптимізації.

Предметом роботи є дослідження процесів клієнтської частини системи ХДУ24, а також методів та засобів розробки та оптимізації інформаційних систем.

Завдання роботи:

1. Уніфікувати та оптимізувати пакет іконок користувацького інтерфейсу системи.
2. Усунути зайві залежності від сторонніх бібліотек.
3. Оптимізувати завантаження виконуваного коду користувачем.

4. Реалізувати функціональний модуль опитувань.
5. Реалізувати функціональний модуль документообігу.
6. Реалізувати функціональний модуль скорочення посилань.
7. Реалізувати функціональний модуль сервісів системи.

У процесі роботи використовувалися такі **методи дослідження** як аналіз, абстрагування, узагальнення, синтез та порівняння.

Структура роботи: робота складається з переліку умовних позначень, вступу, трьох розділів, висновків, списку використаних джерел та додатків.

РОЗДІЛ 1
ОПТИМІЗАЦІЯ ВЕБ-ДОДАТКІВ

1.1. Поняття оптимізації. Рівні оптимізації

Оптимізація продуктивності програм і програмного забезпечення - це процес модифікації програмної системи, щоб зробити її ефективнішою та швидшою. Оптимізація є ключовою для ефективної роботи програми. Вона здійснюється шляхом моніторингу та аналізу продуктивності програми та визначення шляхів її покращення.

Оптимізація продуктивності зазвичай зосереджується на покращенні лише одного або двох аспектів продуктивності системи, наприклад, часу виконання, використання пам’яті, дискового простору, пропускної здатності тощо. Зазвичай це вимагає компромісу, коли один аспект реалізується за рахунок інших. Наприклад, збільшення розміру кешу покращує продуктивність під час виконання, але також збільшує споживання пам’яті. Оптимізація коду неофіційно позначає програму, яка трансформується в кращу форму, тобто продуктивність програмного коду під час виконання покращується без зміни його функціональності.

Обсяг пам'яті для коду не є проблемою для більшості практичних цілей, однак споживання кешу інструкцій може вплинути на швидкість. Ефективність під час виконання може передбачати зменшення кількості інструкцій, заміну дорогих інструкцій дешевшими та впровадження векторизації, багатопотоковості та паралелізму.

Після розробки ефективного алгоритму з точки зору нотації Big-O [2], код можна оптимізувати за допомогою профілювання виконання, тобто вимірювання гарячих точок після автоматичної оптимізації потужними компіляторами за допомогою відповідних прапорів оптимізації компілятора.

Розглянемо основні рівні та методи оптимізації програм.

Алгоритмічна ефективність і нотація Big-O

З точки зору алгоритмічної ефективності, компілятор може вдосконалювати лише постійні фактори. Отже, перший крок до оптимізації коду - це вибрати ефективний алгоритм у виконуваному коді. Нотація Big-O використовується для класифікації алгоритмів за їхньою продуктивністю (час обробки та/або вимоги до пам’яті), коли розміри вхідних даних (n) змінюються, тобто вона характеризує алгоритм відповідно до їх швидкості зростання. Алгоритм сортування "швидке сортування", позначений як O(nlogn), завжди ефективніший, ніж "сортування вибором", позначений як O(n2), незалежно від зусиль з оптимізації коду (див. рис. 1.1).

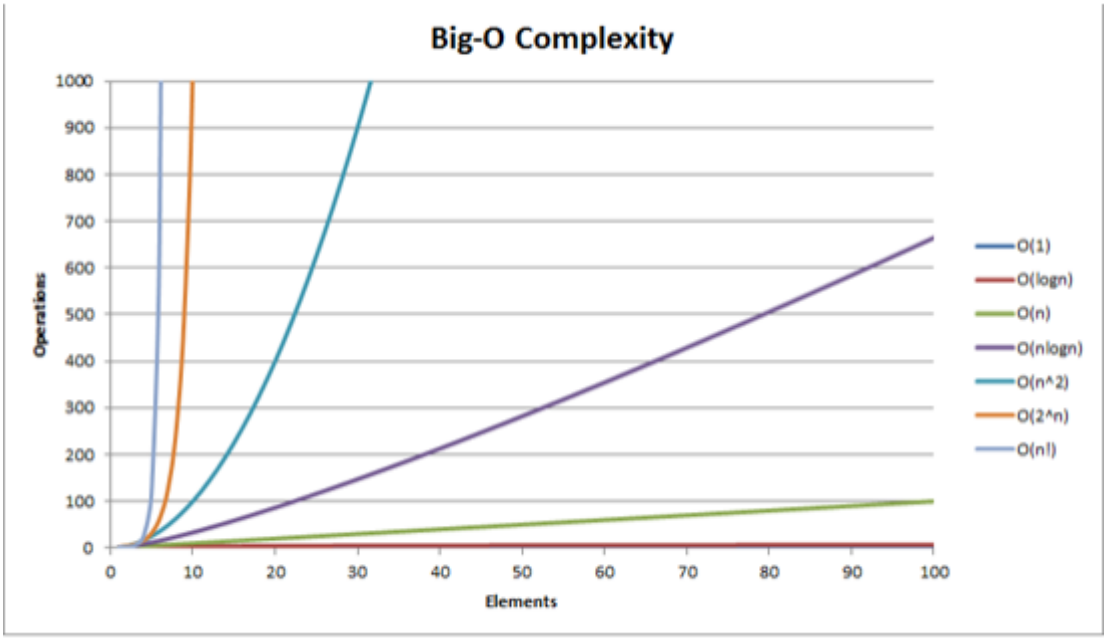


Рис. 1.1. Візуалізація нотації Big-O

Профілювання коду

Профілювання коду допомагає виявити алгоритмічні вузькі місця (гарячі точки), а також накладні витрати на зв’язок у разі розподіленого коду. Більшість часу виконання може витрачатися на функцію, яка постійно викликає інші повільніші функції, або в коді можуть бути повільніші частини. Петлі є найпоширенішим джерелом гарячих точок через їх повторюваний характер. Розглянемо цикл нижче:

```
for (int i=0; i<4; i++){
    sum+=array[i];
}
```

Він складається з 1- присвоєння, 4-порівнянь, 4-збільшень, трьох гілок і одного хибного передбачення гілок. Цикл також може додавати додаткові залежності (дані та пам’ять), навіть якщо процесор здатний внутрішньо розгортати цикл для паралелізму інструкцій.

Паралелізм

Паралелізм можна виділити на 2-х рівнях:

1. На бітовому рівні - з удосконаленням технології VLSI розмір слова процесора збільшувався, що призвело до підвищення продуктивності. Наприклад, 16-розрядний процесор може додати два 16-розрядних цілих числа швидше, ніж 8-розрядний процесор, оскільки є накладні витрати на розділення 16-розрядного цілого числа на два старші та нижні біти.

2. Рівня інструкцій - це досягається за допомогою конвеєрної обробки інструкцій. Компілятори та конструкції процесорів допомагають перекривати виконання кількох інструкцій або змінювати порядок виконання, щоб мінімізувати залежності між різними операціями. Техніки паралелізму рівня інструкцій такі:

Явно паралельне обчислення інструкцій (EPIC) використовує кілька блоків виконання (схеми) для кількох інструкцій, які виконуються паралельно. Суперскалярна архітектура виконує більше ніж одну інструкцію протягом одного тактового циклу, відправляючи кілька інструкцій, а архітектура VLIW дозволяє програмам, які можуть явно вказувати інструкції, виконуватися одночасно. Виконання не за порядком/перейменування реєстру дозволяє виконувати декілька інструкцій не по порядку; залежності даних, про які піклується функція

перейменування реєстру. Це запобігає непотрібній серіалізації програмних операцій, викликаній повторним використанням регістрів цими операціями. Спекулятивне виконання дозволяє виконувати повну або частину інструкцій, незалежно від того, має це відбутися чи ні. Передбачення гілок використовує спекулятивне виконання.

Векторизація

Це особливий випадок SIMD (одна інструкція з кількома даними), визначеного в таксономії архітектури Флінна, де один вектор або інструкція SIMD здатна працювати з кількома елементами даних паралельно. Сучасний чіп x86 має архітектуру набору інструкцій Streaming SIMD Extension (SSE). Автовекторизація виконується автоматично компіляторами після ввімкнення відповідних прапорів компілятора; O2 або вище в компіляторі Intel. У явній векторизації код SIMD може бути створений вручну для конкретного застосування. Компілятор векторизації перетворює цикл, як показано нижче:

```
for (int i=0; i<n; i++)
     c[i] = a[i] + b[i];
```

у послідовність векторних операцій, як показано:

```
C1 = a1 + b1
C2 = a2 + b2
…
Cn = an + bn
```

Потокова/багатопотокова векторизація - це паралелізм рівня потоків (TLP), де потоки створюються та виконуються незалежно. Багатопотоковість завантажує ЦП завдяки залученню кількох потоків. Одночасна багатопотокова обробка або Intel Hyper Threading: заснована на суперскалярних технологіях ЦП. Інструкції з кількох потоків можуть виконуватися на будь-якій стадії конвеєра одночасно. Практичні обмеження на складність мікросхеми обмежили одночасні потоки двома.

Тимчасова багатопотокова обробка: на відміну від одночасної

багатопоточності, тимчасова багатопотокова обробка може мати лише один потік на будь-якій стадії конвеєра в певному циклі. У Course Grained Multi-Processing конвеєр основного процесора містить лише один потік за раз, і процесор повинен ефективно виконувати швидке перемикання контексту (перемикання потоків) перед виконанням іншого потоку; напр. intel itanium 2. У тонкій багатофункціональній обробці конвеєр процесора може містити кілька потоків із перемиканням контексту між етапами конвеєра; напр. Sun UltraSPARC**.**

Багатоядерний паралелізм - паралелізм на рівні процесу, справжній паралелізм, виходить далеко за рамки багатопоточності та гіперпотоковості. Потоки можуть виконуватися одночасно і незалежно як окремі процесори або ядра. У таксономії Флінна мультипроцесори класифікуються як MIMD (кілька інструкцій, багато даних). У тісно зв'язаних процесори з'єднані на рівні шини та мають доступ до центральної спільної пам'яті (процесора спільної пам'яті (SMP) або рівномірного доступу до пам'яті (UMA)), або можуть брати участь в ієрархії пам'яті як з локальною, так і зі спільною пам'яттю (Не-UMA (NUMA)). У моделі UMA усі процесори рівномірно спільно використовують фізичну пам'ять. Однак кожен процесор може використовувати приватний кеш. У моделі NUMA (нерівномірний доступ до пам'яті) доступ до пам'яті є просторовим, тобто залежить від розташування пам'яті відносно процесора. Loosely Coupled використовує кілька автономних комп'ютерів із різними операційними системами, з'єднаних між собою високошвидкісною інфраструктурою зв'язку (наприклад, кластерами).

Планування циклу - паралельні компілятори можуть виконувати планування циклу, розділяючи цикл на кілька частин, які можуть працювати одночасно на кількох процесорах. У MATLAB завдання розподіляється між різними працівниками MATLAB.

Прискорене обчислення - передбачає апаратне прискорення, яке перевершує деякі функції програмного забезпечення, що працюють на центральних процесорах загального призначення. Він розроблений для інтенсивного програмного коду, що містить гарячі точки, що вимагають великої кількості алгебраїчних операцій, таких як медична візуалізація (ультразвук, комп'ютерна томографія, МРТ), моделювання в режимі реального часу та симуляція під час прийняття фінансових рішень, складні алгоритми (кодек, стиснення даних, зберігання). , і шифрування) і так далі. Intel Xeon Phi і графічні процесори (CUDA) — це популярні варіанти апаратного прискорення, які революціонізують HPC, забезпечуючи велику кількість процесорних ядер і внутрішньої пам'яті, що працюють у поєднанні з ЦП. Пристрої Xeon Phi і GPU — це карти, які взаємодіють із ЦП через шину PCI-express. Графічні процесори NVIDIA розроблені для вирішення проблем, реалізованих як модель Single-Instruction-Multiple-Threads (SIMT). На відміну від Intel Xeon Phi, яка є менш спеціалізованою архітектурою для паралельних програмістів, знайомих із Pthreads, OpenMP, MPI, CUDA та OpenCL, GPU є спеціалізованою архітектурою з крутішою кривою навчання.

Не всі коди найкраще підходять для апаратних прискорювачів. Він може бути кандидатом на значне підвищення продуктивності, якщо застосовується така умова:

- Код є масово паралельним, тобто завдання можна розділити на багато незалежних завдань
- Код використовує складний алгоритм, що складається з великої кількості алгебраїчних операцій

Бувають випадки, коли специфічні для прискорювача програми/директиви, включені в код, можуть інтерпретуватися компіляторами (OpenACC, OpenMP, PGI). Також доступні бібліотеки з підтримкою прискорювача, такі як CULA. Крім того, деякі програми, такі

як MATLAB, Jacket, GROMACS тощо, є програмами, які працюють із прискорювачем.

Техніки оптимізації коду:

1. Усунення мертвого коду або видалення мертвого коду
2. Видалення загального підвиразу.
3. Рух інваріантного коду або підйом інваріанта з циклів
4. Loop Unwinding or Unrolling
5. Видалення стрибків/усунення гілок
6. Створення константних значень
7. Inlining
8. Зміна порядку повторення або завершення циклу
9. Усунення змінної індукції
10. Розподіл петлі/розподіл
11. Злияння/об’єднення циклів
12. Попередня вибірка
13. Оптимізація банку пам'яті
14. Комбінація інструкцій
15. Оптимізація умов
16. Інваріантні гілки циклу

1. Усунення мертвого коду або видалення мертвого коду;

   Мертвий код – це недосяжний код, який ніколи не може бути виконаний і який впливає лише на мертву змінну. Розглянемо наведений нижче фрагмент коду:

```
int dead_code () {
    int alive_var = 32;
    int dead_var = 10;
    int shift_val;
    shift_val = alive_var << 2;
    return shift_val;
    dead_var = 32;
return 0;
```

Тут локальна змінна "dead_var" є мертвою, оскільки вона не використовувалася всередині функції "dead_code()" і ніколи не використовуватиметься. Крім того, змінна "dead_var" повторно призначається після повернення без if/else, тому вона не буде виконана, відображаючи її як мертвий код. Компілятори можуть розпізнати мертвий код, відновити його місце для зберігання та усунути його ініціалізацію.

2. Видалення загального підвиразу;

   Йдеться про повторне використання значення виразу в іншому виразі, щоб уникнути повторного обчислення, якщо воно не зберігається в пам’яті чи не завантажується з пам’яті. Вирази:

```
X = sqrt(A) * cos(a);
Y = sqrt(A) * sin(b);
```

   можна переписати так:

```
B = sqrt(A);
X = B*cos(a);
Y = B*sin(a);
```

3. Рух інваріантного коду або підйом інваріанта з циклів;

   Інваріантний код циклу складається з виразів, які не залежать від змінних циклу і тому можуть бути переміщені за межі циклу для виконання лише один раз. Розглянемо наведений нижче фрагмент коду:

```
for (int i = 0; i < n; i++) {
    x = y + z;
    a[i] = k * i + x * x;
}
```

   Тут інструкція «x = y+z» виконується на кожній ітерації, хоча це не обов’язково. Його можна безпечно вийняти з циклу, як показано нижче:

```
x = y + z;
x_2 = x * x; //Expression Elimination
    for (int i = 0; i < n; i++) {
    a[i] = k* i + x_2;
}
```

4. Loop Unwinding or Unrolling - це перетворення циклу, яке зменшує кількість циклів, поєднуючи два або більше циклів за рахунок його двійкового (виконуваного) розміру.

   Наступний цикл перетворюється з:

```
for (int i = 0; i < 100; i++) {
    add(i);
}
```

   в:

```
for (int i = 0; i < 100; i+=5) {
    add(i);
    add(i+1);
    add(i+2);
    add(i+3);
    add(i+4);
}
```

   Тут кількість ітерацій було зменшено зі 100 до 20, що зменшує кількість переходів та умовних гілок, зменшуючи накладні витрати циклу.

5. Видалення стрибків/усунення гілок - стрибки та гілки коштують дорого, тому, усуваючи та мінімізуючи їх, можна підвищити продуктивність. Як показано, у першому фрагменті коду є дві гілки до L1 та L2:

```
goto L1
f()
L1:
goto L2
```

Замінюється лише однією гілкою на L2; Оператор "goto L1" видаляється, як показано нижче:

```
goto L2
f()
L1:
goto L2
```

6. Створення константних значень;

   Компілятор може оцінювати вирази з константними операндами під час компіляції та замінювати їх константами, що може покращити продуктивність під час виконання, зменшуючи розмір коду. Вираз у відповідь "15*4", як показано нижче:

```
F (){
    return 15 * 4;
}
```

   замінюється константою 60 (=15*4), як показано:

```
F (){
    return 60;
}
```

7. Inlining - Це ще одне місце для оптимізації, де виклик функції замінюється тілом визначення функції. Вбудовування виконується автоматично багатьма компіляторами, але може бути вказано вручну за допомогою директив компілятора за допомогою ключового слова, наприклад inline. Кожна вбудована функція працює швидше завдяки (i) усуненню гілок, що покращує локальність посилань кеша, (ii) видалення інструкцій виклику та повернення функції разом з іншим кодом прологу та епілогу, введеним у кожну функцію компілятором, та (iii) додаванням внутрішньопроцедурної опції. Але продуктивність може знизитися при використанні кількох копій функцій, які можуть збільшити розмір коду настільки, що він не міститься в кеші, що призводить

до промахів кешу. Пролог – це рядки коду на початку функції для підготовки стека та регістрів для використання всередині функції, тоді як епілог відновлює стек та реєструє попередній стан при виклику функції. Приклад:

```
fact (x){
    if (x == 1) return x;
    else return x + factorial (x-1);
}
```

Тут компілятор оцінить факт (3) як:

```
3 + fact(2)-> 3 + (2 + fact(1)) -> 3 + (2 + 1) = 6
```

Тепер давайте реалізуємо хвостову рекурсію, як показано нижче:

```
tail_fact (x,total=0){
    if (x==0) return total;
    else return tail_fact(x-1, total + x);
}
```

Компілятор видає висновок tail_fact(3) як:

```
tail_fact(3,0)->tail_fact(2, 3)->tail_fact(1,5)->tail_fact(0,6)->6
```

Тут викликаюча функція (tail_fact) повертає значення, отримане від функції, що викликається.

8. Зміна порядку повторення або завершення циклу - Ітерація від малого до великого числа вимагає порівняння в кінці кожної ітерації циклу. Нульові інструкції розгалуження, цикл закінчується нулем, обчислювально швидші з усуненням порівняння. Розглянемо факторіальні функції за допомогою інкрементного

циклу, як показано:

```
int fact_incr(int n){
    int fact = 1;
    for (int i = 1; i <= n; i++)
        fact *= i;
        return (fact);
}
```

Його можна реалізувати як декрементований цикл, як показано:

```
int fact_decr(int n){
    unsigned int i, fact = 1;
    for (i = n; i != 0; i--)
        fact *= i;
        return (fact);
}
```

Коли порівнюється скомпільований код, збірка для інкрементованого циклу складається з ADD & CMP, які замінюються на SUBS у декрементованому циклі, оскільки порівняння з нулем оптимізовано. Крім того, змінну «n» не потрібно зберігати (не потрібно з нею порівнювати) у циклі під час декременту. Також краще використовувати лічильник типу unsigned int, щоб зменшити накладні витрати на число зі знаком.

9. Усунення змінної індукції;

   Індукційні змінні – це ті, які збільшуються або зменшуються на фіксовану величину на кожній ітерації циклу, або є лінійною функцією іншої індукційної змінної. Компілятор може оптимізувати код, замінивши їх більш простими обчисленнями. Обчислення множення індукційних змінних i та k у наведеному нижче коді:

```
for (i=0; i < 10; ++i) {
    k = 501 * i;
}
```

можна замінити набагато простішою операцією додавання, як показано:

```
k = -501;
for (i = 0; i < 10; ++i) {
    k = k + 501;
}
```

10. Розподіл петлі/розподіл - це оптимізація компілятора, при якій цикл розбивається на кілька циклів в одному діапазоні індексів, щоб задіяти кілька процесорів у кількох завданнях. Він використовує місцевість посилання. Обчислювальні завдання для призначення масиву a і b розподіляються з:

```
for (int i = 0; i < 100; i++) {
    a[i] = 0;
    b[i] = 0;
}
```

в:

```
for (int i = 0; i < 100; i++) {
    a[i] = 0;
}
for (int i = 0; i < 100; i++) {
    b[i] = 0;
}
```

11. Злияння/об’єднення циклів - це протилежність Loop Fission і застосована в архітектурі, де локальність даних у кожному циклі не є значною і не відбувається багаторазова обробка.
12. Попередня вибірка - це механізм, який передає дані, які можуть знадобитися процесору в найближчому майбутньому. Hardware Prefetch не передбачає підтримки компілятора. Процесор Pentium IV має автоматичну апаратну попередню вибірку. Отже, немає потреби використовувати програмну попередню вибірку, оскільки апаратна попередня вибірка набагато ефективніша. Програмне

забезпечення Prefetch працює в парі з апаратним забезпеченням. Коли пропускна здатність шини доступна, процес може розпочати завантаження пам’яті в кеш до того, як це знадобиться. Залежно від алгоритму можна використовувати різні інструкції попередньої вибірки. Якщо ви хочете попередньо вибрати дані для 16 ітерацій циклу в майбутньому в Pentium 4, фрагмент коду виглядає так:

```
for (i=0; i<1000; i++) {
    X = fn(array[i]);
    _mm_prefetch(array[i+16], MM_HINT_T0);
}
```

13. Оптимізація банку пам'яті;

Суперечка за пам’яттю виникає внаслідок послідовних звернень до одного і того ж банку пам’яті, оскільки кожен доступ до банку даних призводить до того, що банк стає недоступним протягом деякої кількості тактових періодів. Кожен банк також має обмежену пропускну здатність, що спричиняє значне сповільнення, якщо елементи даних, до яких потрібно отримати доступ, знаходяться в одному банку. Кількість банків майже завжди є степенем двійки.

Збільшуючи розмір масиву більше ніж двійку (заповнення масиву) і через обмін циклами, можна запобігти сповільненню банків пам’яті. Існує також апаратна техніка, яка вирішує проблему шляхом змішування даних у банках за допомогою методу під назвою «теорія китайського нагадування».

Розглянемо фрагмент коду нижче:

```
float a[100][128], b[100][128]; //Array size: 128 = 2^6 i.e. power of 2
for (i=0;i<128;i++) /* Avoid this order */
    for (j=0;j<100;j++)
        a[j][i] = b[j][i] * 10;
```

"цикли for" для і та j міняються місцями нижче:

```
float a[100][128], b[100][128];
    for (j=0;j<100;j++) //loop Iiand j are interchanged
        for (i=0;i<128;i++)
            a[j][i] = b[j][i] * 10;
```

Розмір масиву збільшується на 1 (тобто 129 = 2^7 + 1), щоб значення більше не було ступенем 2, як показано нижче:

```
float a[100][129], b[100][129];//2^7 is increased by 1
for (i=0;i<128;i++)
    for (j=0;j<100;j++)
        a[j][i] = b[j][i] * 10;
```

14. Комбінація інструкцій - це процес оптимізації, який поєднує дві або більше інструкцій в одну. Розглянемо наступний фрагмент коду, де є дві інструкції x++.

```
int x;
increment(){
    x++;
    x++;
}
```

Їх можна об’єднати в одне, як показано нижче:

```
Int x;
increment(){
    x+=2;
}
```

15. Оптимізація умов - фрагментовані оператори IF, як показано нижче:

```
F(int value){
    if (value) f1();
    if (value) f2()
}
```

можна об’єднати в один оператор IF:

```
F(int value){
    if (value)
        f1();
        f2();
}
```

16. Інваріантні гілки циклу;

Цикл можна оптимізувати, якщо можна уникнути розгалужень всередині циклів, що допомагає компілятору використовувати SIMD. Якщо гілка в коді нижче:

```
for (int i=0; i<size; i++){
    if (a = value1)
    …
    else if ..
    else ..
}
```

можна безпечно вийняти з петлі, як показано:

```
if (a = value1)
    for (int i=0; i<size; i++)
    …
else if …
    for (int i=0; i<size; i++)
else …
    for (int i=0; i<size; i++)
```

Хоч більшість з цих методів використовують саме компілятори, про них необхідно пам’ятати при написанні програмного коду.

1.2. Оптимізація на стороні клієнта та на стороні сервера

Програми являють собою суміш серверного та клієнтського коду. Програма може мати проблеми з продуктивністю з обох боків, і обидві потребують оптимізації.

Клієнтська сторона стосується того, як продуктивність відображається у веб-браузері чи інтерфейсі користувача. Це включає час

початкового завантаження сторінки, завантаження всіх ресурсів, JavaScript, який працює в браузері, час завантаження зображень тощо.

Серверна сторона стосується того, скільки часу потрібно для роботи на сервері для виконання запитів. Оптимізація продуктивності на сервері зазвичай полягає в оптимізації таких речей, як запити до бази даних та інші залежності додатків.

1.2.1. Оптимізація продуктивності на стороні клієнта

Кешування та мережі доставки вмісту:

- Мережі доставки вмісту — це розумний спосіб обробки статичних файлів, таких як JavaScript, CSS і файли зображень, які не змінюються.
- Приклади cdns; cloudflare, Amazon AWS, MaxCDN, Fastly тощо.

Набір і мінімізація:

- Об’єднання файлів разом і створення меншої кількості файлів підвищує продуктивність
- Зменшення файлів і видалення всіх непотрібних символів, наприклад пробілів, також покращує продуктивність.

Оптимізація використання зображення:

- Більшість зображень можна оптимізувати та зменшити
- Крім того, якщо ви використовуєте кілька маленьких значків, краще використовувати лише шрифти значків, наприклад. чудовий шрифт.

Видалення повторюваного коду JavaScript і CSS:

- Видалення повторюваного коду зменшує розмір файлів, отже, покращує продуктивність.

Використання мінімалістичного Framework’у зі стилями:

- Це допомагає з аспектом стилю, і оскільки структуру стилю вже оптимізовано та мінімізовано для кращої

продуктивності, це має бути ефективним для підтримки продуктивності вашої системи на оптимальному рівні.

1.3. Ризик передчасної оптимізації

Передчасна оптимізація — це витрачати багато часу на те, що вам насправді може не знадобитися.

Урок, який ми, інженери програмного забезпечення, засвоюємо на початку своєї кар'єри, полягає в тому, що «передчасна оптимізація — це корінь усього зла». Цитата Дональда Кнута: «Справжня проблема полягає в тому, що програмісти витратили занадто багато часу, турбуючись про ефективність у невідповідних місцях і не в той час; передчасна оптимізація є корінням усіх зол (або принаймні більшої частини) у програмуванні». Ця перлина поради від Кнута спрямована на надмірне бажання новачків — а іноді й не дуже — розробників програмного забезпечення оптимізувати ефективність свого коду. Натомість першим кроком має бути надсилання коду, який працює, а потім з'ясування того, які критичні компоненти, якщо такі є, потрібно оптимізувати для продуктивності.

Сьогодні ми все частіше стикаємося з можливостями передчасної оптимізації в контексті моделей машинного навчання, де продуктивність стосується точності, а не ефективності. Гнучка розробка: це не лише розробка програмного забезпечення. Наріжним каменем сучасної інженерії програмного забезпечення є гнучка розробка, яка говорить нам, що ми повинні спочатку створити мінімально життєздатний продукт (MVP), а потім поступово повторювати його. У цьому контексті зрозуміло, чому вам слід уникати передчасної оптимізації продуктивності: це порушує визначення MVP. Ви повинні спочатку створити щось, що працює, а потім поступово вдосконалювати це. Хороша розробка машинного навчання дотримується подібного принципу: кожен проект машинного навчання має починатися з MVP.

Продуктивність — не єдина мета моделей машинного навчання. Більшість команд розробників сьогодні звикли до безперервної доставки коду та швидкої ітерації. Більшість усіх команд використовують гнучкі методології. Ми знаємо, що якщо в нашому програмному забезпеченні є помилка, ми можемо досить легко розгорнути виправлення на нашому веб-сервері.

Думка про передчасну оптимізацію все ще актуальна, і ця концепція стосується сучасного розвитку. Передчасна оптимізація – це те, про що розробникам слід завжди думати. Це те, чого вони завжди повинні намагатися уникати у своїй щоденній роботі. Сьогодні це стосується так само, як і за часів мейнфреймів і перфокарт.

Класичним прикладом цього є стартап, який витрачає величезну кількість часу, намагаючись з’ясувати, як масштабувати своє програмне забезпечення для роботи з мільйонами користувачів. Це дуже серйозне занепокоєння, про яке варто подумати, але не обов’язково діяти. Перш ніж турбуватися про роботу з мільйонами користувачів, вам потрібно переконатися, що 100 користувачам навіть подобається ваш продукт і вони хочуть використовувати його. Перевірка відгуків користувачів має бути на першому місці.

Чим більше у вас впевненості в тому, що ви створюєте правильні речі, тим більше часу вам слід приділити правильній архітектурі програмного забезпечення, продуктивності, масштабованості тощо. Встановлення цього балансу завжди є складним завданням. Як і більшість речей у житті, відповідь майже завжди така: «це залежить».

Продуктивність і масштабованість вашої програми важливі. Вам просто потрібно спочатку переконатися, що ви створюєте правильний набір функцій. Уникайте передчасної оптимізації, завчасно й часто отримуючи відгуки від користувачів. Ви можете подумати, що з моєю аналогією є проблема: ефективність у розробці програмного забезпечення є другорядною метою (після правильності), тоді як продуктивність моделі

в машинному навчанні є основною метою. Безсумнівно, надзвичайно неточна модель машинного навчання не є особливо корисною.

Але коли модель досягає мінімально прийнятної точності, важливо враховувати інші фактори. Дійсно, незначні покращення продуктивності моделі часто є вторинними щодо інших факторів. Зокрема, система машинного навчання повинна розглядати «трьох колишніх»:

- Вираження корисності та вхідного розподілу
- Пояснення результатів
- Експериментування

Якщо цільова функція не справляється з утилітою адекватного моделювання, ви оптимізуєтеся для неправильної речі. Якщо вхідний розподіл спотворений систематичним зміщенням, введення сміття забезпечує виведення сміття. У будь-якому випадку немає сенсу оптимізувати продуктивність отриманої моделі. У той же час пояснюваність має вирішальне значення для налагодження моделей і виявлення можливостей їх покращення. Нарешті, вартість експериментів визначає, наскільки швидко ви зможете покращити моделі.

РОЗДІЛ 2
ОПТИМІЗАЦІЯ ГРАФІЧНИХ ЕЛЕМЕНТІВ ІНТЕРФЕЙСУ КОРИСТУВАЧА, ВИКОНУВАНОГО КОДУ ТА ВИКОРИСТАННЯ РЕСУРСІВ ВЕБ-ДОДАТКУ КСУ24

1.1. Оптимізація графічних елементів додатку

Веб-сайт складається з ряду елементів, таких як текст, кольори, зображення, ілюстрації тощо. Одним із цих елементів є піктограми, які використовуються на різних веб-сайтах у різних стилях. Піктограми (від грецького «eikōn», що означає «зображення») є одними з найбільш використовуваних графічних елементів у нашу цифрову епоху, але походять із найдавніших часів. Ікони служили ключовими точками різноманітної інформації впродовж історії, і існують численні дослідження важливості іконопису в різних людських цивілізаціях.

Оптимізація піктограм у сучасних багатофункціональних програмах є обов'язковою. Піктограми різних форматів і розмірів часто переводять користувачів у нудні екранні завантажування. Такий досвід може збентежити й розчарувати користувача ще до того, як він побачить основну програму, що призведе до поганої взаємодії з користувачем.

Основна місія піктограми - надавати, не відволікаючи, конкретну візуальну інформацію. Щоб успішно виконати цю місію, значок повинен мати ряд характеристик, серед яких виділяються наступні:

- Піктограми мають бути максимально простими, але з достатньою кількістю деталей, щоб висловити передбачуване повідомлення. Якщо піктограми надто складні та/або мають забагато слідів, вони можуть заплутати та потенційно передавати неправильне повідомлення.
- Важливо пам'ятати про різні моделі значків, які вже широко прийняті загалом. Наприклад, якщо під час заповнення форми з'являється повідомлення про помилку, значок сповіщення,

який супроводжує текст, допоможе нам зрозуміти, що це важлива помилка чи попередження. Якби ми використали символ серця, це б повністю заплутало користувача. Зміна цієї моделі, до якої користувачі звикли, може призвести до втрати чіткості значка та призвести до збою зв'язку та інформації.

- Піктограми мають бути ідеально розроблені для конкретної інформації, яку вони представлятимуть.

Чому іконки так важливі?

- Значки описують функції.

  Хоча використання слів для пояснення понять є найбільш прямим підходом, користувачі можуть занудьгувати або втратити увагу через занадто багато тексту та взагалі пропустити повідомлення. З цієї причини додавання піктограм для опису цих функцій і особливостей може бути ще ефективнішим підходом. Вони не тільки додають більше різноманітності до розділу, але й підсилюють повідомлення візуальним поясненням. Це значно полегшує для користувача пов'язування піктограми з концепцією та розуміння описаних особливостей або функцій.

- Значки пояснюють історію.

  Піктограми часто використовуються під час пояснення процесу, оскільки ви можете візуально представити кожен крок. Вони допомагають нам візуально зрозуміти кожен крок процесу та полегшують візуальне розділення та розуміння елементів, які вони містять.

- Значки допомагають заповнювати форми.

  Веб-форма є одним із найкращих способів залучити потенційних клієнтів і побічно встановити з ними теплі стосунки. Час, який ви витрачаєте на те, щоб залучити

користувача на свій веб-сайт, має бути доповнений такою ж кількістю часу на вдосконалення взаємодії з користувачем, включаючи ваші веб-форми. Текстовий опис даних, необхідних для кожного поля, часто супроводжує значок; однак у певних формах можна знайти пов'язані значки, зрозумілі без використання слів. Основна мета кожного дизайнера полягає в тому, щоб веб-форму можна було заповнити швидко та легко. Додавання типу даних і відповідного значка покращує швидке розуміння.

- Значки допомагають оптимізувати навігацію.

  Піктограми допомагають пояснити поняття візуально. Їх також можна використовувати як відмінні елементи під час навігації веб-сайтом, наприклад, полегшуючи пошук певної інформації на сайті та її відстеження. Наприклад, припустімо, що у вашій компанії є 5 різних послуг, і ці служби перераховані на домашній сторінці з окремим значком для кожної з них. Після натискання на один із значків сервісу користувачеві буде набагато простіше зрозуміти, до якого сервісу він занурюється, якщо ця нова сторінка матиме той самий логотип сервісу в розділі героя. З точки зору перспективи, піктограми можуть надати певного стилю сайту.

Існує багато стилів іконок. Деякі значки використовують тіні та рельєфи, площини, лінійні, 3D-ефекти тощо. Використовуючи певний стиль у ваших значках, індивідуальність додається до загального дизайну. З цієї причини важливо використовувати піктограми, які візуально відповідають естетиці вашого веб-сайту для більш корпоративного дизайну. Створення піктограм є складним процесом — не тому, що вони вимагають багато роботи, а тому, що дуже важко відобразити точне повідомлення, яке ви хочете висловити простою фігурою. Значки мають величезний вплив на веб-сайт, оскільки кожен елемент, який ви

розміщуєте на своєму сайті, може вплинути на загальну взаємодію з користувачем. Треба завжди пам’ятати про відповідність значків, які використовуються на вашому сайті, оскільки це може допомогти уникнути їх неефективного використання та помилкового додавання їх до неправильних розділів сайту.

Проблеми для вирішення:

- Розмір файлу піктограм – значки великого розміру можуть з’явитися на екрані довше.
- Масштабування піктограми – та сама піктограма в різних розмірах розглядається як нова піктограма.
- Формати значків – керування різними форматами значків в одній програмі.
- Не вдалося завантажити – з будь-якої причини значок не завантажується.

Можливі рішення:

- SVG (масштабована векторна графіка)
- Шрифт значків – працює з платформами Web, Android та iOS

Оскільки мова йде про веб-розробку, то бажаніше використовувати векторну графіку, оскільки вона меньше важить, у порівнянні з расторовою, а отже і швидше завантажується а інші. Розглянемо їх більш етально.

Плюси SVG: він дуже стисливий і легкий; якість зображення залишається високою незалежно від ступеня стиснення. Він легко масштабується, тому добре виглядає на екрані незалежно від розміру зображення(див. рис. 2.1.1).

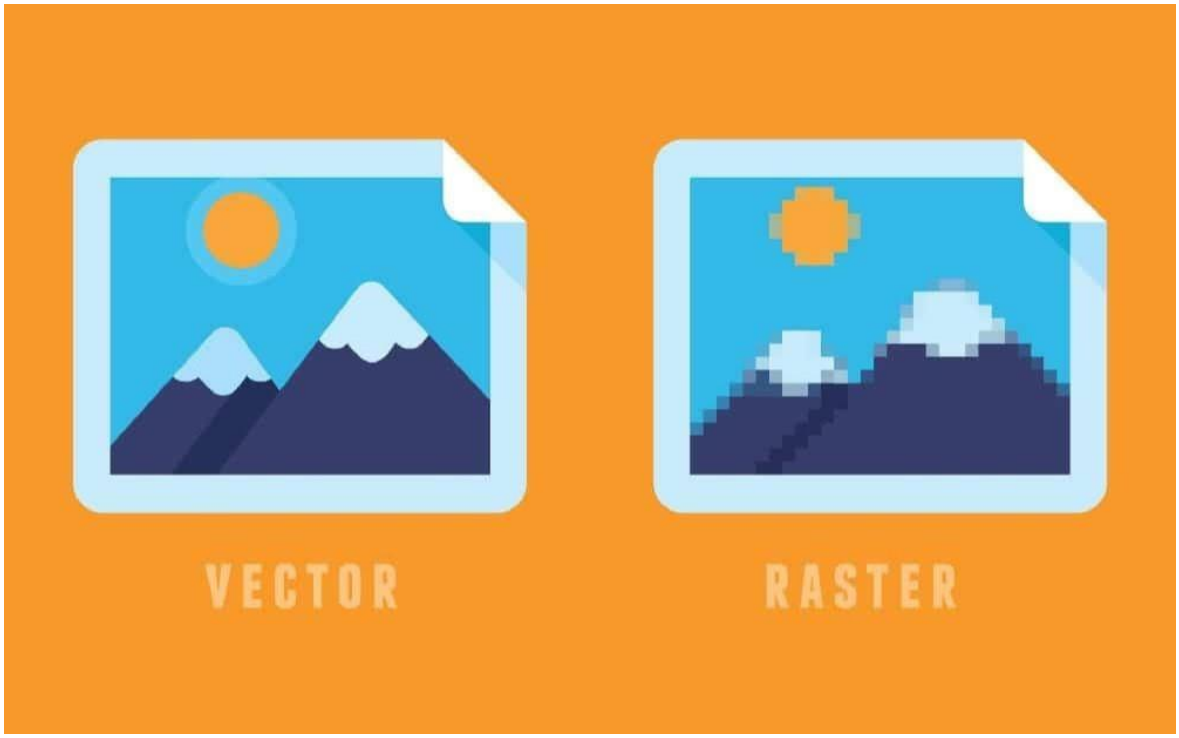


Рис. 2.1.1. Втрата якості растрового зображення підчас масштабування

- Він підтримується всіма сучасними браузерами (Chrome, Opera, Firefox, Safari, навіть Internet Explorer).
- Дизайнери можуть створювати прості візуалізації SVG у редакторі коду чи тексту, а також експортувати складну графіку за допомогою Adobe Illustrator або Sketch.
- Текст, вставлений як окремий прошарок, полегшує пошук зображення SVG. Формат SVG дозволяє дизайнерам зберігати статичні чи анімовані зображення.
- Ці зображення легко редагувати, змінюючи параметри в коді CSS.
- Ви можете зберігати об'єкт багато разів під час роботи з ним без шкоди для початкової якості.

Однак SVG не є ідеальним вибором для кожного випадку через наступні недоліки:

- У користувачів все ще можуть виникати проблеми з кросбраузерною сумісністю, особливо коли мова йде про поштові клієнти та старіші версії програмного забезпечення.
- Він не підходить для фотографій, оскільки векторна графіка спочатку була призначена для роботи з фігурами, лініями тощо.
- Існують труднощі з використанням SVG у картографічних програмах, де для відображення невеликого фрагмента зображення потрібно прочитати весь файл.

1.2. Оптимізація виконуваного коду та використання ресурсів

Внутрішньо React використовує кілька розумних прийомів, щоб мінімізувати кількість дорогих операцій DOM, необхідних для оновлення інтерфейсу користувача. Хоча це призведе до швидшого інтерфейсу користувача без спеціальної оптимізації продуктивності для багатьох випадків, є способи, за допомогою яких ви можете пришвидшити свою програму React.

Основний принцип, закладений у React - це те, що додаток складається з окремих компонентів, які можна замінити або оновити самостійно, не оновлюючи всю сторінку. Це економить пропускну здатність. Зовнішні файли не завантажуються кожного разу при повторному завантаженні сторінки (SPA)[8] (див. Рис.2.1.2.).

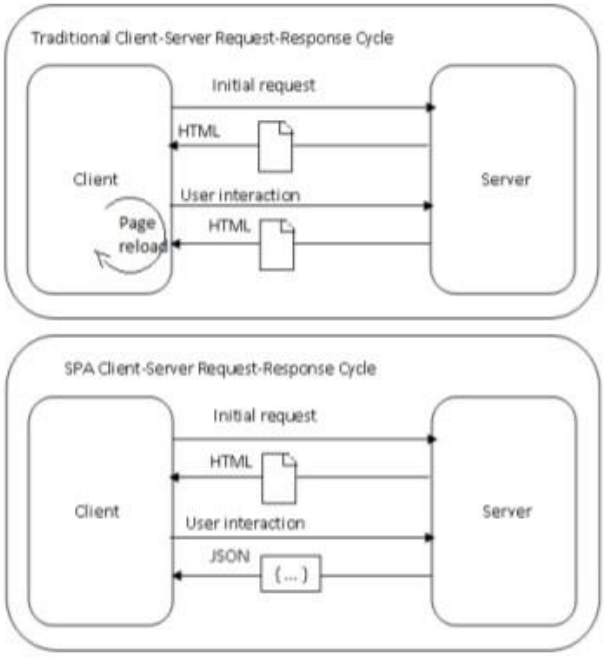


Fig. 1 Client-Server Request-Response Cycle

Рис. 2.1.2. Приклад «Цикл запит-відповідь SPA »

Крім того React використовує JSX [9] та вбудований стиль для створення та стилювання компонентів

без ускладнення структури або коду. У React стиль визначається як об'єкт і додається до елемента всередині тегу HTML(див. рис. 2.1.3.).

```
var divStyle = {color: blue};

React.render(
  <h1 style={divStyle}>Hello, world!</h1>,
  document.getElementById('myDiv')
);
```

Рис. 2.1.3. Приклад додавання стилю до блоку розмітки

інтерфейсу, а також підтримку для створення власних власних модулів та подань, якщо це потрібно. Розглянемо більш детально основні прицнипи, що були закладені в React.

Компоненти Function/Stateless і React.PureComponent

У React функціональні компоненти та PureComponent пропонують два різні способи оптимізації програм React на рівні компонентів. Функціональні компоненти запобігають створенню екземплярів класу, одночасно зменшуючи загальний розмір комплекту, оскільки він мінімізується краще, ніж класи. З іншого боку, щоб оптимізувати оновлення інтерфейсу користувача, ми можемо розглянути можливість перетворення функціональних компонентів у клас PureComponent (або клас із спеціальним методом shouldComponentUpdate). Однак, якщо компонент не використовує стан та інші методи життєвого циклу, початковий час візуалізації дещо складніший порівняно з функціональними компонентами з потенційно швидшими оновленнями.

React.PureComponent робить неглибоке порівняння зміни стану. Це означає, що він порівнює значення при перегляді простих типів даних і порівнює посилання на об’єкти. Через це ми повинні переконатися, що під час використання React.PureComponent виконуються два критерії:

- Component State/Props є незмінним об’єктом;
- State/Props не повинні мати багаторівневий вкладений об’єкт.

Усі дочірні компоненти React.PureComponent також мають бути чистими або функціональними компонентами.

Кілька фрагментів файлів

Ваша програма завжди починається з кількох компонентів. Ви починаєте додавати нові функції та залежності, і, перш ніж ви це усвідомлюєте, ви отримуєте величезний робочий файл. Ви можете мати два окремих файли, відокремивши код бібліотеки постачальника або сторонньої бібліотеки від коду програми, скориставшись перевагами webpack. Ви отримаєте vendor.bundle.js і app.bundle.js. Завдяки розділенню файлів ваш браузер рідше кешує та паралельно завантажує ресурси, щоб зменшити час очікування завантаження.

Оптимізація залежностей

Розглядаючи оптимізацію розміру набору додатків, варто перевірити, скільки коду ви фактично використовуєте із залежностей. Використовуйте React.Fragments, щоб уникнути додаткових обгорток елементів HTML. React.fragments дозволяє групувати список дочірніх елементів без додавання додаткового вузла:

```
class Comments extends React.PureComponent{
    render() {
        return (
            <React.Fragment>
                <h1>Comment Title</h1>
                <p>comments</p>
                <p>comment time</p>
            </React.Fragment>
        );
    }
}
```

Існує альтернативний і більш стислий синтаксис з використанням React.fragments:

```
class Comments extends React.PureComponent{
    render() {
        return (
            <>
                <h1>Comment Title</h1>
                <p>comments</p>
                <p>comment time</p>
            </>
        );
    }
}
```

Уникайте вбудованого визначення функції у функції відтворення.

Оскільки функції є об’єктами в JavaScript, вбудована функція завжди не виконуватиме prop diff, коли React виконує перевірку diff. Крім того, функція зі стрілкою створить новий екземпляр функції для кожного рендера, якщо вона використовується у властивості JSX. Це може створити багато роботи для збирача сміття:

```
default class CommentList extends React.Component {
    state = {
        comments: [],
        selectedCommentId: null
    }

    render(){
        const { comments } = this.state;
        return (
           comments.map((comment)=>{
               return <Comment onClick={(e)=>{
                    this.setState({selectedCommentId:comment.commentId})
               }} comment={comment} key={comment.id}/>
           })
        )
    }
}
```

Замість визначення вбудованої функції для props, ви можете визначити функцію стрілки:

```
default class CommentList extends React.Component {
    state = {
        comments: [],
        selectedCommentId: null
    }

    onCommentClick = (commentId)=>{
        this.setState({selectedCommentId:commentId})
    }

    render(){
        const { comments } = this.state;
        return (
           comments.map((comment)=>{
               return <Comment onClick={this.onCommentClick}
                comment={comment} key={comment.id}/>
           })
        )
    }
}
```

Регулювання та усунення стрибків подій у JavaScript

Частота запуску події — це кількість викликів обробника події за певний проміжок часу. Загалом клацання мишею має нижчу частоту спрацьовування подій порівняно з прокручуванням і наведенням миші. Вищі частоти запуску подій іноді можуть призвести до збою вашої програми, але це можна контролювати.

Давайте обговоримо деякі техніки.

По-перше, визначте обробник подій, який виконує дорогу роботу. Наприклад, запит XHR або маніпуляція DOM, яка виконує оновлення інтерфейсу користувача, обробляє велику кількість даних або виконує дорогі обчислювальні завдання. У цих випадках методи дроселювання та усунення стрибків можуть стати порятунком, не вносячи жодних змін у слухача подій.

Дроселювання

Дроселювання означає затримку виконання функції. Таким чином, замість негайного виконання обробника/функції події, ви додасте кілька мілісекунд затримки, коли подія запускається. Це можна використовувати, наприклад, при реалізації нескінченної прокрутки. Замість отримання наступного набору результатів, коли користувач прокручує, ви можете відкласти виклик XHR. Іншим хорошим прикладом цього є миттєвий пошук на основі Ajax. Можливо, ви не захочете звертатися до сервера при кожному натисканні клавіші, тому краще гальмувати, доки поле введення не перестане працювати на кілька мілісекунд. Дроселювання можна реалізувати кількома способами. Ви можете регулювати кількість запущених подій або обробник подій затримки, що виконується.

Усунення стрибків

На відміну від дроселювання, усунення стрибків — це техніка, яка запобігає надто частому запуску тригера події. Ось демонстраційний код для пошуку коментарів:

```
import debouce from 'lodash.debounce';

class SearchComments extends React.Component {
 constructor(props) {
   super(props);
   this.state = { searchQuery: “” };
 }

 setSearchQuery = debounce(e => {
   this.setState({ searchQuery: e.target.value });

   // Fire API call or Comments manipulation on client end side
 }, 1000);

 render() {
   return (
     <div>
       <h1>Search Comments</h1>
       <input type="text" onChange={this.setSearchQuery} />
     </div>
   );
 }
}
```

Уникайте використання індексу як ключа для елементу маперу

Досить часто індекси використовуються як ключі під час візуалізації списку:

```
{
    comments.map((comment, index) => {
        <Comment
            {..comment}
            key={index} />
    })
}
```

Але використання ключа як індексу може показати вашу програму неправильними даними, оскільки вони використовуються для ідентифікації елементів DOM. Коли ви натискаєте або видаляєте елемент зі списку, якщо ключ той самий, що й раніше, React припускає, що елемент DOM представляє той самий компонент. Завжди доцільно використовувати унікальну властивість як ключ, або, якщо ваші дані не мають унікальних атрибутів, ви можете подумати про використання модуля uuid, який генерує унікальний ключ. Однак якщо дані мають унікальну властивість, наприклад ідентифікатор, краще використовувати цю властивість:

```
{
    comments.map((comment, index) => {
        <Comment
            {..comment}
            key={comment.id} />
    })
}
```

У певних випадках цілком допустимо використовувати індекс як ключ, але лише якщо виконується наведена нижче умова:

- Список і елементи статичні.

- Елементи в списку не мають ідентифікаторів, і список ніколи не буде змінено або відфільтровано.
- Список незмінний.

Використовуйте Reselect у Redux, щоб уникнути частого повторного відтворення

Reselect — це проста бібліотека селекторів для Redux, яку можна використовувати для створення запам’ятованих селекторів. Ви можете визначити селектори як функції, що отримують фрагменти стану Redux для компонентів React. Давайте подивимося на цей код, який не використовує Reselect:

```
import { createSelector } from "reselect";
const socialPathSelector = state => state.social;
const addStaticPath = createSelector(
  socialPathSelector,
  social => ({
    iconPath: `../../image/${social.iconPath}`
  })
);
```

Memoize React Components

Запам'ятовування — це техніка оптимізації, яка використовується головним чином для прискорення роботи комп'ютерних програм шляхом зберігання результатів дорогих викликів функцій і повернення кешованого результату, коли ті самі введення повторюються знову. Запам’ятана функція зазвичай працює швидше, тому що якщо функція викликається з тими самими значеннями, що й попередня, тоді замість виконання логіки функції вона отримуватиме результат із кешу. Цей метод виконає неглибоке рівне порівняння як атрибутів, так і контексту компонента на основі суворої рівності. Приклад використання для всього компоненту:

```
const UserDetails = ({user, onEdit}) =>{
    const {title, full_name, profile_img} = user;

    return (
        <div className="user-detail-wrapper">
            <img src={profile_img} />
            <h4>{full_name}</h4>
            <p>{title}</p>
        </div>
    )
}

export default React.memo(UserDetails)
```

Для окремої функції:

```
function App() {
    const [count, setCount] = useState(0)

    const expFunc = (count)=> {
        waitSync(3000);
        return count * 90;
    }

    const resCount = useMemo(()=> {
        return expFunc(count)
    }, [count])

    return (
        <>
            Count: {resCount}
            <input type="text" onChange={(e)=>
setCount(e.target.value)} placeholder="Set Count" />
        </>
    )
}
```

Аналіз та оптимізація вашого пакету Webpack Bundle

Перед розгортанням робочої версії вам слід перевірити й проаналізувати комплект програм, щоб видалити непотрібні плагіни чи модулі. Ви можете розглянути можливість використання Webpack Bundle Analyzer, який дозволяє візуалізувати розмір вихідних файлів webpack за допомогою інтерактивної масштабованої дерева карти. Який і був використаний в рамках цієї роботи. Тут ви можете бачити як саме ця

бібліотека візуалізує виконуваний код, що допомагаю у визначенні місць, які потребуют усунення або оптимізації коду (див. Додаток В).

Увімкніть стиснення Gzip на веб-сервері

Стиснення Gzip дозволяє веб-серверу забезпечувати менший розмір файлу, що означає, що ваш веб-сайт завантажується швидше. Gzip працює так добре тому, що у файлах JavaScript, CSS і HTML використовується багато повторюваного тексту з великою кількістю пробілів. Оскільки gzip стискає звичайні рядки, це може зменшити розмір сторінок і таблиць стилів до 70%, скорочуючи час першого візуалізації вашого веб-сайту.

Використання відкладеного завантаження компонентів

Групування — це процес імпортування та об’єднання кількох файлів в один файл, щоб програмі не потрібно було імпортувати багато зовнішніх файлів. Усі основні компоненти та зовнішні залежності об’єднуються в один файл і надсилаються через мережу, щоб веб-програма була запущена та запущена. Це заощаджує багато мережевих викликів, але також призводить до проблеми, коли цей єдиний файл стає великим файлом і споживає велику пропускну здатність мережі. Програма постійно чекає, поки цей великий файл буде завантажено та виконано, тому затримки в передачі цього файлу через мережу призводять до збільшення часу візуалізації для програми. Щоб вирішити цю проблему, ми можемо включити концепцію розбиття коду. Концепція поділу коду підтримується пакетами, такими як webpack, які можуть створювати кілька пакетів для програми та які можна динамічно завантажувати під час виконання. Завантаження під час виконання зменшує розмір початкового пакету, який створюється. Ми можемо планувати розбити пакети таким чином, щоб компоненти, які спочатку не завантажені в програму, можна було відкласти для завантаження пізніше, коли вони знадобляться. Це зменшить розмір основного комплекту та зменшить час завантаження програми. Для цього ми використовуємо Suspense і lazy. Таким чином, завантаження всіх компонентів, які можуть

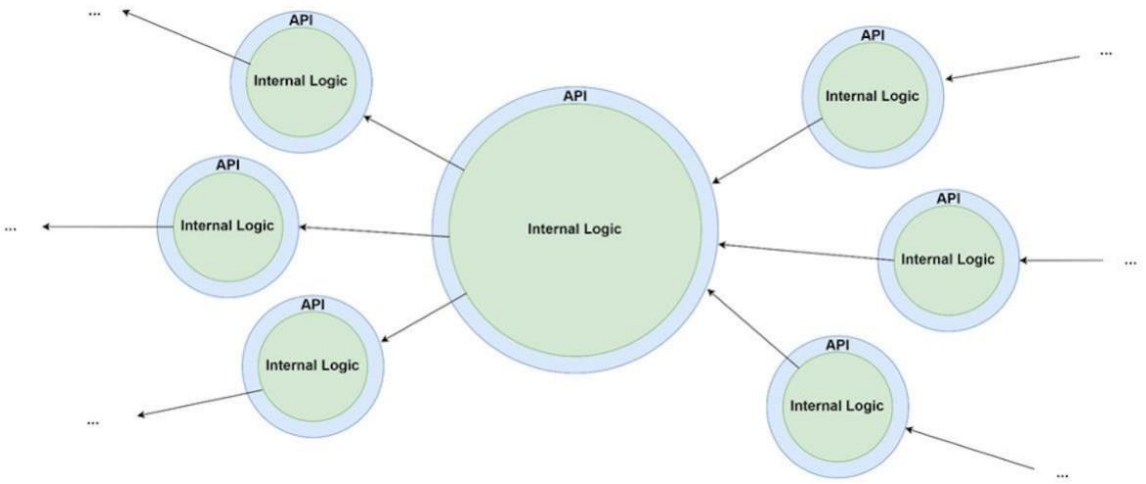
або не можуть бути додані в поданні, призведе до зниження продуктивності. За потреби ми можемо ліниво завантажувати компоненти, і ці компоненти є частиною окремого блоку, завантажуваного під час виконання, що покращує загальну продуктивність програми. Ми можемо зрозуміти це за допомогою іншого прикладу. Припустімо, що є два різні компоненти, які відображаються залежно від того, увійшов користувач чи ні. Відтворюється один із двох компонентів: WelcomeComponent або GuestComponents, залежно від того, увійшов користувач чи ні. Замість того, щоб завантажувати обидва компоненти у початковому файлі комплекту, ми можемо відкласти завантаження компонента на основі умови пізніше. Щоб початковий пакет не містив обидва ці компоненти, новий комплект буде завантажено на льоту відповідно до вказаної умови.

Переваги підходу: основний пакет зменшиться в розмірі, оскільки WelcomeCompoment і GuestComponents не завантажуються в початковий пакет, споживаючи менше часу мережі для початкового завантаження пакета. Окремий запит на завантаження необхідного компонента робиться на льоту, відповідно до вказаної умови. Окремо завантажений пакет — це невеликий пакетний файл, і його можна завантажити без затримки.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ФУНКЦІОНАЛЬНИХ МОДУЛІВ

Перш за все, перед початком роботи зі створення нового функціоналу та інтеграції його у вже існуючу систему треба розуміти з чого складається та як функціонує веб-додаток. Візуалізувавши архітектуру платформи ХДУ24 стає зрозуміло, як саме збудований додаток, з яких частин він складається та яким чином усі його елементи поєднані між собою (див. Додаток Б). Таким чином ми можемо дізнатися, що клієнтська частина додатку отримує від відсилає усю необхідну інформацію до серверної частини за допомогої API.

Термін API розшифровується як прикладний програмний інтерфейс, концепція, яка застосовується скрізь, від інструментів командного рядка до корпоративного коду, мікросервісів і хмарних архітектур. API — це інтерфейс, який розробники програмного забезпечення використовують для програмної взаємодії з компонентами або ресурсами програмного забезпечення за межами власного коду. Ще простіше визначення полягає в тому, що API — це частина програмного компонента, яка доступна для інших компонентів. Якщо ви не будете писати кожен рядок коду з нуля, ви взаємодіятимете із зовнішніми програмними компонентами, і кожен із них матиме власний API. Навіть якщо ви пишете весь свій код з нуля, добре спроектована програма повинна мати внутрішні API, щоб допомогти впорядкувати код і зробити його компоненти зручнішими для повторного використання. Візуалізувати взаємодію декількох компонентів через свої API можна наступним чином:



Також побачити, що API додатку є задокументоване за допомогою Swagger. Він дозволяє описати структуру API, щоб машини могли їх читати. Здатність API описувати власну структуру є корінням усієї приголомшливості Swagger. Прочитавши структуру вашого API, ми можемо автоматично створити красиву та інтерактивну документацію API. Ми також можемо автоматично створювати клієнтські бібліотеки для API багатьма мовами та досліджувати інші можливості, як-от автоматичне тестування. Swagger робить це, запитуючи наш API повернути YAML або JSON, що містить детальний опис усього вашого API. Ця задокументованість відкидає можливість помилкового використання існуючого коду, а також пришвидшує розробку, оскільки спрощую отримання потрібної інформації поміж команди розробників.

Оскільки сучасні SPA складаються з модулів, які в свою чергу складаються з компонентів було розроблено декілька з них, а саме:

1. Модуль опитувань (див. Додаток Д).
2. Модуль документообігу (див. Додаток Е).
3. Модуль скорочення посилань (див. Додаток Є).
4. Модуль сервісів системи (див. Додаток Ж).

Усі вони були успішно реалізовані та інтегровані у систему ХДУ24 та вже використовується користувачами.

ВИСНОВКИ

На першому етапі даної роботи було досліджено та проаналізовано основні методи та рівні оптимізації виконуваного коду, а також ризик передчасної оптимізації під час розробки програмного забезпечення. Визначено основні можливості оптимізації програмного коду, що реалізовані у фреймворку для розроки веб-додатків React. Визначено основні напрями оптимізації виконуваного коду клієнтської частини веб-додатку ХДУ24.

На другому єтапі даної роботи були розглянуті відмінності у використанні векторної та растрової графіки у розробці веб-додатків, за результатами чого було оптимізовано та уніфіковано набір піктограм платформи. Було зменшено розмір завантажуваного коду шляхом усунення зайвих залежностей клієнтської частини додатку, а також впроваджено «ліниве завантаження» частин коду, що пришвидшило завантаження коду користувачем та взаємодію в цілому.

Також у результаті роботи було реалізовано повноцінні функціональні модулі опитувань, цифрового документообігу, скорочування посилань та сервісів веб-платформи ХДУ24.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Цифровізація державних процесів [Електронний ресурс]. – Режим доступу: URL: https://plan2.diia.gov.ua/.

2. The Software Optimization Cookbook – Richard Gerber

3. Optimization [Електронний ресурс]. – Режим доступу: URL: https://suif.stanford.edu/dragonbook/lecture-notes/Stanford-CS143/20-Optimization.pdf

4. Нотація Big-O [Електронний ресурс]. – Режим доступу: URL: https://web.mit.edu/16.070/www/lecture/big_o.pdf.

5. Code optimization [Електронний ресурс]. – Режим доступу: URL: https://sites.google.com/a/case.edu/hpcc/guides-and-training/helpful-references/programming/code-optimization.

6. Оптимізація продуктивності веб-додатку [Електронний ресурс]. – Режим доступу: URL: https://reactjs.org/docs/optimizing-performance.html

7. Review of framework terminology [Електронний ресурс]. – Режим доступу: URL: https://riehle.org/computer-science/research/dissertation/chapter-2.html

8. Фреймворки у веб-розробці. [Електронний ресурс]. – Режим доступу: URL: https://web-creator.ru/articles/about_frameworks.

9. Single Page Web Applications: JavaScript end-to-end. September 2013 ISBN: 978-1-61729-075-6

10. Introduction JSX [Електронний ресурс]. – Режим доступу: URL: https://facebook.github.io/react/docs/introducing-jsx.html

11. Маніфест гнучкої розробки програмного забезпечення [Електронний ресурс]. – Режим доступу: URL: https://agilemanifesto.org/.

12. Introduction to Swagger [Електронний ресурс]. – Режим доступу: URL: https://swagger.io/docs/specification/2-0/what-is-swagger/.

13. Ризик передчасної оптимізації [Електронний ресурс]. – Режим доступу: URL: https://dtunkelang.medium.com/premature-optimization-is-still-the-root-of-all-evil-a3502c2ea262.

ДОДАТКИ

ДОДАТОК А. КОДЕКС АКАДЕМІЧНОЇ ДОБРОЧЕСНОСТІ ЗДОБУВАЧА ВИЩОЇ ОСВІТИ ХЕРСОНСЬКОГО ДЕРЖАВНОГО УНІВЕРСИТЕТУ

КОДЕКС АКАДЕМІЧНОЇ ДОБРОЧЕСНОСТІ
ЗДОБУВАЧА ВИЩОЇ ОСВІТИ
ХЕРСОНСЬКОГО ДЕРЖАВНОГО УНІВЕРСИТЕТУ

Я, Валяєв Кирило Віталійович,
учасник 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;

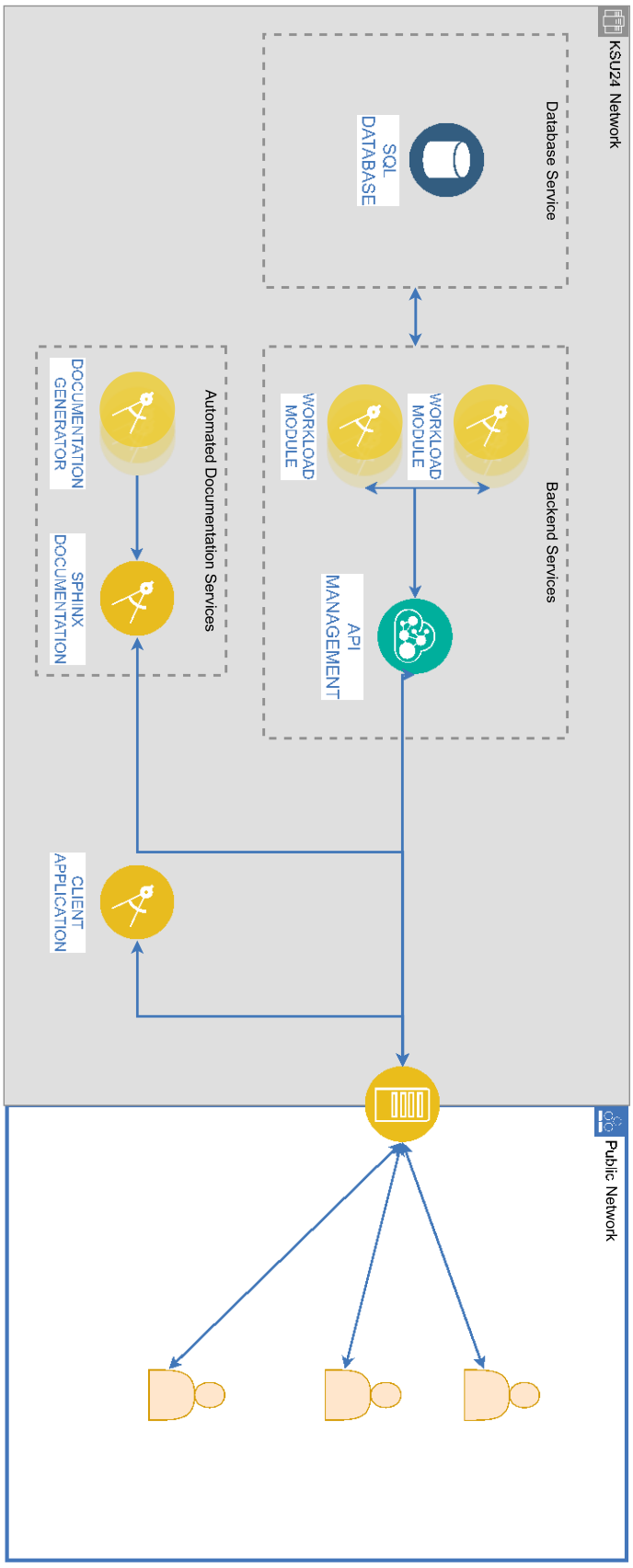
– надавати згоду на:
- безпосередню перевірку курсових, кваліфікаційних робіт тощо на ознаки наявності академічного плагіату за допомогою спеціалізованих програмних продуктів;
- оброблення, збереження й розміщення кваліфікаційних робіт у відкритому доступі в інституційному репозитарії;
- використання 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 науководослідницькі 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.

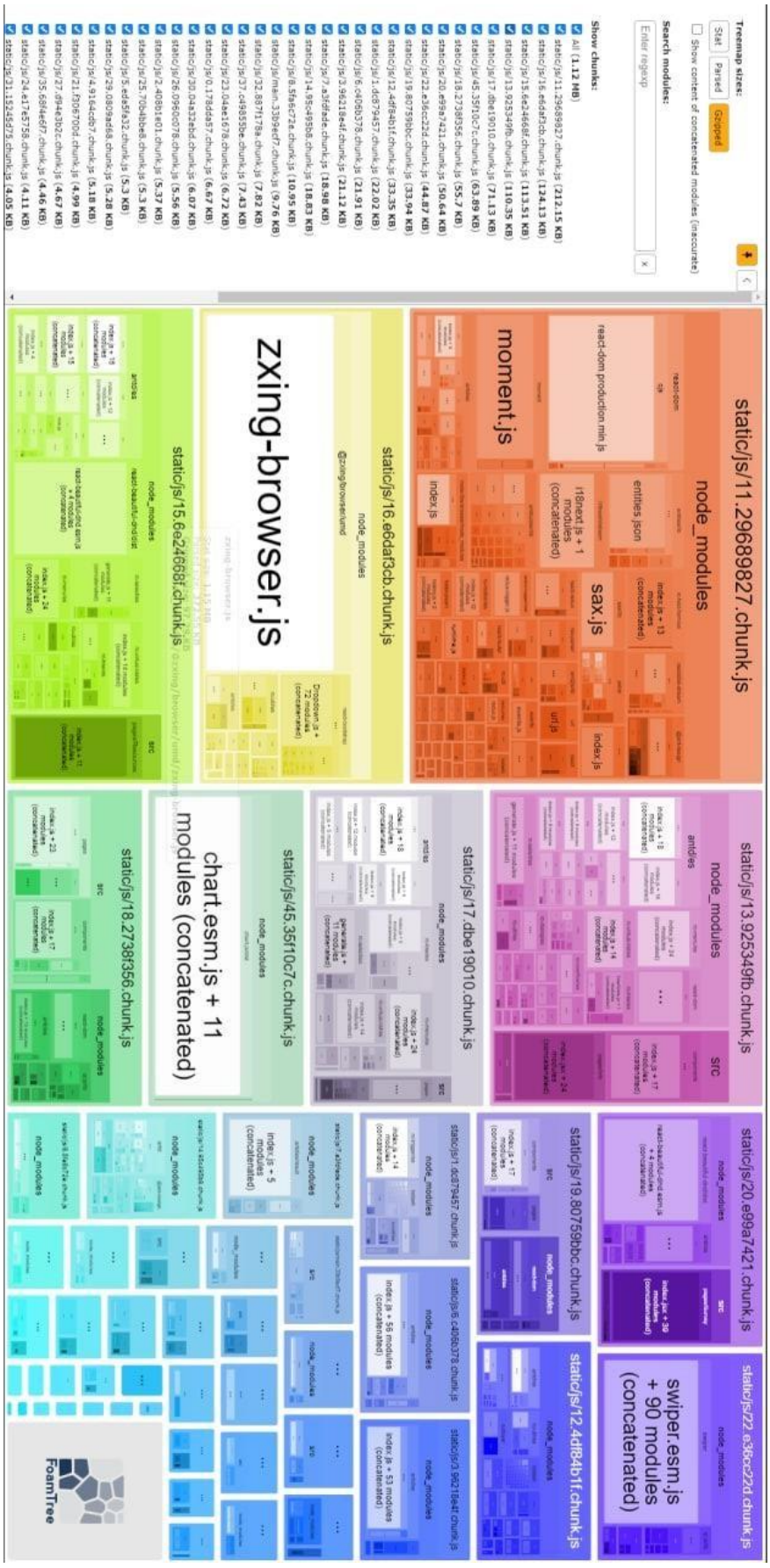
УСВІДОМЛЮЮ, що відповідно до чинного законодавства у разі недотримання Кодексу академічної доброчесності буду нести академічну та/або інші види відповідальності й до мене можуть бути застосовані заходи дисциплінарного характеру за порушення принципів академічної доброчесності.

20.11.2022		Кирило Валяєв
(дата)	(підпис)	(ім'я, прізвище)

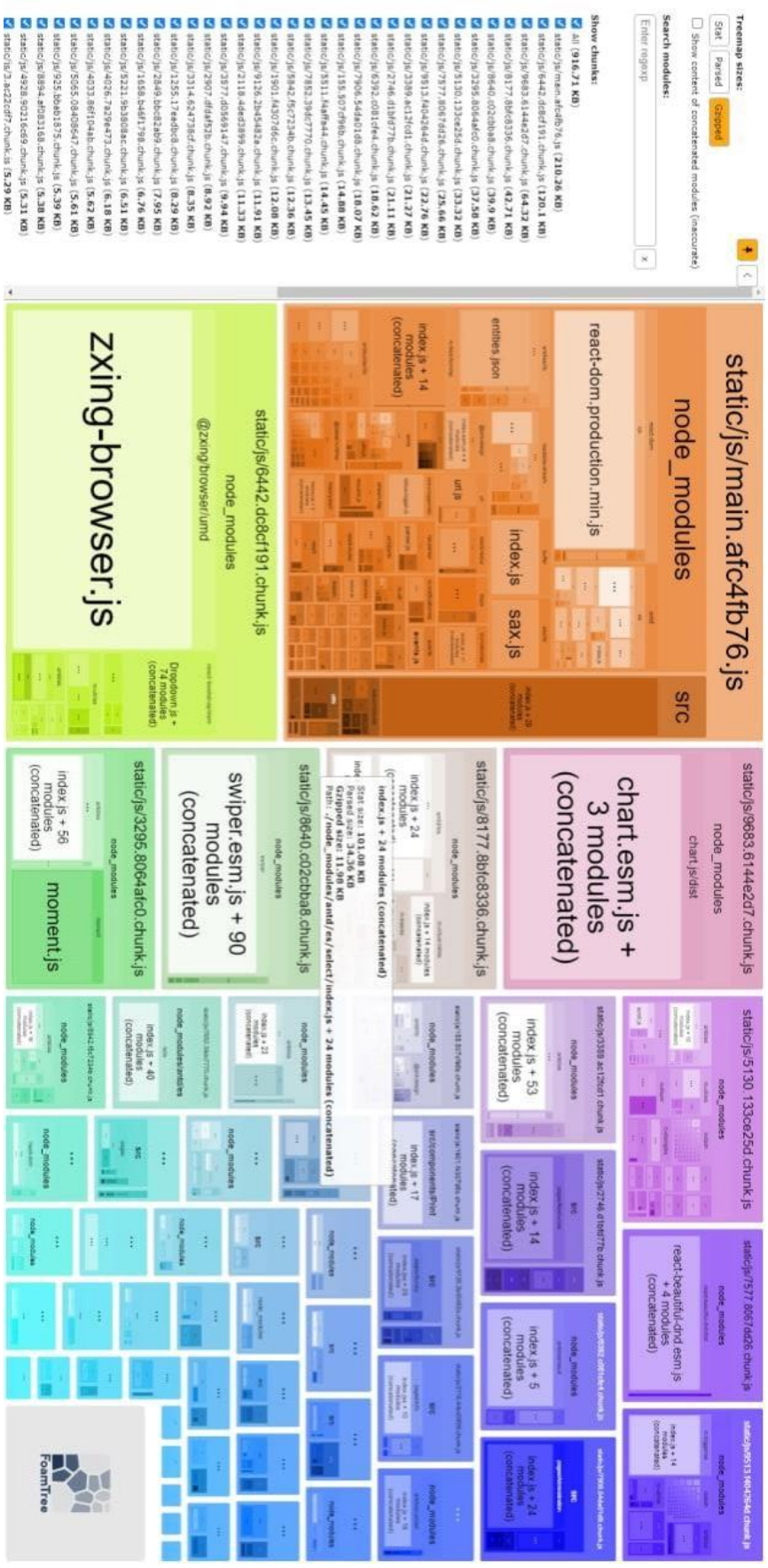
ДОДАТОК Б. ВІЗУАЛІЗАЦІЯ АРХІТЕКТУРИ СЕРВІСУ



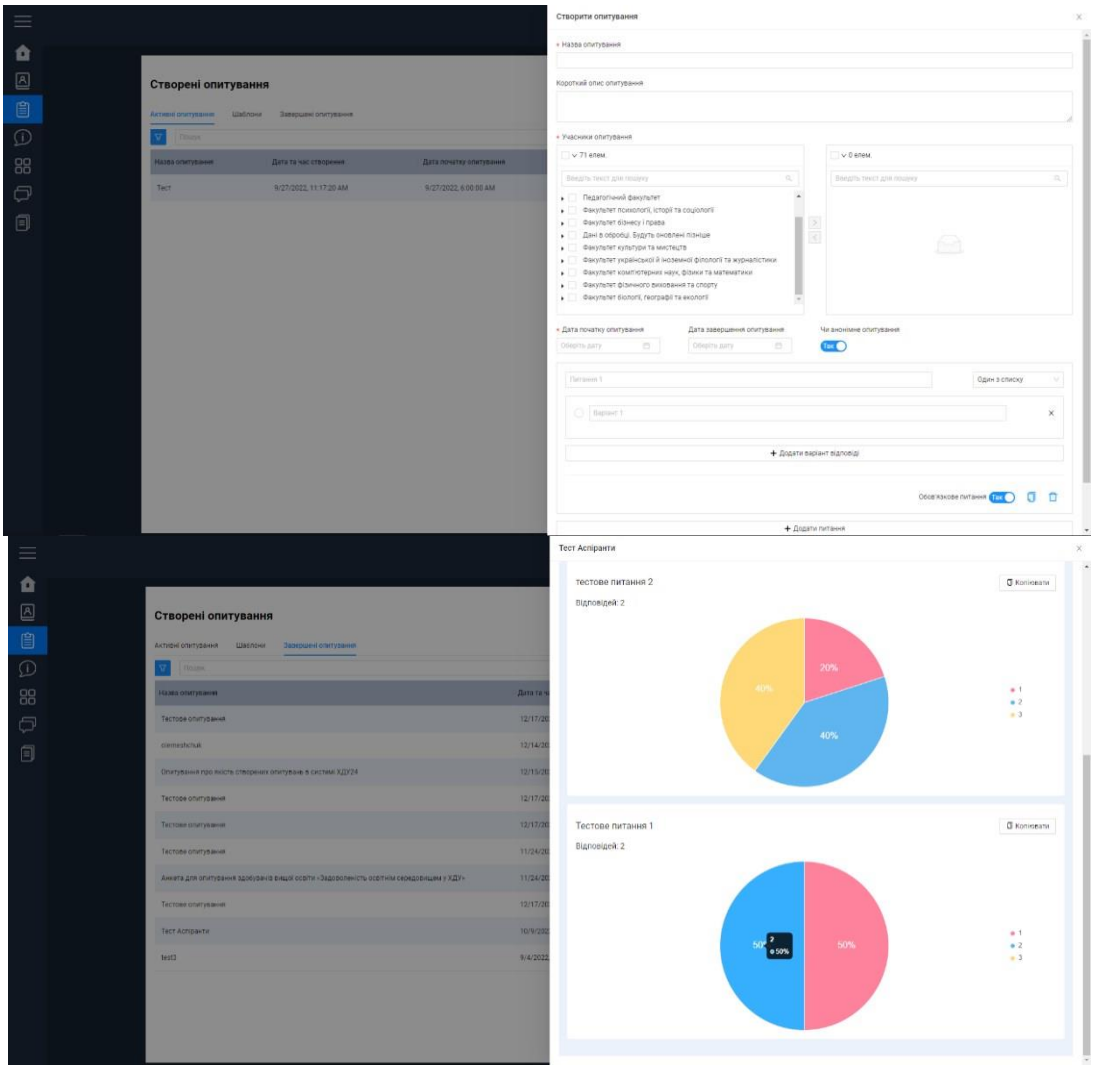
ДОДАТОК В. ФРАГМЕНТИ ВИКОНУВАНОГО КОДУ ДОДАТКУ ДО ОПТИМІЗАЦІЇ



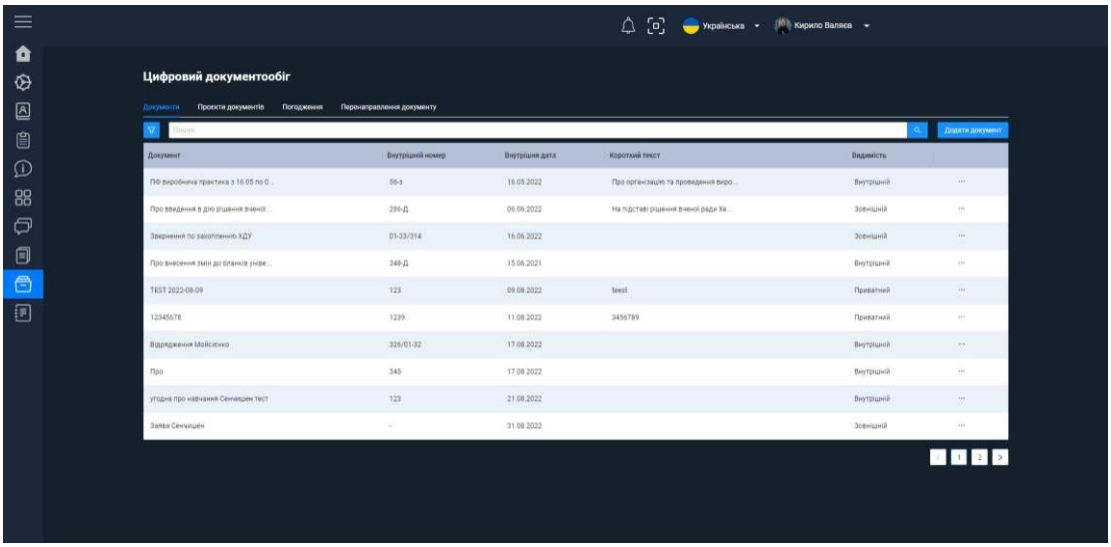
ДОДАТОК Г. ФРАГМЕНТИ ВИКОНУВАНОГО КОДУ ДОДАТКУ ПІСЛЯ ОПТИМІЗАЦІЇ

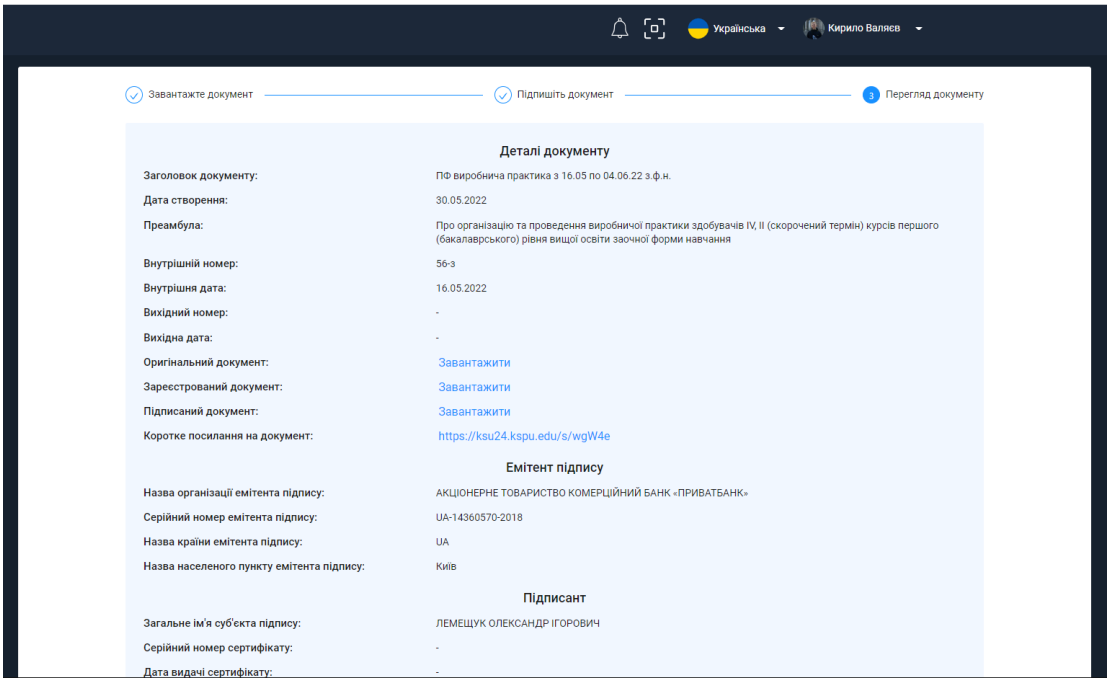


ДОДАТОК Д. МОДУЛЬ ОПИТУВАНЬ

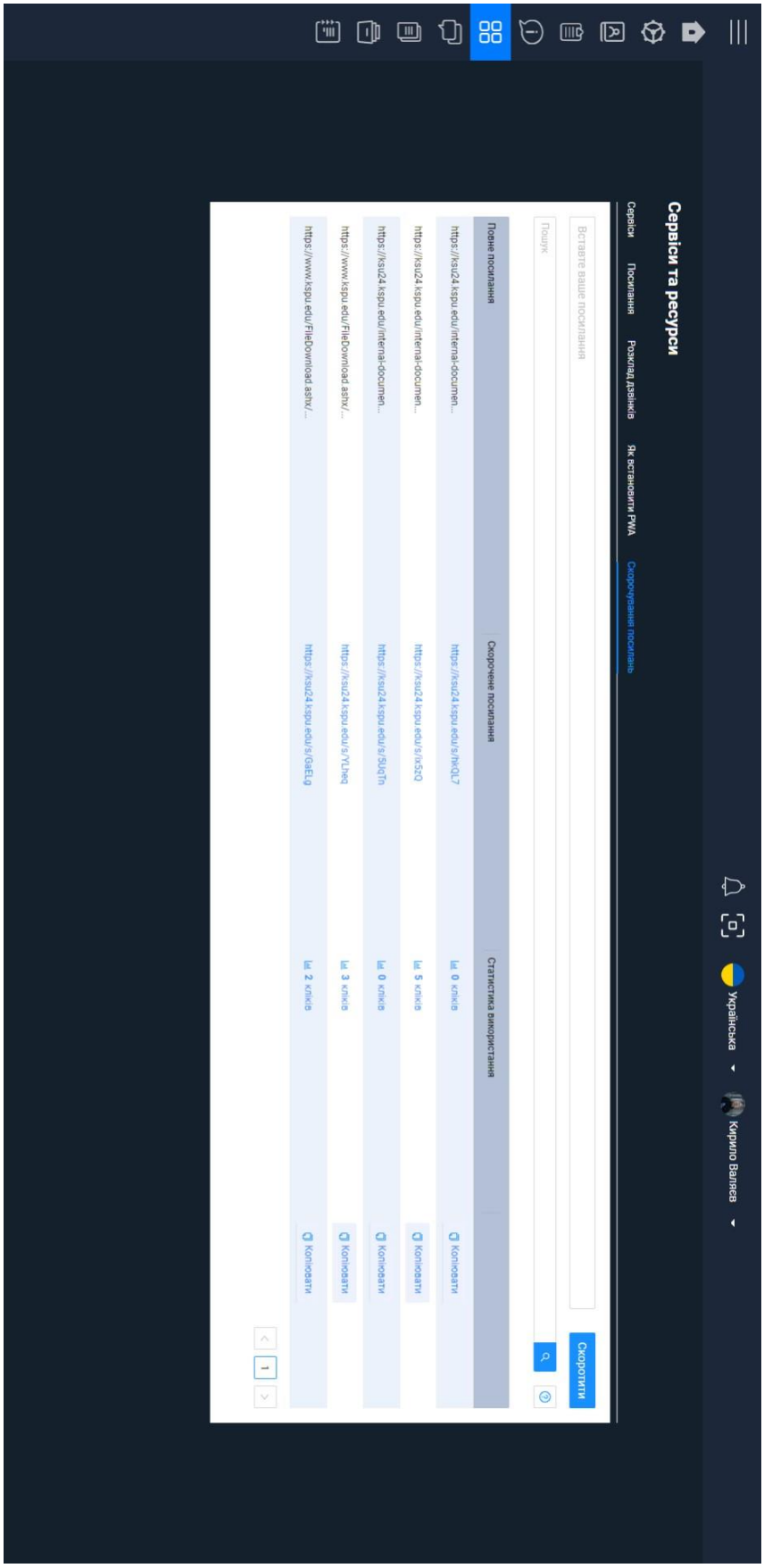


ДОДАТОК Е. МОДУЛЬ ДОКУМЕНТООБІГУ

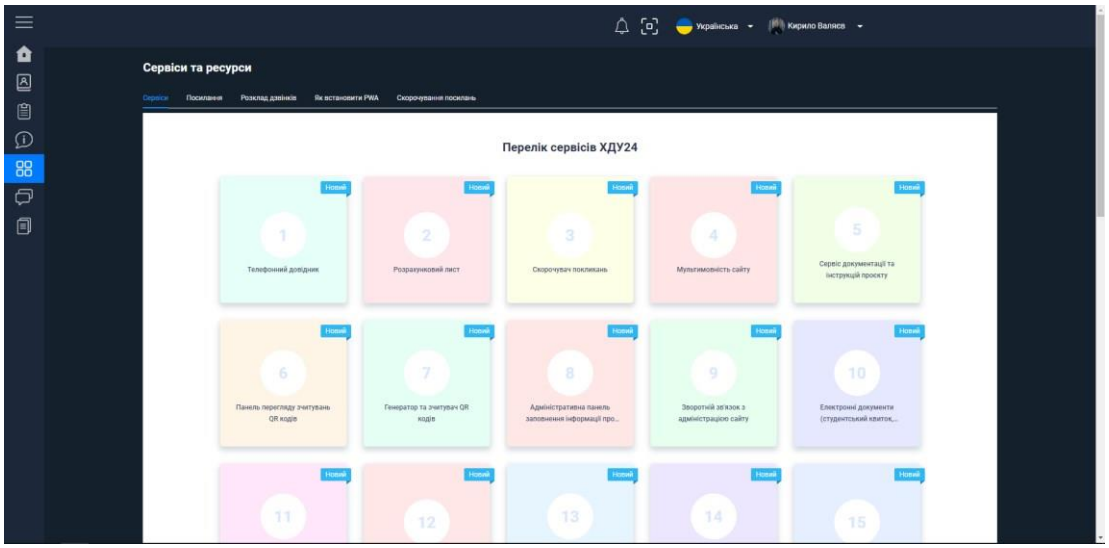


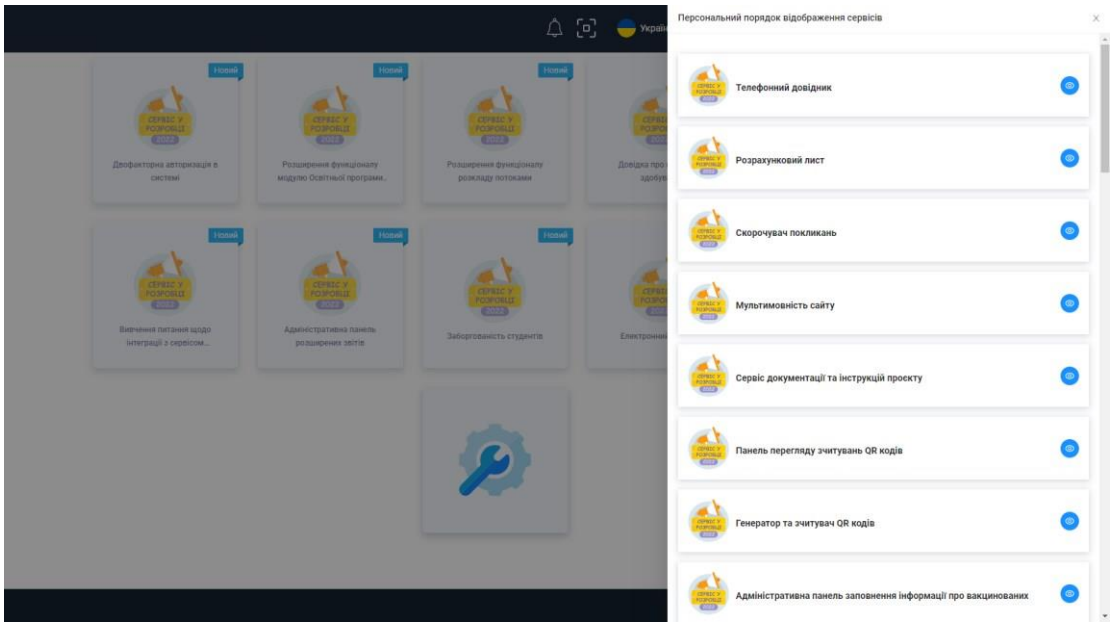


ДОДАТОК Є. МОДУЛЬ СКОРОЧУВАННЯ ПОСИЛАНЬ



ДОДАТОК Ж. МОДУЛЬ СЕРВІСІВ





UNICHECK

Ім'я користувача:
Оксана Блінова

ID перевірки:
1013034325

Дата перевірки:
27.11.2022 10:51:08 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
27.11.2022 12:59:50 EET

ID користувача:
94124

Назва документа: **121_ValiaievKV_fknfm_2022_check - Кирило Валяєв**

Кількість сторінок: **35** Кількість слів: **6732** Кількість символів: **51229** Розмір файлу: **75.27 KB** ID файлу: **1012738622**

2.24%
Схожість

Найбільша схожість: **2.24%** з джерелом з Бібліотеки (ID файлу: **1007607272**)

Не знайдено джерел з Інтернету

2.24% Джерела з Бібліотеки	1	Сторінка 37

0.59% Цитат

Цитати	2	Сторінка 38

Не знайдено жодних посилань

0.68%
Вилучень

Деякі джерела вилучено автоматично (фільтри вилучення: кількість знайдених слів є меншою за **8 слів** та **1%**)

0.28% Вилучення з Інтернету	42	Сторінка 39
0.45% Вилученого тексту з Бібліотеки	48	Сторінка 39

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	1

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Відгук наукового керівника
на кваліфікаційну роботу (проєкт)

Навчальний рік: 2021-2022
Факультет: комп'ютерних наук, фізики та математики
Спеціальність: 121 Інженерія програмного забезпечення
Освітньо-професійна (наукова) програма: «Інженерія програмного забезпечення»
Форма навчання: денна
Ступінь вищої освіти: другого (магістерського) рівня освіти
Тема: Проєктування та розроблення сервісної архітектури управління бізнес-процесами університету. Візуалізація та покращення функціональних інструментів веб-платформи
Виконавець: Валяєв Кирило Віталійович

Зміст відгуку:

Представлена кваліфікаційна робота студента Валяєва К.В. присвячена покращенню та автоматизації бізнес процесів Херсонського державного університету, як закладу вищої освіти, шляхом оптимізації вже реалізованого функціоналу клієнтської частини веб-платформи ХДУ24 та реалізації нових функціональних модулів. В ході виконання роботи були отримані наступні результати:

- Проаналізовано предметну область веб-додатку.
- Визначено критерії оптимального функціонування веб-додатку.
- Досліджено можливі способи оптимізації виконуваного коду.
- Визначені вимоги до розробки веб-додатку.
- Розроблено 4 функціональні модулі веб-додатку.

Під час виконання магістерської кваліфікаційної роботи студент Валяєв К. В. проявив себе грамотним, кваліфікованим спеціалістом здатним приймати самостійно складні технічні рішення та застосовувати отриманні за час навчання знання та навички. Робота успішно пройшла перевірку на унікальність у системі Unicheck, показавши рівень унікальності 97.76%.

Кваліфікаційна робота підготовлена з урахуванням вимог до робіт відповідного рівня для ЗВО III-IV рівнів вищої освіти, пройшла перевірку на академічну доброчесність і може бути рекомендована до захисту.

Науковий керівник _____ д.п.н. проф. Співаковський О.В.
(підпис) (наук. ступінь, вчене звання, П.І.Б.)

Дата 12.12.2022