ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Інкрементальний алгоритм обробки та мінімізації CSS і JavaScript файлів з інтелектуальним кешуванням для npm пакетів у Gulp»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Юрій ФЕТЬКО
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63
Юрій ФЕТЬКО

Керівник: Віталій ЗАЛИВА
ст. кафедри, доктор філософії

Рецензент: _____
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Фетьку Юрію Івановичу

1. Тема кваліфікаційної роботи: «Інкрементальний алгоритм обробки та мінімізації CSS і JavaScript файлів з інтелектуальним кешуванням для npm пакетів у Gulp»

керівник кваліфікаційної роботи Віталій ЗАЛИВА, ст. викладач кафедри, доктор філософії,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024 р.

3. Вихідні дані до кваліфікаційної роботи: структура веб-проектів із використанням CSS і JavaScript файлів, методи інкрементальної обробки даних та інтелектуального кешування, інструменти автоматизації.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз існуючих інструментів автоматизації збірки файлів (Gulp, Webpack, Grunt) та методів мінімізації CSS і JavaScript.

2. Дослідження механізмів кешування та інкрементальної обробки файлів у веб-розробці.
3. Формалізація математичної моделі інкрементального алгоритму обробки файлів із врахуванням залежностей між файлами.
4. Розробка алгоритму інтелектуального кешування з інтеграцією графів залежностей.
5. Реалізація розробленого алгоритму у вигляді npm-пакета для інтеграції з Gulp
6. Тестування алгоритму на основі реальних та симульованих проектів, оцінка його продуктивності порівняно з традиційними методами.
7. Аналіз результатів моделювання, включаючи показники часу збірки та використання ресурсів.
8. Розробка рекомендацій для впровадження алгоритму у веб-проектах.

5. Перелік ілюстративного матеріалу: *презентація*

1. Порівняння інструментів автоматизації збірки: gulp, webpack, grunt.
2. Порівняння технік кешування в gulp та розробленого алгоритму.
3. Формалізована модель об’єктів алгоритму.
4. Блок-схема алгоритму.
5. Формалізована модель реалізованого механізму інтелектуального кешування.
6. Блок-схема алгоритму механізму інтелектуального кешування.
7. Реалізація та публікація алгоритму як npm-пакету.
8. Інтеграція в існуючий проект
9. Порівняльний аналіз продуктивності алгоритмів
10. Проведення аналізу продуктивності алгоритмів
11. Результати моделювання

6. Дата видачі завдання «16» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10-23.10.2024	
2	Вивчення матеріалів для аналізу існуючих методів автоматизації збірки файлів (Gulp, Webpack, Grunt) та кешування	24.10-01.11.2024	
3	Дослідження методів інкрементальної обробки файлів	02.11-10.11.2024	
4	Аналіз впливу механізмів кешування на продуктивність збірки файлів	11.11-15.11.2024	
5	Розробка математичної моделі інкрементального алгоритму	15.11-18.11.2024	
6	Реалізація алгоритму інтелектуального кешування	18.11-20.11.2024	

7	Тестування розробленого алгоритму	20.11-22.11.2024	
8	Аналіз результатів тестування та формування висновків	22.11-24.11.2024	
9	Оформлення роботи: вступ, висновки, реферат	25.11-26.11.2024	
10	Розробка демонстраційних матеріалів	27.11-28.11.2024	
11	Попередній захист роботи	29.11.2024	

Здобувач вищої освіти _____ Юрій ФЕТЬКО
(підпис)

Керівник
кваліфікаційної роботи _____ Віталій ЗАЛИВА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 83 стор., 6 табл., 16 рис., 30 джерел.

Мета роботи – оптимізація процесу збірки та обробки CSS і JavaScript файлів шляхом розробки інкрементального алгоритму з використанням інтелектуального кешування для npm-пакетів у Gulp.

Об'єкт дослідження – процес автоматизованої збірки та обробки CSS і JavaScript файлів у веб-застосунках.

Предмет дослідження – механізми інкрементальної обробки, кешування, і мінімізації файлів із врахуванням залежностей між ними.

У роботі використано різноманітні методи, такі як: математичне моделювання для формалізації алгоритму, методи оптимізації для підвищення продуктивності збірки, а також сучасні бібліотеки для розробки та тестування алгоритмів.

Проведено аналіз існуючих інструментів автоматизації збірки, таких як Gulp, Webpack, і Grunt, а також методів кешування, зокрема gulp-cached і gulp-newer.

Розроблено інкрементальний алгоритм обробки файлів, який інтегрує графи залежностей для точнішого врахування змін у проєкті.

Проведено експерименти для оцінки ефективності розробленого алгоритму, які показали скорочення часу обробки файлів до 27.7% і зниження використання пам'яті на 30.4%.

КЛЮЧОВІ СЛОВА: ІНКРЕМЕНТАЛЬНИЙ АЛГОРИТМ, ІНТЕЛЕКТУАЛЬНЕ КЕШУВАННЯ, GULP, МАТЕМАТИЧНА МОДЕЛЬ, ОПТИМІЗАЦІЯ ЗБІРКИ, ГРАФИ ЗАЛЕЖНОСТЕЙ.

ABSTRACT

Text part of the master's qualification work: 83 pages, 16 pictures, 6 table, 30 sources.

The purpose of the work is to optimize the process of CSS and JavaScript file processing and bundling by developing an incremental algorithm with intelligent caching for npm packages in Gulp.

Object of research – the process of automated bundling and processing of CSS and JavaScript files in web applications.

Subject of research – mechanisms of incremental file processing, caching, and minimization with dependency consideration.

Summary of the work: The thesis focuses on the development of an incremental algorithm that integrates intelligent caching mechanisms and dependency graphs to optimize the processing and bundling of CSS and JavaScript files. The research involved an analysis of modern automation tools such as Gulp, Webpack, and Grunt, as well as existing caching techniques like gulp-cached and gulp-newer. A mathematical model of the algorithm was formalized, and its implementation as an npm package in Gulp demonstrated significant performance improvements. Experimental results showed a reduction of up to 27.7% in file processing time and a 30.4% decrease in memory usage.

KEYWORDS: INCREMENTAL ALGORITHM, INTELLIGENT CACHING, GULP, MATHEMATICAL MODEL, BUNDLING OPTIMIZATION, DEPENDENCY GRAPHS.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 10
ВСТУП 11
1 ОГЛЯД ЛІТЕРАТУРИ 14
1.1 Інструменти автоматизації збірки 15
1.2 Техніки обробки та мінімізації файлів 17
1.3 Механізми кешування 22
1.4 Алгоритми інкрементної обробки 27
2 РЕАЛІЗАЦІЯ 31
2.1 Розробка інкрементального алгоритму 31
2.2 Інтелектуальне кешування 37
2.3 Інтеграція з Gulp 45
3 ВПРОВАДЖЕННЯ ІНКРЕМЕНТАЛЬНОГО АЛГОРИТМУ У ВИГЛЯДІ NPM ПАКЕТА 50
3.1 Особливості npm 50
3.2 Ініціалізація пакета 54
3.3 Експортування алгоритму 56
3.4 Публікація в npm 58
3.5 Використання пакета 60
3.6 Показники продуктивності 63
3.7 Аналіз переваг кешування 67
ВИСНОВКИ 71
ПЕРЕЛІК ПОСИЛАНЬ 73
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 76

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

SPA – Single-Page Application (односторінковий веб-додаток).

CSS – Cascading Style Sheets (каскадні таблиці стилів).

JS – JavaScript (мова програмування для роботи з веб-контентом).

Gulp – Інструмент автоматизації процесів для веб-розробки.

NPM – Node Package Manager (менеджер пакетів для JavaScript).

MD5 – Message-Digest Algorithm 5 (алгоритм хешування).

JSON – JavaScript Object Notation (формат обміну даними).

HTTP – Hypertext Transfer Protocol (протокол передачі гіпертексту).

URL – Uniform Resource Locator (уніфікований покажчик ресурсу).

ПЗ – програмне забезпечення.

ОЗУ – оперативна пам'ять.

Граф залежностей – структура даних, яка показує взаємозв'язки між файлами.

Кеш – структура для зберігання даних для швидкого доступу.

Хеш – результат хешування, унікальний підпис для файлу.

ESLint – інструмент для перевірки якості JavaScript-коду.

Stylelint – інструмент для перевірки якості CSS-коду.

Webpack – інструмент для бандлінгу модулів веб-додатків.

Grunt – система автоматизації задач для JavaScript-проєктів.

ВСТУП

На сьогодні не можна побачити веб-застосунки які не включали в себе обробку і мінімізацію CSS та JS файлів[1, 2].

Вони мають значний вплив на продуктивність роботи веб-сайту. Для того, щоб отримати готові вже мінімізовані файли на допомогу приходять такі інструменти як: Gulp, Webpack, Grunt. Але методи які пропонують дані інструменти не завжди виконують покладену на них функцію і можуть обробляти всі файли, попри те, що зміни було внесено тільки у конкретні файли.

Актуальність даного дослідження відображається в умовах стрімкого зростання складності та обсягів веб-проектів, розробка ефективних алгоритмів автоматизації стає дедалі важливішою. Сучасні веб-додатки характеризуються високою динамічністю та залежністю між компонентами, що потребує нових підходів до оптимізації процесів збірки. Інкрементальний підхід до обробки файлів із використанням інтелектуального кешування — це рішення, що відповідає сучасним вимогам. Застосування такого підходу дозволяє не тільки скоротити час збірки, але й забезпечити гнучкість та точність у роботі з динамічними файлами.

Мета роботи: оптимізація процесу збірки веб-проектів через розробку інкрементального алгоритму обробки та мінімізації CSS і JavaScript файлів із використанням механізму інтелектуального кешування в середовищі Gulp.

Для досягнення поставленої мети в роботі потрібно виконати наступні завдання:

1. Провести аналіз чинних інструментів автоматизації збірки, оцінити їхні переваги, недоліки та сфери застосування.

2. Дослідити методи мінімізації CSS і JavaScript файлів; здійснити порівняння існуючих підходів з точки зору їх ефективності та складності.

3. Розглянути сучасні механізми інкрементальної обробки файлів і системи кешування; оцінити їхній вплив на продуктивність збірки.

4. Розробити інкрементальний алгоритм обробки файлів, який враховує залежності між ними, реалізувати механізм інтелектуального кешування для оптимізації.

5. Імплементувати розроблений алгоритм у середовищі Gulp, створити npm-пакет для його інтеграції у веб-проекти.

6. Провести тестування розробленого алгоритму, порівняти його результати з традиційними методами збірки; виконати аналіз продуктивності та ефективності.

Об'єктом дослідження є процес автоматизованої збірки та обробки CSS та JavaScript файлів у веб-застосунках.

Предмет дослідження - механізми інкрементальної обробки та кешування CSS і JavaScript файлів із врахуванням залежностей між файлами.

Методи дослідження базуються на системному підході та включають: методи аналізу та синтезу (для дослідження існуючих рішень), експериментальні методи (для тестування розробленого алгоритму), методи математичної статистики (для оцінки ефективності запропонованого рішення).

Наукова новизна роботи спрямована на розробку нового підходу до обробки файлів, що поєднує інкрементальний алгоритм із механізмом інтелектуального кешування. Запропонований підхід дозволяє враховувати залежності між файлами.

Практична значущість: розроблений алгоритм та створений npm-пакет можуть бути використані як у нових проектах, так і для модернізації вже існуючих веб-додатків. Це забезпечить скорочення часу збірки, зменшення витрат ресурсів і підвищення продуктивності проекту. Результати роботи також можуть бути корисними для навчальних дисциплін, пов’язаних із веб-розробкою та автоматизацією процесів.

Апробація результатів дослідження:

1. Фетько Ю.І., Залива В.В “Оптимізація процесу розробки програмного забезпечення шляхом впровадження алгоритму інкрементної обробки файлів” // Всеукраїнська науково-технічна конференція "Виклики та рішення в програмній інженерії". – Київ: ДУІКТ, 2024. – С. 79-80.

2. Фетько Ю.І., Залива В.В “Підвищення ефективності розробки програмного забезпечення шляхом оптимізації процесів обробки ресурсів” // Всеукраїнська науково-технічна конференція "Виклики та рішення в програмній інженерії". – Київ: ДУІКТ, 2024. – С. 204-206.

1 ОГЛЯД ЛІТЕРАТУРИ

У сучасній веб-розробці автоматизація процесів збірки є не лише вимогою часу, а й фундаментальною частиною створення ефективних і масштабованих додатків[11, 13]. З кожним роком зростає як обсяг коду, так і складність роботи з ресурсами: CSS та JavaScript файли повинні бути правильно оброблені, мінімізовані й організовані для забезпечення високої продуктивності системи. Ці завдання виходять за межі стандартних інструментів розробника й потребують спеціалізованих рішень. Саме тому були створені такі інструменти, як Gulp, Webpack та Grunt — кожен із яких має свої сильні сторони та недоліки. Вони стали своєрідним "еталоном" у сфері автоматизації завдань, даючи змогу значно скоротити час ручної роботи та зробити процес обробки файлів більш прогнозованим і контрольованим.

Автоматизація забезпечує не лише зменшення трудомісткості процесів, а й оптимізацію ключових параметрів веб-додатків: швидкості завантаження, споживання ресурсів браузера та загальної продуктивності. Проте, із розвитком веб-технологій стало очевидним, що традиційні методи обробки файлів часто не враховують специфіку великих проектів: наприклад, надлишкова обробка всіх файлів незалежно від змін або недостатнє врахування залежностей між модулями може створювати серйозні технічні обмеження.

Це питання породжує необхідність не лише в огляді існуючих рішень, але й у аналізі їхніх підходів до оптимізації процесів. Оскільки вибір інструменту для автоматизації збірки залежить від конкретних умов та вимог проекту, у цьому розділі розглядаються основні характеристики й можливості сучасних інструментів, які дозволяють автоматизувати збірку веб-додатків, з подальшим акцентом на розробці інноваційних рішень для покращення їхньої ефективності.

1.1 Інструменти автоматизації збірки

Автоматизація процесів збірки - одна з найважливіших складових сучасної веб-розробки. Зі зростанням масштабів та складності проектів, розробники стикаються з необхідністю зменшення ручної праці та оптимізації робочих процесів. У цьому контексті особливої популярності набули інструменти автоматизації, такі як Gulp, Webpack та Grunt. Вони не лише спрощують процес обробки файлів, але й забезпечують стабільність, продуктивність і передбачуваність результатів.

Gulp - це інструмент, який широко використовується для виконання завдань у веб-розробці[3]. Його головна особливість - застосування потокової обробки файлів (streaming), що дозволяє значно скоротити час виконання завдань[4, 5]. Завдяки простій архітектурі, створення конфігураційного файлу (Gulpfile) є інтуїтивно зрозумілим навіть для початківців. Основні переваги Gulp включають:

- Простоту створення та налаштування завдань;
- Ефективну обробку файлів за допомогою потоків;
- Велику кількість плагінів для автоматизації завдань, таких як мінімізація, компіляція та об’єднання файлів;

Однак, Gulp має і певні недоліки. Один із головних - традиційний підхід до обробки файлів: кожен запуск процесу передбачає повторну обробку всієї кодової бази, незалежно від того, чи були файли змінені. Це створює додаткове навантаження на систему та збільшує час виконання.

Webpack є більш складним і потужним інструментом для автоматизації збірки[29]. Його головна перевага - підтримка модульної архітектури. Webpack дозволяє використовувати *code splitting* для поділу коду на частини, що оптимізує швидкість завантаження веб-додатків. Крім того, Webpack підтримує динамічний імпорт, що значно спрощує управління залежностями. Але незважаючи на переваги, початкове налаштування Webpack може бути складним для новачків. Велика кількість параметрів і гнучкість інструменту часто створюють проблеми на

етапі конфігурації. До того ж, за недостатньо оптимізованої конфігурації - час компіляції для великих проектів може суттєво збільшуватися.

Grunt - один із найперших інструментів автоматизації, що заклав основу для таких сучасних рішень, як Gulp та Webpack. Його головна відмінність полягає у використанні тимчасових файлів для обробки, що робить його менш ефективним у порівнянні з потоковою обробкою Gulp. Проте Grunt все ще використовується у стабільних проектах завдяки простій архітектурі та великій кількості плагінів. Основними недоліками Grunt є його відносно повільна робота і складність налаштування для великих проектів.

Нижче наведена таблиця 1.1 з порівнянням інструментів Gulp, Webpack та Grunt, де наведено основні плюси та мінуси кожного інструменту:

Таблиця 1.1

Порівняння інструментів збірки

Інструмент	Переваги	Недоліки
Gulp	- Простота налаштування; - Використання потоків для обробки файлів, що пришвидшує процеси; - Велика кількість плагінів для різних завдань; - Гнучкість і простота написання завдань;	- Обробляє всі файли незалежно від змін; - Вимагає сторонніх плагінів для завдань із об’єднання модулів у єдиний файл (бандлінг).
Webpack	- Підтримує об’єднання модулів (бандлінг), що спрощує роботу з сучасним JavaScript; - Широкі можливості налаштування; - Code splitting для покращення продуктивності;	- Складна конфігурація, що потребує налаштування і додаткових знань; - Може бути надмірно складним для малих проектів.

Продовження таблиці 1.1

Порівняння інструментів збірки

Інструмент	Переваги	Недоліки
Grunt	- Велика кількість плагінів, що дозволяють виконувати різні завдання; - Стабільність і підтримка великих проєктів.	- Застаріла архітектура, яка працює повільніше, ніж Gulp і Webpack; - Менш ефективний підхід до обробки файлів через використання тимчасових файлів.

Таким чином, вибір інструменту автоматизації залежить від специфіки проекту та вимог команди. Для невеликих і середніх проектів Gulp є оптимальним вибором завдяки простоті та гнучкості; для великих проектів або SPA-додатків Webpack є найкращим варіантом через потужні можливості конфігурації. Grunt, у свою чергу, доцільно використовувати для стабільних старих додатків, де вже налаштована робоча інфраструктура. Однак розвиток сучасної веб-розробки диктує необхідність нових підходів, таких як інкрементальні алгоритми обробки файлів, що пропонують ще більш ефективні рішення.

1.2 Техніки обробки та мінімізації файлів

Мініфікація є ключовим прийомом оптимізації веб-застосунків[27, 28], спрямованим на підвищення продуктивності шляхом зменшення розміру файлів CSS і JavaScript. Цей процес передбачає методичне видалення надлишкових символів (пробілів, коментарів, розривів рядків) без порушення функціональності коду, що суттєво впливає на швидкість обробки файлів браузером.

З огляду на зростаючу складність сучасних веб-застосунків, обсяг необхідних CSS і JavaScript ресурсів демонструє постійну тенденцію до збільшення. Емпіричні

дослідження показують пряму кореляцію між розміром файлів і часом завантаження сторінки, що безпосередньо впливає на:

- користувацький досвід (UX);
- рейтинг у пошукових системах;
- конверсію відвідувачів;

Крім того, мініфікація допомагає зменшити необхідну пропускну здатність, що може бути особливо корисним для користувачів із повільнішим підключенням до Інтернету або обмеженням даних. Цей процес дозволяє швидше завантажувати різні елементи веб-сторінок, забезпечуючи безперебійну взаємодію користувача. Зокрема, у поєднанні з іншими методами оптимізації, такими як кешування та стиснення, мініфікація підвищує загальну ефективність доставки веб-вмісту.

Обробка та мінімізація файлів є важливими етапами під час підготовки до випуску у версію виробництва (продакшн) веб-додатків[4, 28]. CSS та JavaScript файли можуть містити зайві пробіли, коментарі, та непотрібні символи, що збільшує їхній розмір і час завантаження.

Мінімізація CSS файлів полягає у видаленні всіх непотрібних символів (пробіли, коментарі, нові рядки), які не впливають на функціональність. Інструменти, як-от CSSNano та CleanCSS, дозволяють суттєво зменшити розмір файлів, що покращує швидкість завантаження веб-сторінок.

CSSNano - це інструмент який використовує постпроцесор PostCSS для видалення зайвих пробілів, коментарів та оптимізації коду[24]. Може бути інтегрований у різні збірки, такі як Webpack або Gulp, через відповідні плагіни.

CleanCSS - це швидкий і ефективний інструмент, який зменшує розмір файлу за рахунок видалення зайвих символів та оптимізації структури коду. Підтримує різні рівні оптимізації, включаючи злиття медіа-запитів та видалення дублюючих правил.

Для кращого розуміння можливостей даних бібліотек, створимо порівняльну таблицю 1.2 для визначення ефективності відповідно

Таблиця 1.2

Порівняльний аналіз інструментів мінімізації CSS

Характеристика	CSSNano	CleanCSS
Базова мінімізація	Так	Так
Злиття медіа-запитів	Так	Так
PostCSS інтеграція	Так	Ні
Оптимізація селекторів	Повна	Часткова
Видалення дублікатів	Так	Так
Підтримка препроцесорів	Так	Обмежена

JavaScript мінімізація являє собою комплексний процес оптимізації, що включає: видалення надлишкових символів, оптимізацію структури коду та обфускацію (процес перетворення змінних і функцій у короткі символи). Сучасні інструменти мінімізації, такі як UglifyJS та Terser, забезпечують різні рівні оптимізації коду. Для кращого розуміння можливостей JavaScript мініфікаторів результати наведені у таблиці 1.3

Таблиця 1.3

Порівняння можливостей JavaScript мініфікаторів

Функціональність	UglifyJS	Terser
Мінімізація базового JS	Так	Так
Підтримка ES6+	Ні	Так
AST оптимізація	Часткова	Повна
Видалення мертвого коду (Dead code elimination)	Так	Так
Мапінг вихідного коду (Source maps)	Так	Так
Збереження коментарів	Опціонально	Опціонально

AST оптимізація (Abstract Syntax Tree — абстрактне синтаксичне дерево) - це представлення вихідного коду у вигляді деревоподібної структури, де кожен вузол відповідає конструкції мови програмування, дозволяє аналізувати та трансформувати код на структурному рівні.

Dead code elimination (видалення мертвого коду) - це процес виявлення та видалення коду, який ніколи не виконується, який включає: видалення недосяжного коду, видалення невикористовуваних змінних, спрощення умовних виразів.

Загальний алгоритм мінімізації та обробки файлів зображено на рисунку 1.1

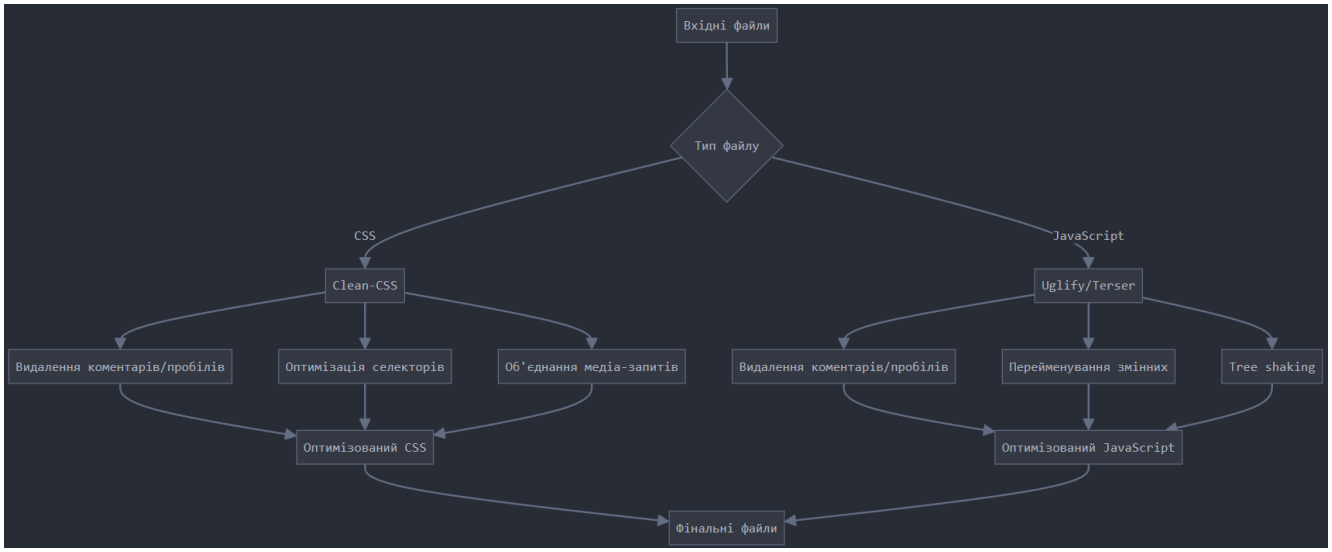


Рис. 1.1 Процес мінімізації CSS та JavaScript файлів

Техніка об'єднання файлів (concatenation) є важливим підходом в оптимізації веб-додатків. Вона передбачає злиття кількох файлів в один для зменшення кількості HTTP-запитів – однієї з головних причин уповільнення завантаження сторінок. Особливо значущим цей процес стає у великих проєктах, де наявна велика кількість залежностей і окремих файлів, що потребують інтеграції.

У контексті Gulp для об'єднання файлів часто використовують gulp-concat – потужний і зручний у використанні плагін. Він забезпечує ефективне злиття кількох файлів JavaScript або CSS в один файл, що зменшує кількість запитів, покращуючи швидкість завантаження. Крім того, цей плагін дозволяє легко

інтегрувати його в конвеєр завдань Gulp разом з іншими плагінами, такими як мініфікатори або транспілятори.

У тандемі gulp-concat і gulp-uglify дозволяють розробникам не лише об'єднувати файли для зменшення кількості HTTP-запитів, але й ефективно мінімізувати їхній розмір. Цей підхід забезпечує покращення швидкості завантаження веб-сторінок, зниження використання ресурсів та підвищення загальної продуктивності. Переваги використання комбінації gulp-concat і gulp-uglify:

- Зменшення кількості HTTP-запитів;
- Оптимізація розміру файлів;
- Покращення швидкості завантаження;
- Гнучкість налаштування;
- Інтеграція з існуючими процесами збірки.

Загальна схема роботи з файлами продемонстрована на рисунку 1.2

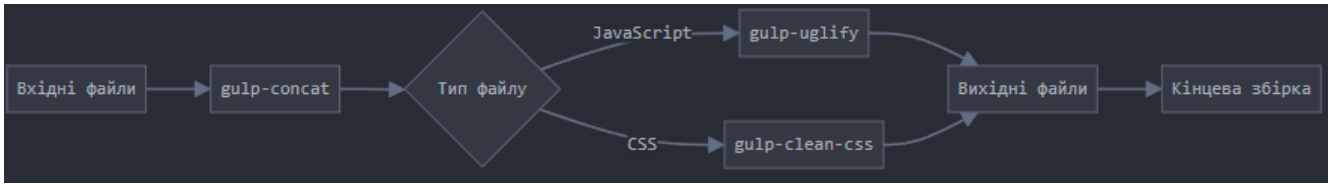


Рис. 1.2 Схема роботи з файлами

Таким чином, використання описаних плагінів у системі збірки Gulp забезпечує ефективну оптимізацію веб-ресурсів, що призводить до покращення продуктивності веб-застосунків та користувацького досвіду.

Підсумовуючи, мініфікація є критично важливим процесом у сфері веб-розробки. Усуваючи непотрібні символи з файлів CSS і JavaScript, це забезпечує швидший час завантаження та покращений досвід користувача. Простий, але ефективний характер мініфікації робить її важливою практикою для розробників, які прагнуть оптимізувати свої веб-сайти для кращої продуктивності та залучення користувачів.

1.3 Механізми кешування

Кешування — це технологія оптимізації[16, 17], яка передбачає збереження оброблених даних у тимчасовій пам’яті для їх повторного використання без необхідності повторної обробки. У процесі виконання завдань кешування дозволяє значно скоротити час доступу до даних, зменшуючи навантаження на систему та підвищуючи загальну ефективність обчислень. Розробники можуть скористатися швидшими циклами зворотного зв’язку під час кодування, тестування та налагодження, коли зміни файлів запускають лише необхідні етапи обробки.

Принцип роботи звичайного кешування:

- Збереження даних: після першого виконання операції результати обробки файлів зберігаються у спеціальному сховищі (кеші);
- Повторне використання: при наступних зверненнях до тих самих даних система використовує кешовані результати;
- Оновлення кешу: кеш залишається незмінним до ручного очищення або до спеціального налаштування, яке може автоматично оновлювати його після заданого часу;

Переваги звичайного кешування:

- Простота реалізації;
- Прискорення доступу до часто використовуваних даних;
- Зменшення навантаження на систему;

Але існують певні недоліки, такі як:

- Відсутність механізмів перевірки змін у даних: система може використовувати застарілі версії файлів;
- Обмежена гнучкість: не підходить для складних систем із динамічними залежностями;

Наприклад, у веб-розробці звичайне кешування застосовується для статичних ресурсів, таких як зображення, CSS або JavaScript файли, коли немає необхідності часто оновлювати ці дані.

Інтелектуальне кешування - це вдосконалений механізм, що поєднує традиційні методи з адаптивними алгоритмами перевірки змін у даних[16, 17]. Цей підхід забезпечує актуальність даних, динамічне оновлення кешу та більш ефективне використання ресурсів.

Цей механізм постійно оновлює кеш у відповідь на зміни даних, використовуючи такі алгоритми перевірки, як ETag, Last-Modified або спеціальні рішення. Зокрема даний метод може використовувати машинне навчання для прогнозування майбутніх запитів даних на основі аналізу поведінки користувачів або статистики доступу. Цей механізм використовує логіку для автоматичного вибору найкращих стратегій кешування залежно від завантаження сервера, часу доби чи інших умов. Це дозволяє коригувати правила кешування в реальному часі.

До особливостей інтелектуального кешування належать:

- Перевірка актуальності: система використовує алгоритми, такі як порівняння хешів файлів або аналіз дати останньої модифікації (ETag, Last-Modified), щоб визначити, чи змінювався файл.
- Автоматичне оновлення: при виявленні змін у файлі кеш автоматично оновлюється.
- Аналіз залежностей: враховуються зв’язки між файлами, що дозволяє уникати помилок у проектах із динамічними залежностями.

Перевагами ж є:

- Обробка лише змінених файлів;
- Зменшення обсягу непотрібних операцій;
- Забезпечення актуальності даних;
- Підвищення ефективності у великих проектах із численними залежностями;

Також є й недоліки, які можна охарактеризувати як:

- Вища складність реалізації порівняно із звичайним кешуванням;
- Вимагає більше ресурсів для перевірки змін;

Для порівняння цих двох методів була створена табличка 1.3 з характеристикою кожного з кешувань.

Таблиця 1.3

Порівняння звичайного та інтелектуального кешування

Характеристика	Звичайне кешування	Інтелектуальне кешування
Основний принцип	Зберігає оброблені дані для повторного використання.	Зберігає дані та перевіряє їх актуальність перед використанням.
Перевірка змін	Відсутня: кеш працює без перевірки стану даних.	Виконується: перевіряються хеші файлів або дати змін.
Оновлення кешу	Вручну або за заданим часом.	Автоматично при зміні файлу.
Ефективність	Швидкий доступ, але можливе використання застарілих даних.	Швидкий доступ із гарантією актуальності даних.
Оптимізація обробки	Уникає повторної обробки, але може зберігати застарілі дані.	Обробляє лише змінені файли, забезпечуючи оптимальний результат.
Складність реалізації	Низька: простий механізм зберігання й доступу до даних.	Висока: потребує додаткових алгоритмів і логіки перевірки.
Приклад використання	Зберігання статичних ресурсів (зображень, CSS, JS).	Обробка CSS і JS у Gulp із плагінами (наприклад, gulp-cached).

У Gulp, популярному інструменті автоматизації збірки, доступні кілька механізмів кешування, що спрямовані на оптимізацію обробки файлів. Ці механізми допомагають зменшити час виконання завдань і підвищити продуктивність, зосереджуючись на обробці лише тих файлів, які зазнали змін. До основних плагінів для кешування в Gulp належать gulp-cached, gulp-remember та gulp-newer. Кожен із цих плагінів виконує унікальну роль у процесі оптимізації збірки файлів.

Плагін gulp-cached реалізує кешування файлів у пам’яті, що дозволяє уникнути повторної обробки файлів, які залишилися незміненими з часу останнього виконання завдання. Його основними перевагами є, скорочення часу обробки, завдяки кешуванню незмінених файлів значно зменшується обсяг виконуваної роботи, що особливо помітно у великих проектах із численними файлами. Простота інтеграції в роботі з іншими бібліотеками. Але є і недоліки, а

саме – локальність кешу. Кеш працює лише в межах одного завдання. Після завершення або перезапуску Gulp кеш скидається, що змушує систему повторно обробляти всі файли. Також присутню обмежена гнучкість. Плагін не зберігає стану між різними завданнями чи сеансами.

Наступний плагін gulp-remember працює у зв'язці з gulp-cached, дозволяючи "запам'ятовувати" всі файли, які були оброблені раніше. Це особливо корисно для завдань, які вимагають повного набору вхідних файлів. Його основні переваги це - збереження повного набору файлів. Навіть якщо деякі файли пропускаються за допомогою gulp-cached, gulp-remember додає їх назад до потоку для наступних завдань. Також він ефективно працює разом із плагінами для конкатенації або об'єднання файлів.

Але його основний недолік - відсутність перевірки змін. Плагін не перевіряє, чи були файли змінені, тому його необхідно використовувати разом із gulp-cached або іншими інструментами для перевірки змін.

Gulp-remember ідеально підходить для завдань, що вимагають об'єднання кількох файлів у один (concatenation) та підготовки повного набору файлів для фінальної збірки. Приклад інтеграції зображено на рисунку 1.3

```
14  import cached from 'gulp-cached';
15  import remember from 'gulp-remember';
16  import concat from 'gulp-concat';
17
18  gulp.task('scripts', function () {
19      return gulp.src('src/js/**/*.js')
20          .pipe(cached( name: 'scripts')) // кешування
21          .pipe(remember( cacheName: 'scripts')) // відновлення повного набору файлів
22          .pipe(concat( file: 'all.js')) // об'єднання файлів
23          .pipe(gulp.dest('dist/js'));
24  });
```

Рис. 1.3 Інтеграція з gulp плагіну Gulp-remember

Наступний плагін gulp-newer орієнтований на обробку лише тих файлів, які були змінені після останньої збірки. Він порівнює дати модифікації вихідних файлів із кінцевими файлами в каталозі виведення(dist).

Його основні переваги заключаються в оптимізації часу та простої інтеграції, так як він обробляє тільки нові або змінені файли, що значно прискорює процес збірки.

Але його головний недолік полягає в ігноруванні залежностей між файлами. Він не враховує залежності між файлами, що може викликати проблеми у проектах із динамічними залежностями.

Щоб зрозуміти різницю між наданими плагінами, наведо таблицю 1.4 під назвою порівняння механізмів кешування у gulp.

Таблиця 1.4

Порівняння механізмів кешування в Gulp

Плагін	Призначення	Переваги	Недоліки	Сценарій використання
Gulp-cached	Кешує файли в пам’яті, щоб уникнути повторної обробки незмінених файлів.	Скорочує час обробки; Простий у використанні	Кеш скидається при перезапуску Gulp; Працює тільки для одного завдання.	Використовується для пропускання незмінених файлів.
Gulp-remember	"Запам’ятовує" всі оброблені файли для використання у наступних етапах.	Зберігає повний набір файлів; Добре працює з gulp-cached	Не перевіряє зміни у файлах; Потребує інтеграції з іншими плагінами для повної функціональності	Завдання, які вимагають повного набору файлів (об’єднання, конкатенація)
Gulp-newer	Порівнює час модифікації файлів у source і dest для обробки лише змінених	Ефективна робота зі зміненими файлами; Простий у налаштуванні	Не враховує залежності між файлами; Підходить тільки для завдань без складних залежностей	Мінімізація, копіювання чи інші завдання, що потребують обробки лише змінених файлів

У процесі дослідження та реалізації механізму кешування для автоматизації збірки я зробив вибір на користь інтелектуального кешування з кількох ключових причин.

По-перше, традиційне кешування, хоча й забезпечує певне прискорення завдань, має низку обмежень: воно не враховує зміни в залежностях файлів і може використовувати застарілі дані, що призводить до додаткових витрат часу та ресурсів. Інтелектуальне кешування вирішує ці проблеми завдяки перевірці актуальності даних на основі хешів або дат модифікації. Це дозволяє уникнути обробки незмінених файлів і забезпечити актуальність результатів.

По-друге, інтелектуальне кешування дозволяє динамічно враховувати зміни в залежностях між файлами, що є критично важливим для великих проєктів із численними взаємопов’язаними модулями. Наприклад, у випадку змін у файлі залежності, система автоматично обробляє пов’язані файли, мінімізуючи ризик помилок і забезпечуючи цілісність проекту.

На відміну від традиційних рішень, таких як плагіни gulp-cached, gulp-remember та gulp-newer, які вже широко використовуються у процесі автоматизації збірки файлів, їхні можливості мають певні обмеження. Основними недоліками цих підходів є:

- обмежена підтримка динамічних залежностей між файлами;
- необхідність вручну налаштовувати поведінку кешу або очищувати його;
- відсутність механізмів для глибокої перевірки зв’язків між файлами.

Ці обмеження відкривають перспективу для пошуку нових, більш адаптивних рішень, що враховують складну структуру проектів із великою кількістю залежностей.

1.4 Алгоритми інкрементної обробки

Інкрементна обробка даних представляє собою фундаментальний підхід у сучасній розробці програмного забезпечення[23, 25], що базується на принципі поступового оновлення та обробки інформації. В контексті оптимізації веб-

розробки, даний підхід набуває особливого значення при роботі з статичними ресурсами та їх трансформацією.

Основоположним принципом інкрементної обробки є селективний підхід до модифікації даних[23, 25], при якому обробці підлягають виключно змінені елементи системи, що суттєво знижує обчислювальне навантаження та оптимізує використання системних ресурсів. Нижче наведена таблиця 1.5 класифікації алгоритмів із поясненням їхніх особливостей та областей застосування:

Таблиця 1.5

Види інкрементальних алгоритмів

Категорія	Принцип роботи	Область застосування	Ефективність
Диференціальні	Аналіз змін між поточною та попередньою версіями файлів; обробляються лише відмінності.	Системи контролю версій (наприклад, Git).	Висока
Кумулятивні	Зберігають накопичені зміни, які обробляються партіями.	Системи кешування, де важлива періодична оптимізація.	Середня
Реактивні	Миттєва реакція на будь-які зміни у системі, з автоматичною обробкою відповідних даних.	Інтерактивні системи реального часу (наприклад, чат-боти, стримінг).	Максимальна
Предиктивні	Прогнозують можливі зміни даних і готують відповідні ресурси заздалегідь.	Системи машинного навчання та аналітичні інструменти.	Варіативна

Їхні особливості можна розділити як:

- Диференціальні алгоритми є найпоширенішими, адже дозволяють уникнути повторної обробки незмінних файлів. Наприклад, системи контролю версій, такі як Git, використовують диференціальний підхід для визначення відмінностей між комітами.

- Кумулятивні алгоритми корисні для оптимізації роботи систем кешування, де накопичені зміни можна обробляти, наприклад, у нічний час, щоб мінімізувати вплив на продуктивність у пікові періоди.
- Реактивні алгоритми забезпечують швидку адаптацію до змін, що робить їх незамінними для інтерактивних платформ, таких як відеоконференції чи системи онлайн-чатів.
- Предиктивні алгоритми, хоч і мають високу варіативність ефективності, є перспективним напрямком, оскільки дозволяють використовувати прогнози для оптимізації процесів.

Система автоматизації збірки Gulp є однією з найпопулярніших платформ завдяки своїй потоковій архітектурі, що дозволяє ефективно організувати процеси обробки файлів. Проте існуючі підходи до інкрементної обробки в Gulp мають певні обмеження, які спонукають до розробки нових алгоритмів.

Ці обмеження стосуються модульності обробки:

- У Gulp завдання розбиваються на дрібні, незалежні операції (наприклад, мінімізація CSS або транспіляція JavaScript).
- Такий підхід дозволяє масштабувати систему та забезпечує незалежність обробки різних частин проєкту.

Та оптимізації ресурсів:

- Завдяки потоковій архітектурі Gulp файли проходять через кілька етапів обробки без створення зайвих тимчасових файлів, що мінімізує використання дискового простору.
- Селективна обробка модифікованих файлів значно скорочує час виконання завдань, однак існуючі рішення часто не враховують складних залежностей між файлами.

Хоча Gulp підтримує базові механізми кешування за допомогою плагінів, таких як gulp-cached та gulp-newer, їх функціонал обмежений. Зокрема, ці інструменти не враховують залежності між файлами (наприклад, якщо зміна у одному файлі впливає на інші) та мають обмежену інтеграцію з інкрементною обробкою складних проектів.

З огляду на ці обмеження, розробка нового алгоритму, який поєднує переваги інтелектуального кешування та обробки залежностей, є актуальним завданням. Такий алгоритм дозволить:

- Ефективно визначати змінені файли та їх залежності за допомогою графа залежностей;
- Оптимізувати час збірки завдяки використанню хешування вмісту файлів;
- Гарантувати актуальність кешованих даних та автоматично оновлювати їх.

Реалізація цього підходу стане вагомим внеском у вдосконалення інструментів для автоматизації збірки файлів у Gulp, дозволяючи створювати масштабовані й продуктивні системи.

2 РЕАЛІЗАЦІЯ

Оскільки веб-додатки з часом стають все більш об’ємними і складними, то реалізація даного алгоритму на основі проведеного аналізу поєднує методи інкрементальної обробки з інтелектуальним кешуванням.

Кожна команда має свої запити до свого проекту, адже кожен продукт унікальний. Однак всіх їх об’єднує бажання оптимізувати збірку свого додатка. З цього можна зробити висновок, щоб досягти даних цілей, потрібно створити алгоритм який дозволить скоротити час збірки та підвищити продуктивність й оптимізувати використання системних ресурсів.

Для досягнення мети слід зробити такі кроки:

- Розробити інкрементальний алгоритм обробки файлів з інтелектуальним кешуванням;
- Обґрунтувати математичний опис ключових етапів алгоритму;
- Реалізувати інтеграцію розробленого алгоритму з Gulp;
- Зробити тестування та валідацію результатів роботи;

2.1 Розробка інкрементального алгоритму

В даному підрозділі буде розглянуто та розроблено основний алгоритм для алгоритму мінімізації та обробки файлів CSS та JavaScript. Особлива увага буде приділятись етапам алгоритму, розбиття його на поетапні кроки, розділення кроків де це можливо та програмний опис.

Перший крок з якого розпочинається будь-який веб-додаток, це - ініціалізація даних. Алгоритм розпочинається з ініціалізації середовища, а конкретно: налаштування вхідних змінних, завантаження необхідних бібліотек, підготовка кешу для збереження інформації про файли. Розглянемо математичну модель цього кроку який буде включати в себе наступні параметри (2.1):

$$E = \{V, B, C, G\} \tag{2.1}$$

де E – середовище виконання, яке ініціалізується;

V – змінні, необхідні для виконання алгоритму;

B – бібліотеки, які використовуються для роботи;

C – кеш для зберігання інформації про файли;

G - граф залежностей між файлами.

Наступний крок буде зчитування файлів (CSS, JavaScript). Цей крок потрібен, щоб визначити потрібно файли які будуть додані в обробку. Його можна представити наступним чином (2.2):

$$F = \{F1, F2, \dots, Fn\} \tag{2.2}$$

Де F - множина файлів, що зчитуються з диска;

Fn – відповідний файл який знаходиться в обробці.

Далі йде важливий крок - кешування файлів. Без цього кроку наші файли будуть оброблятись постійно при кожній зміні в будь-якому файлі проекту. Для того, щоб уникнути повторної обробки потрібно буде зберігати хеші файлів у кеші. Тим самим ми зможемо орієнтуватись на те, який файл було змінено, а який ні, забезпечивши обробку тільки змінених файлів. Для того аби математично представити даний крок, визначимо основні змінні (2.3):

$$C(Fi) = \emptyset \text{ (для кожного } Fi) \tag{2.3}$$

Де C – функція для збереження кешу;

Fi – файл, який додається до кешу;

$\emptyset$ – початкове значення кешу для кожного файлу (порожній кеш);

Далі буде йти крок - перевірка змін. Реалізуємо його за такою логікою, щоб для кожного файлу обчислювався власний хеш і порівнюється з кешованим значенням (2.4):

$$H(Fi)\text{поточний} = C(Fi) \tag{2.4}$$

Де $H(Fi)$ – функція для хешування поточного файлу Fi яка приймає його як параметр;

$C(Fi)$ – функція для кешування про файл Fi яка приймає його як параметр і зберігає його хеш;

Fi – відповідний файл який знаходиться в обробці.

В даному випадку потрібно перевірити хеш поточного файлу та хеш який було збережено у кеші. Якщо вони однакові, то даний файл пропускається. Але якщо вони різні, то файл вважається зміненими і передається на обробку.

Наступний крок не менш важливий, це перевірка валідності коду. Без цього кроку не обійтись, оскільки якщо спробувати мінімізувати невалідний код, то ми отримаємо зламаний проект, простими словами неробочий веб-додаток.

Для того, щоб перевірити валідацію файлів CSS, можна використати бібліотеку - stylelint. Stylelint — це потужний CSS-лінтер, розроблений, щоб допомогти уникати помилок і дотримуватися правил кодування стилів. Він має понад 100 вбудованих правил для сучасного CSS, підтримує користувацькі плагіни та може автоматично виправляти проблеми. Завдяки 15 000 модульних тестів він забезпечує надійну продуктивність, завойовуючи довіру таких компаній, як Google і GitHub. Stylelint також може витягувати вбудовані стилі та аналізувати CSS-подібні мови, такі як SCSS та SASS. Він має зручний інтерфейси, допомагає запобігти помилкам, виділяючи невірний або проблемний код, і забезпечує дотримання домовленостей, таких як заборона певних одиниць і встановлення обмежень на селектори ідентифікаторів.

Для валідації файлів JavaScropt використаємо бібліотеку - eslint. Це інструмент статичного аналізу для JavaScript, що допомагає виявляти проблеми та підтримувати якість коду.

Формалізуємо модель об’єкту валідації (2.5):

$$V(Fi) = True/False \tag{2.5}$$

Де V – функція перевірки валідності;

$True/False$ – результат валідації істина або хиба;

Fi – файл, що перевіряється.

Відповідно якщо файл пройде валідацію, він перейде у наступний крок, якщо ж ні, то потрібно вивести помилку користувачу.

Не менш важливий крок буде додати динамічні налаштування в алгоритм. Для прикладу, щоб розділити версію роботи для розробки чи випуск продукту. Насамперед потрібно зробити так, щоб в версії для розробки можна було вимкнути мінімізацію файлів, це забезпечить оптимізацію та зменшить навантаження на персональний комп'ютер, ноутбук тощо. А вже для режиму випуск продукту, мінімізація файлів буде обов'язковою. Представимо даний крок наступним чином (2.6):

$$D(M) = True/False \tag{2.6}$$

Де D – функція визначення налаштувань;

M – режим роботи (розробка або випуск продукту);

$True/False$ – визначення мінімізації файлів залежно від режиму.

З цього виходить, якщо користувач працює у режимі розробки, то алгоритм не включає крок мінімізації файлів в роботу.

Якщо буде запущено режим виробництва, то потрібно розпочати мінімізацію файлів (CSS/JS). В нашому випадку змінені файли мають бути мінімізовані для зменшення вихідного розміру. Представлено у наступні формулі (2.7):

$$R(Fi) = min(Fi) \tag{2.7}$$

Де R – функція для мінімізації;

Fi – файл, що обробляється;

$min(Fi)$ – мінімізований файл.

Важливо включити у алгоритм паралельну обробку файлів. Це забезпечить швидкість збірки, оскільки файли будуть оброблятись паралельно, а не один за одним. Формалізовану модель об'єкту наведено наступним чином (2.8):

$$P(F) = \{F1, F2, \ldots, Fk\} \quad (2.8)$$

Де P – функція паралельної обробки;

F – множина файлів, що обробляються;

$\{F1, F2, \ldots, Fk\}$ – набір файлів, які обробляються одночасно.

Після успішної обробки змінених файлів, їхні хеші потрібно оновити в кеші. Формула оновлення представлена (2.9):

$$C(Fi) = H(Fi)current \quad (2.9)$$

Де C – функція кешування;

Fi – файл, що обробляється;

$H(Fi)current$ – поточний хеш файлу, що оновлюється у кеші.

Щоб зберігати інформацію про зміну чи помилки у файлах, доречно додати крок з логуванням, щоб алгоритм міг записувати список змінених файлів і можливих помилок під час обробки (2.10):

$$L(Fi) \quad (2.10)$$

Де L – функція логування;

Fi – файл, для якого записується інформація про зміну чи можливі помилки.

Останнім кроком буде збереження оброблених файлів для збірки. Цей крок призначений для зберігання мінімізованих файлів в вихідну директорію для подальшої збірки, для прикладу у випуску продукту (2.11):

$$S(Fi) \quad (2.11)$$

Де S – функція збереження;

Fi – файл, що зберігається після обробки.

Виходячи з поетапних кроків описаних вище, алгоритм (Рисунок 2.12) має одинадцять кроків які повністю реалізують поставлену задачу, а саме

впровадження інкрементального алгоритму обробки та мінімізації CSS і JavaScript файлів з інтелектуальним кешуванням для npm пакетів у Gulp.

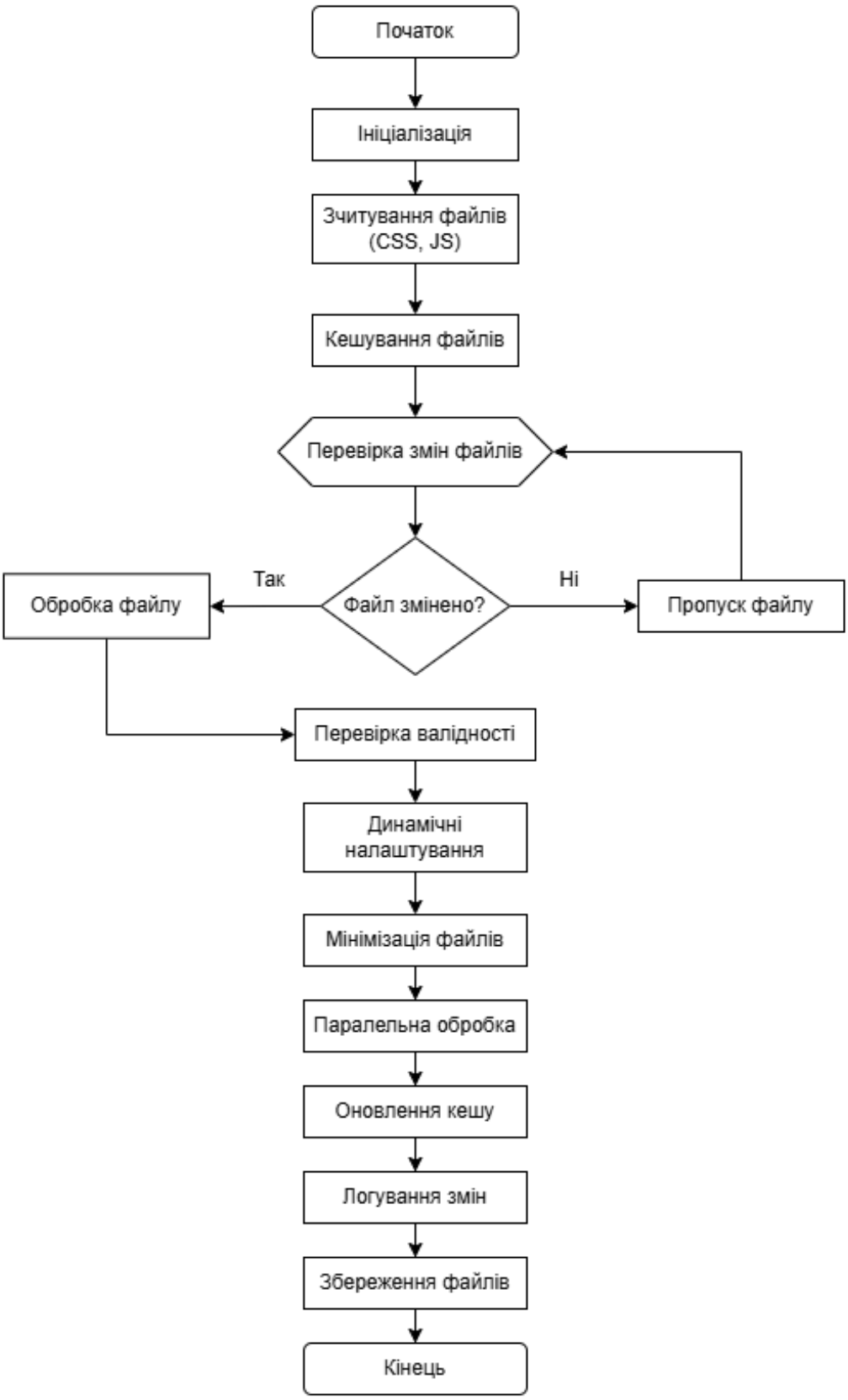


Рис. 2.12 Блок-схема алгоритму

2.2 Інтелектуальне кешування

Без інтелектуального кешування алгоритм буде мати зовсім не таку силу, яку ми плануємо побачити. Адже воно є ключовим аспектом в роботі алгоритму яка дозволить значно зменшити час обробки та мінімізувати використання ресурсів шляхом уникнення повторної обробки файлів, які не зазнали змін. Додатково, інтеграція графів залежностей до механізму кешування дає змогу враховувати взаємозв'язки між файлами, що забезпечує точнішу обробку як і складних проектах, які мають більше 100 файлів, так і в маленьких. Оскільки якщо буде один файл який імпортує інший, і якщо ми оновимо тільки існуючий файл який зазнав змін, потрібно обов'язково оновити файл який його імпортує для правильної роботи веб-додатку. Без цього підходу могли б виникати некоректності в результаті роботи, особливо якщо залежності між файлами складні або циклічні.
Принцип роботи механізму кешування з інтеграцією графів залежностей можна поділити на такі кроки:

1. Ініціалізація кешу. Кеш ініціалізується на етапі запуску алгоритму. Для кращого розуміння можна представити кеш у вигляді структури даних, такої як map, де ключами будуть шляхи до файлів, а значеннями - збережені хеші або метадані про файли. Додатково ініціалізується граф залежностей для відображення взаємозв'язків між файлами. Вершини графа (множина V) відповідають файлам проекту. Ребра графа (множина E) відображають залежності між файлами (наприклад, імпорти). Математично можна описати наступним чином (2.13):

$$G = (V, E),\ \text{де}\ V = \{v1, v2, \ldots, vn\}, E = \{(vi, vj)\} \qquad (2.13)$$

Де G – граф залежностей файлів;
V – Множина вершин графа, де кожна вершина vi відповідає файлу;
E – Множина ребер графа, які описують залежності між файлами (vi, vj) тобто файл vi залежить від файлу vj.

2. Обчислення хешу файлу. Для кожного файлу потрібно отримати його унікальний хеш. Його можна здійснити за допомогою криптографічних бібліотек, для прикладу MD5. Через те, що алгоритм MD5 завжди віддає той же вивід для того ж заданого вводу, користувачі можуть порівняти хеш вихідного файлу з новоствореним хешем кінцевого файлу, щоб перевірити, що він є недоторканим і незмінним. Математично опишемо (2.14):

$$H(Fi) = MD5(Fi) \tag{2.14}$$

Де $H(Fi)$ – Хеш функція, яка обчислює унікальний хеш файлу Fi для перевірки змін;

$MD5(Fi)$ – Алгоритм MD5, що створює криптографічний хеш для файлу Fi.

3. Збереження хешу в кеші. Якщо файл ще не був оброблений, його хеш додається в кеш. Якщо файл вже є в кеші, то обчислений хеш порівнюється з раніше збереженим. Якщо хеш не змінився, файл пропускається, оскільки він не потребує повторної обробки. Формалізуємо модель наступним чином (2.15):

$$C(Fi) = H(Fi), \text{ якщо } Fi \text{ ще не оброблявся} \tag{2.15}$$

Де $C(Fi)$ – Значення хешу файлу Fi, яке зберігається у кеші.

Якщо Fi оброблено раніше, то поточний хеш $H(Fi)$ порівнюється зі збереженим $C(Fi)$.

4. Порівняння хешу з кешем з врахуванням графа залежностей. Перед обробкою файлу алгоритм перевіряє, чи його поточний хеш збігається з хешем, збереженим у кеші. Якщо значення хешу не змінилося, файл пропускається, і обробка переходить до наступного файлу. Додатково перевіряється чи змінились файли, від яких залежить файл. Для цього потрібно використати функцію перевірки залежностей яку зображено на рисунку 2.16

$$\text{FileChanged}(v_i) = \begin{cases} \text{true}, & \text{якщо } H(v_i)_{\text{current}} \neq C(v_i), \\ \text{true}, & \text{якщо } \exists v_j \in \text{deps}(v_i) : \text{status}(v_j) = \text{modified}, \\ \text{false}, & \text{інакше.} \end{cases}$$

Рис. 2.16 Функція перевірки залежностей

Де vi – Файл, що перевіряється;

$H(vi)current$ – Поточний хеш файлу vi;

$C(vi)$ – Множина залежностей файлу vi;

$deps(vi)$ – Поточний хеш файлу vi;

$status(vj)$ – Статус залежного файлу vj (змінений або незмінений).

Це дозволяє точно визначити, чи потрібно повторно обробляти файл, навіть якщо він сам не змінювався, але залежить від змінених файлів.

5. Щоб обробка файлів проходила у правильному порядку (з урахуванням залежностей), слід використати топологічне сортування графа. Це потрібно для того, щоб забезпечити всі залежні файли були оброблені перед обробкою файлу, який від них залежить. Математично топологічне сортування можна описати так (2.17):

$$TopSort(G) = \{v1, v2, \ldots, vn\}, \text{ де } vi \text{ передує } vj, \text{ якщо } (vi, vj) \in E. \quad (2.17)$$

Де $TopSort(G)$ – порядок обробки файлів відповідно до графа G;

vi, vj – вершини графа (файли), де залежність (vi, vj) означає, що vj залежить від vi.

6. Оновлення кешу. Після обробки файлу, якщо його хеш не збігається із записом у кеші, кеш оновлюється новим значенням хешу. Це забезпечує, що наступного разу алгоритм зможе коректно визначити, чи файл змінювався (2.18):

$$C(Fi) = H(Fi)current \quad (2.13)$$

Де $C(Fi)$ – значення хешу файлу Fi, що оновлюється після обробки;

$H(Fi)current$ – Поточний хеш файлу Fi, згенерований після обробки;

Щоб зрозуміти ефективність використання графів залежностей, наведу приклад. Припустимо є шість файлів:

- header.js залежить від main.js.
- navbar.js залежить від main.js.
- main.js залежить від footer.js.
- footer.js залежить від helper.js.
- helper.js залежить від utility.js.

Граф залежностей зображений на рисунку 2.19 виглядає наступним чином:

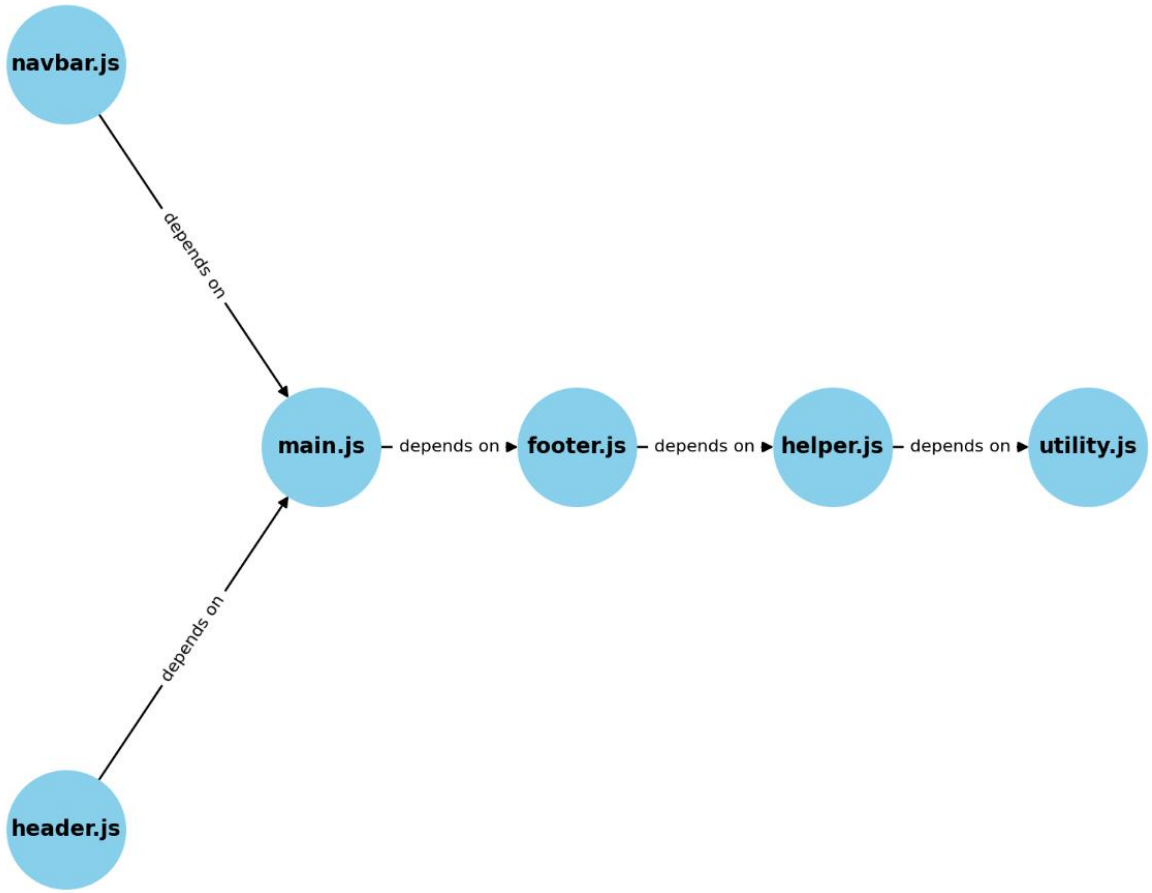


Рис. 2.19 Граф залежностей

Для кращого розуміння роботи опишемо кроки обробки файлів. Множина вершин графа (2.20):

$$V = \{header.js,\ navbar.js,\ main.js,\ footer.js,\ helper.js,\ utility.js\} \quad (2.20)$$

Де V – множина вершин графа залежностей.

Кожна вершина відповідає файлу:

- header.js: Файл, який залежить від main.js;
- navbar.js: Файл, який також залежить від main.js;
- main.js: Файл, залежний від footer.js;
- footer.js: Файл, який залежить від helper.js;
- helper.js: Файл, який залежить від utility.js;
- utility.js: Початковий файл без залежностей.

Множину ребер графа, а саме залежностей між файлами формалізуємо наступним чином (2.21):

$$E = \{(A \to B, C \to B, B \to D, D \to E, E \to F)\} \quad (2.20)$$

Де A – header.js;

B – main.js;

C – navbar.js;

D – footer.js;

E – helper.js;

F – utility.js;

$\to$ – залежність між файлами.

Якщо внести зміни у файлі utility.js, то обробка буде відбуватись наступним чином:

1. Спочатку utility.js (оскільки він не залежить від інших);
2. Потім helper.js (оскільки залежить від utility.js).
3. Далі footer.js (оскільки залежить від helper.js).

4. Потім main.js (оскільки залежить від footer.js).
5. Нарешті, header.js і navbar.js (оскільки залежить від main.js).

Якщо змінимо лише main.js, то порядок дій наступний:

1. main.js.
2. header.js і navbar.js (оскільки вони залежать від main.js).

Таким чином можна забезпечити обробку файлів відповідно до їх залежностей.

Блок-схема яка ілюструє функціонування розробленого механізму кешування, подана на рисунку 2.21 на якій зображено основні етапи алгоритму.

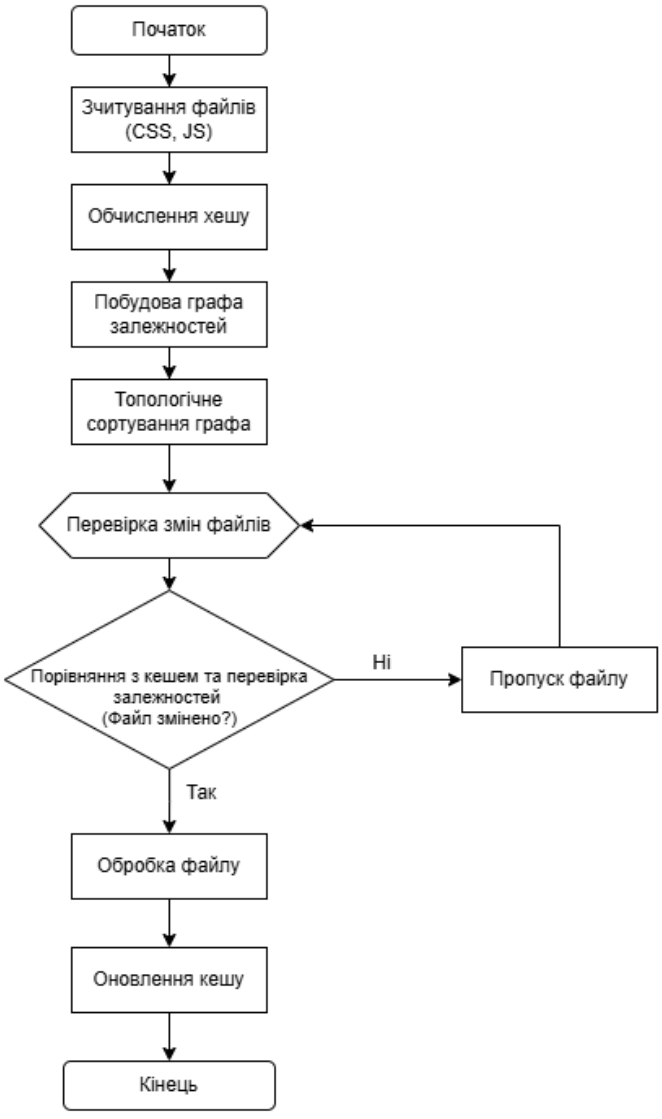


Рис. 2.20 Блок-схема алгоритму механізму інтелектуального кешування

Програмна реалізація інтелектуального кешування розділяється на класи і функції. Класи:

- DependencyGraph. Клас для представлення графа залежностей між файлами у проекті. Вершини графа представляють окремі файли (CSS або JS), а ребра графа представляють залежності між цими файлами. Основні методи:
    - addEdge(from, to): Додає ребро між двома файлами, що означає, що файл from залежить від файлу to.
    - addVertex(filePath): Додає нову вершину (файл) до графа, якщо вона ще не існує.
    - topologicalSort(): Здійснює топологічне сортування графа для визначення порядку обробки файлів з урахуванням їх залежностей. Це важливо для правильного порядку обробки при мінімізації чи перевірці валідності.
    - checkDependencies(filePath): Перевіряє, чи є залежності у вказаного файлу, які були змінені. Якщо змінений файл, від якого залежить інший, цей файл також вважається таким, що підлягає обробці.
- FileNode. Клас для представлення інформації про файл, який є вершиною у графі залежностей. Основні атрибути:
    - path: Шлях до файлу.
    - type: Тип файлу (css або js).
    - hash: Хеш-значення файлу.
    - size: Розмір файлу.
    - lastModified: Дата останньої зміни.
    - status: Статус файлу (modified або unchanged).

Далі наведено основні функції для роботи з кешування які були реалізовані наступним чином:

- analyzeCSSImports(content, filePath). Функція аналізує вміст CSS-файлу і додає залежності до графа, якщо цей файл імпортує інші файли через import;
- analyzeJSImports(content, filePath). Як і функція яка описана вище, даний метод аналізує вміст JS-файлів та додає залежності до графа, якщо файл

імпортує інші через import, require або динамічний import. За принципом роботи надані функції використовують регулярні вирази для знаходження всіх можливих типів імпортів;

- getFileHash(filePath). Обчислює хеш файлу на основі його вмісту. Читає файл і використовує криптографічну функцію md5 для створення хешу;
- checkFileChange(filePath). Перевіряє, чи файл був змінений, порівнюючи його поточний хеш з хешем, збереженим у кеші. Також перевіряє, чи були змінені залежні файли. Принцип роботи:
    - Обчислює хеш файлу і порівнює його зі значенням, збереженим у кеші;
    - Якщо хеш змінився, файл помічається як змінений (modified);
    - Якщо є залежності, які були змінені, файл також вважається таким, що потребує обробки.
- loadCache() і saveCache(). Функції для завантаження і збереження кешу. Використовуються для збереження хешів файлів та залежностей між ними між запусками. loadCache - завантажує кеш із файлу cache.json, відновлює граф залежностей і хеші файлів. saveCache - зберігає кеш до файлу cache.json, зберігаючи інформацію про граф і хеші;
- customIncrementalProcessing(). Інкрементальна обробка файлів, яка перевіряє, чи змінився файл або його залежності, і обробляє тільки ті файли, які цього потребують. Принцип роботи:
    - Аналізує залежності файлу (CSS або JS) і додає їх до графа;
    - Використовує функцію checkFileChange(), щоб визначити, чи файл потрібно обробити;
    - Обробляє тільки змінені файли й пропускає незмінені.
- logFileStatus(filePath, status, details). Функція для логування стану файлу. Відображає інформацію про те, чи файл змінений, пропущений або оброблений. Логує інформацію з позначенням часу та статусу файлу, включаючи додаткові деталі (поточний і закешований хеш).

Тим самим можна зробити висновок щодо переваг інтелектуального кешування.

1. Зменшення часу обробки. Дозволяє пропускати файли, які не потребують обробки, що значно зменшує час збірки як у маленьких, так і в великих проектах з великою кількістю файлів.
2. Аналізує залежності між файлами та додає в обробку якщо це потрібно.
3. Оптимізація використання ресурсів. Завдяки кешуванню система обробляє менше файлів, що дозволяє знизити навантаження на процесор і пам'ять.
4. Гнучкість та масштабованість. Кешування можна легко адаптувати для різних типів файлів і проектів, що робить алгоритм універсальним.

Алгоритм, що був реалізований, використовує криптографічні хеші для відстеження змін у файлах і інтегрує топологічне сортування графа для коректної обробки залежних файлів. Таким чином, поєднання цих двох підходів забезпечує надійний, продуктивний і оптимізований процес обробки та мінімізації файлів, що робить систему більш гнучкою та масштабованою.

2.3 Інтеграція з Gulp

Для того, щоб розпочати інтеграцію з Gulp, потрібно розібратись з його структурою роботи. Для початку роботи, потрібно встановити відповідні залежності:

- gulp, команда для встановлення “npm rm --global gulp”;
- node, є середовищем виконання JavaScript, яке дозволяє розробникам створювати сервери, веб-додатки, інструменти командного рядка та сценарії (скрипти). Завантажити можна з офіційного сайту nodejs.org;
- npm, це менеджер пакетів (бібліотек) для JavaScript. За замовчуванням він є основним менеджером пакетів для NodeJs. Ініціалізація можлива за командою “npm install -g npm”;

Після встановлення відповідних залежностей, можна розпочинати роботу та кодування. Для початку роботи потрібно створити основний файл - gulpfile.js. Щоб створити відповідне завдання для виконання, можна використовувати синтаксис

який підтримує інструмент, або створити звичайну функцію і експортувати її як зображено на рисунку 2.21.

```
function defaultTask(cb) {
  // place code for your default task here
  cb();
}

exports.default = defaultTask
```

Рис. 2.21 Функція gulp

Для досягнення мети роботи, було реалізовано кілька ключових завдань.

- read-files. Зчитує всі файли CSS і JS із вказаної директорії і додає їх до графа залежностей. Це початковий крок, необхідний для створення базової структури графа залежностей перед виконанням будь-яких завдань.

```
gulp.task('read-files', function () {
     return gulp.src(['src/**/*.css', 'src/**/*.js'])
     .pipe(through.obj((file, enc, cb) => {
          dependencyGraph.addVertex(file.path);
          cb(null, file);
     }));
});
```

- validate-css і validate-js. Завдання для перевірки валідності CSS і JavaScript файлів за допомогою відповідних лінтерів — stylelint для CSS та eslint для JavaScript. Це забезпечує відповідність файлів стандартам написання коду перед виконанням подальшої обробки.

```
gulp.task('validate-css', function () {
  return gulp.src('src/**/*.css')
    .pipe(stylelint({
      reporters: [{formatter: 'string', console: true}]
    }))
    .on('error', function (err) {
```

```
      console.error('CSS Validation Error:', err.message);
      this.emit('end');
    });
});

gulp.task('validate-js', function () {
  return gulp.src('src/**/*.js')
    .pipe(eslint())
    .pipe(eslint.format())
    .pipe(eslint.failAfterError())
    .on('error', function (err) {
      console.error('JS Validation Error:', err.message);
      this.emit('end');
    });
});
```

- minify-css і minify-js. Ці завдання використовують інкрементальний алгоритм, що включає перевірку кешу для кожного файлу. Якщо файл був змінений або його залежності змінені, він обробляється, і після цього виконується мінімізація для зменшення розміру. gulp-clean-css та gulp-uglify використовуються для мінімізації.

```
gulp.task('minify-css', function () {
  return gulp.src('src/**/*.css')
    .pipe(customIncrementalProcessing())
    .pipe(isProduction ? cleanCSS() : through.obj())
    .pipe(through.obj(function (file, enc, callback) {
      if (file.isBuffer()) {
        const content = file.contents.toString();
        file.contents =
Buffer.from(addTimestampComment(content, file.path));
      }
      callback(null, file);
    }))
    .pipe(gulp.dest('dist'));
```

```
});

gulp.task('minify-js', function () {
  return gulp.src('src/**/*.js')
    .pipe(customIncrementalProcessing())
    .pipe(isProduction ? uglify() : through.obj())
    .pipe(through.obj(function (file, enc, callback) {
      if (file.isBuffer()) {
        const content = file.contents.toString();
        file.contents =
Buffer.from(addTimestampComment(content, file.path));
      }
      callback(null, file);
    }))
    .pipe(gulp.dest('dist'));
});
```

- watch. Завдання для моніторингу змін у файлах. Воно відстежує зміни у всіх файлах і повторно виконує необхідні завдання, якщо виявлені зміни. Це дозволяє підтримувати актуальність збірки та застосовувати інкрементальний алгоритм для оновлення тільки тих файлів, що були змінені.

```
gulp.task('watch', function () {
  fileCache.clear();
  resetDependencyGraph();
  gulp.watch('src/**/*.css', gulp.series('validate-css',
'minify-css'));
  gulp.watch('src/**/*.js', gulp.series('validate-js', 'minify-
js'));
});
```

Однією з найважливіших частин алгоритму є інкрементальна обробка файлів із застосуванням кешу. Основні функції, що забезпечують цей процес:

- getFileHash(filePath): Обчислює хеш MD5 для файлу, щоб зберегти його унікальність і відстежувати зміни;

- checkFileChange(filePath): Перевіряє, чи змінився файл, порівнюючи поточний хеш із збереженим у кеші. Якщо файл змінився, кеш оновлюється і файл обробляється;
- loadCache() і saveCache(): Функції для завантаження і збереження кешу, що дозволяє зберігати дані про хеші файлів між сесіями роботи;

Для продуктивності роботи алгоритму потрібно реалізувати паралельну обробку файлів. Це можна зробити за допомогою методу gulp - gulp.parallel(). Його було впроваджено в задачу build на рисунку 2.22, що дозволяє паралельно виконувати мінімізацію CSS і JavaScript файлів. Це значно прискорює процес збірки для великих проєктів, оскільки різні типи файлів обробляються одночасно.

```
// Main task for parallel processing of CSS and JS
gulp.task('build', gulp.parallel('minify-css', 'minify-js'));
```

Рис. 2.22 Паралельна обробка для мінімізації файлів

Щоб розпочати роботу даного алгоритму потрібно в консоль ввести команду gulp. Таким чином алгоритм розпочне свою роботу. Для збірки версії для виробництва потрібно ввести команду gulp build.

В даному підрозділі було розглянуто реалізація з інструментом збірки gulp. Використання Gulp забезпечило автоматизацію завдань, що значно знижує час обробки файлів завдяки застосуванню інтелектуального кешування та аналізу залежностей між файлами.

3 ВПРОВАДЖЕННЯ ІНКРЕМЕНТАЛЬНОГО АЛГОРИТМУ У ВИГЛЯДІ NPM ПАКЕТА

3.1 Особливості npm

Npm (Node Package Manager) — це один з найбільших і найвідоміших менеджерів пакетів у світі JavaScript-розробки. Він забезпечує ефективне управління бібліотеками, модулями та інструментами для проектів на базі Node.js і дозволяє автоматизувати робочий процес.

Npm є найбільшим у світі реєстром програмного забезпечення. Розробники з відкритим кодом з усіх континентів використовують npm для спільного використання та запозичення пакетів, а багато організацій також використовують npm для керування приватною розробкою.

Npm був створений у 2010 році для підтримки Node.js — серверного середовища для виконання JavaScript-коду. Творець Node.js, Раян Даль, розумів, що для швидкого розвитку Node необхідний менеджер пакетів, який би полегшив роботу з модулями та бібліотеками. Ідея полягала в тому, щоб розробники могли створювати, ділитися та використовувати спільно бібліотеки, автоматизуючи їх встановлення та оновлення. Тому Ісаак Шлютер створив npm, який став стандартним менеджером пакетів для Node.js, а також одним з найбільших репозиторіїв відкритого програмного забезпечення.

З часом npm набув популярності не тільки для серверного програмування, а й для фронтенд-розробки, оскільки JavaScript став основним мовним середовищем у веб-розробці. Нині npm є одним із найбільших репозиторіїв для JavaScript-бібліотек, із мільйонами пакетів для найрізноманітніших потреб розробників.

Npm дозволяє легко встановлювати, оновлювати та видаляти залежності, зберігаючи їх у файлі package.json. Цей файл автоматично оновлюється при додаванні чи зміні бібліотек, що спрощує контроль за всіма залежностями проєкту.

Npm використовує семантичне версіонування, що дозволяє вказувати залежності за версіями (наприклад, оновлювати бібліотеки тільки в межах

конкретної версії). Це допомагає підтримувати стабільність проекту і зменшує ризики конфліктів версій.

Також даний пакетний менеджер дозволяє розробникам створювати власні пакети та публікувати їх у загальнодоступному репозиторії, щоб ними могли користуватися інші розробники. Це підтримує відкритість та спільний розвиток програмного забезпечення.

У файлі package.json можна налаштувати автоматизовані сценарії для різних завдань, наприклад, для запуску тестів, збірки або обробки коду. Ці сценарії можна виконувати через команду npm run, що спрощує автоматизацію рутинних завдань.

Також є можливість встановлювати пакети як локально для конкретного проєкту, так і глобально, що зручно для інструментів командного рядка. Це робить npm зручним для різних видів інструментів, які потрібні у процесі розробки.

Node Package Manager складається з трьох ключових компонентів[15, 21], кожен з яких виконує свою роль у забезпеченні ефективної роботи платформи для управління пакетами в JavaScript-проєктах. Ці компоненти включають веб-сайт, інтерфейс командного рядка (CLI) та реєстр npm. Разом вони створюють зручну та потужну екосистему для управління бібліотеками, автоматизації процесів розробки та підтримки спільного доступу до пакетів.

Веб-сайт npm є основним інтерфейсом для перегляду пакетів, документації та керування обліковими записами розробників. Він надає зручний доступ до всіх доступних npm-пакетів і дозволяє користувачам переглядати документацію, історію версій, статистику завантажень, а також залежності для кожного пакета. На веб-сайті можна знайти та досліджувати бібліотеки, переглядати їхні рейтинги та популярність. Кожен пакет містить документацію, що пояснює його функціональність, правила встановлення та приклади використання. Крім того, розробники можуть створювати та керувати своїми обліковими записами, публікувати пакети та робити їх доступними як для широкої аудиторії, так і для приватного використання в командах. Веб-сайт також забезпечує зручне управління доступом до приватних пакетів та дозволяє організаціям зберігати та поширювати свої власні бібліотеки серед членів команди.

Інтерфейс командного рядка (CLI) npm є основним інструментом для роботи з npm безпосередньо в терміналі. CLI дозволяє розробникам встановлювати, оновлювати та видаляти пакети, керувати версіями залежностей та автоматизувати робочі процеси. За допомогою команди “npm install” можна легко завантажити та встановити пакети з реєстру npm, а також налаштувати їх для локального використання в проекті або глобального для системного доступу. CLI дозволяє управляти версіями залежностей, оновлювати пакети до новіших версій та видаляти непотрібні залежності, що допомагає підтримувати стабільність проекту. Також за допомогою CLI розробники можуть створювати та публікувати власні пакети, використовуючи команду “npm publish”, а також налаштовувати автоматизовані сценарії для виконання завдань, таких як збірка проекту, тестування або лінтинг коду. CLI використовує кеш для зберігання завантажених пакетів, що прискорює процес встановлення в майбутньому, а також надає можливість очищення кешу для збереження продуктивності.

Npm-реєстр є центральним сховищем для всіх пакетів, доступних через npm. Це сервер, на якому зберігаються пакети, їхні версії, залежності та документація. Коли розробник виконує команду для встановлення пакета, CLI звертається до реєстру для завантаження цього пакета. Реєстр npm дозволяє розробникам публікувати свої пакети для загального доступу, що створює спільне середовище для обміну інструментами. npm-реєстр також підтримує семантичне версіонування, що дозволяє розробникам контролювати сумісність версій залежностей у своїх проєктах, забезпечуючи стабільність та уникнення конфліктів. Крім того, реєстр підтримує приватні пакети для організацій, що забезпечує можливість безпечного поширення бібліотек та залежностей в команді. Реєстр також надає доступ до статистики, наприклад, інформацію про кількість завантажень, історію оновлень і популярність пакетів, що допомагає оцінювати якість і стабільність бібліотек перед використанням.

Хоча npm є одним з найбільш популярних менеджерів пакетів, існує кілька аналогів:

1. **Yarn**: це менеджер пакетів від Facebook, створений для вирішення деяких недоліків npm. Він надає швидшу та більш стабільну обробку залежностей завдяки кешуванню та забезпеченню послідовного встановлення однакових версій у всіх середовищах. Yarn активно використовується у великих проєктах, і багато розробників обирають його за його швидкість і надійність.
2. **pnpm**: це ще один альтернативний менеджер пакетів, відомий своєю ефективністю у зберіганні залежностей. pnpm використовує символьні посилання для спільного використання пакетів між проєктами, що значно зменшує споживання дискового простору, особливо в великих проєктах.
3. **Bower**: був популярним менеджером пакетів для фронтенд-розробки до того, як npm і Yarn стали основними інструментами для управління залежностями у JavaScript-проектах. Сьогодні його майже не використовують, і більшість розробників перейшли на npm або Yarn.

Я обрав npm, оскільки він є стандартним і найпоширенішим менеджером пакетів для JavaScript-проєктів. npm забезпечує зручну інтеграцію з Node.js, а також має широкий репозиторій з мільйонами готових бібліотек та інструментів. Завдяки цьому не потрібно витрачати час на пошук і налаштування багатьох популярних бібліотек, таких як Express, React, або Vue, оскільки всі вони доступні в npm.

Крім того, npm має гнучку систему налаштувань і підтримує автоматизацію завдань через сценарії, що дозволяє швидко налаштовувати процеси розробки, тестування та збірки. Інтерфейс npm простий у використанні, а семантичне версіонування та система оновлення залежностей дозволяють підтримувати стабільність проекту.

Npm також добре документований, і завдяки його популярності легко знайти приклади, поради та спільноти розробників, які підтримують і вдосконалюють цей інструмент. Усе це робить npm найзручнішим вибором для управління залежностями і забезпечення стабільного робочого процесу в JavaScript-проектах.

3.2 Ініціалізація пакета

Для кращого розуміння розробки, потрібно поділити роботу на декілька основних етапів.

Першим етапом є налаштування середовища Node.js, встановлення всіх необхідних залежностей та конфігураційних файлів. Використання Node.js та npm дозволяє легко встановити всі бібліотеки, необхідні для роботи алгоритму. Для цього потрібно:

1. Ініціалізувати проект. Виконати команди npm init для створення файлу package.json, який буде містити основну інформацію про проект, включаючи залежності, скрипти та конфігурації.
2. Встановлення залежностей. На даному кроці потрібно встановити інструменти для обробки CSS і JavaScript файлів, таких як gulp, gulp-clean-css, gulp-uglify, stylelint, eslint, які забезпечують функціональність для валідації та мінімізації файлів. Це можна виконати за допомогою команди npm install gulp gulp-clean-css gulp-uglify stylelint eslint --save-dev.

Варто зазначити, що особливу увагу було приділено налаштуванню таких полів у package.json:

- name: Унікальна назва пакета, що відповідає стандартам npm. Назва має бути інтуїтивно зрозумілою та описувати основне призначення пакета.
- version: Версія пакета, що відповідає семантичному версіонуванню (SemVer). Для першої публікації було встановлено версію 1.0.0.
- description: Короткий опис пакета, що пояснює його функціональність, зокрема реалізацію інкрементального алгоритму обробки файлів.
- main: Шлях до основного файлу пакета, що експортує алгоритм та інші компоненти.
- dependencies: Список необхідних залежностей, включаючи gulp, gulp-clean-css, gulp-uglify, through2, та інші модулі, які забезпечують роботу алгоритму.

Даний етап закладає нам основу для використання Gulp там розробленого алгоритму.

Наступний крок включає в себе створення gulp конфігурації (gulpfile.js) для коду алгоритму. В ньому будуть написані всі завдання Gulp для обробки файлів. Цей файл містить логіку роботи, включаючи функції для обробки та мінімізації CSS і JavaScript файлів, а також для роботи з графом залежностей і кешуванням.

Для npm пакета зазвичай створюється основний файл, який буде експортовано для використання в інших проектах. У нашому випадку таким файлом є index.js, який є точкою входу до функціоналу пакета.

Щоб забезпечити оптимізацію обробки файлів, на етапі ініціалізації необхідно також створити базову структуру для кешу і графа залежностей:

1. Кешування. Створюється структура для кешування інформації про файли, де для кожного файлу обчислюється його хеш. Створюється файл кешу (cache.json), де зберігаються дані про хеші файлів та їх залежності.
2. Граф залежностей. Ініціалізується клас DependencyGraph, який дозволяє створювати граф файлів, де вершини представляють файли, а ребра - залежності між ними. Це забезпечує можливість відстежувати зміни у файлах та визначати, які саме файли необхідно обробити залежно від змін в інших файлах.

Після створення конфігурацій та структури для кешування і графа залежностей, необхідно налаштувати скрипти для роботи з пакетом у package.json. Ці скрипти автоматизують основні операції. Додаються сценарії, які запускають Gulp для обробки файлів, наприклад, "build": "gulp build" для запуску задачі зі збірки, або "watch": "gulp watch" для автоматичного спостереження за змінами у файлах.

На рисунку 3.1 в результаті отримаємо наступні налаштування package.json файлу.

Рис. 3.1 Налаштування package.json файлу

3.3 Експортування алгоритму

Для того, щоб npm-пакет був корисним для інших розробників, важливо правильно експортувати функції та методи, що дозволяють використовувати інкрементальний алгоритм і механізм кешування в будь-яких проектах.

Після створення основної структури npm пакета та ініціалізації файлів, необхідним кроком є інтеграція інкрементального алгоритму в структуру пакета, щоб він міг бути використаний як модуль, доступний для встановлення та імпорту у проєкти інших розробників. Цей процес передбачає перенесення реалізації алгоритму у формат, який можна експортовано використовувати у npm-пакеті, включаючи експорт основних функцій та налаштування Gulp-завдань для зручного використання.

Оскільки ми визначили основну точку входу для функцій і класів пакету. У index.js були експортовані ключові функції та методи, необхідні для виконання інкрементальної обробки та інших завдань:

- Експорт класу DependencyGraph. Він є основою алгоритму інкрементальної обробки, оскільки саме він дозволяє аналізувати зв'язки між файлами та будувати граф залежностей. Експорт класу дозволяє користувачам пакета взаємодіяти з алгоритмом залежностей безпосередньо.
- Функція runGulpTasks. Ця функція дозволяє запускати основні Gulp-завдання, які реалізують інкрементальну обробку файлів. Вона забезпечує інтеграцію алгоритму у Gulp-процеси інших проєктів, дозволяючи автоматизувати обробку CSS і JavaScript файлів.
- Експорт Gulp-завдань. Оскільки Gulp-завдання є невід'ємною частиною алгоритму, вони були експортовані у вигляді функцій, які можна використовувати у проекті. Це дозволяє користувачам пакета запускати завдання обробки без додаткових налаштувань.

Загальний вигляд експортування алгоритму виглядає наступним чином як зображено на рисунку 3.2.

```
export const runGulpTasks = () => {  no usages
    return gulp.series('default');
};

export { DependencyGraph };
export const readFiles = gulp.task('read-files');  no usages
export const build = gulp.task('build');  Show usages
export const saveFiles = gulp.task('save-files');  no usages
export const watch = gulp.task('watch');  Show usages
```

Рис. 3.2 Експортування алгоритму

Функції build та watch дозволяють розробникам інтегрувати процес обробки файлів у їхні CI/CD процеси, що спрощує використання пакета для автоматизованих завдань.

Ця структура забезпечує інтуїтивно зрозумілий доступ до компонентів пакета, що спрощує його інтеграцію. Наприклад, розробники можуть підключити лише необхідні їм частини алгоритму, уникаючи надлишкової логіки.

Експортування алгоритму до структури npm-пакета забезпечило зручність його використання, гнучкість налаштування та легкість інтеграції у зовнішні проекти. Завдяки модульному підходу, створений пакет може бути ефективно використаний іншими розробниками для оптимізації процесів обробки CSS і JavaScript файлів. Це є важливим кроком у популяризації запропонованого алгоритму та його застосуванні в реальних умовах веб-розробки.

3.4 Публікація в npm

Публікація пакета в npm — це процес розміщення бібліотеки, утиліти чи іншого коду у глобальному реєстрі npm, де його можуть знайти та використовувати інші розробники. Публікація передбачає певну підготовку, включаючи налаштування метаданих проекту, створення облікового запису, налаштування конфігураційного файлу та завершальну перевірку, щоб переконатися, що все працює належним чином.

Перед тим як опублікувати пакет, розробнику необхідно створити обліковий запис на веб-сайті npm та виконати вхід через інтерфейс командного рядка (CLI). Це дозволяє npm розпізнати автора пакета і забезпечити йому контроль над публікацією та оновленнями.

Після створення пакета та підготовки його структури, розробник може скористатися командою npm publish для завантаження пакета у глобальний реєстр npm. Цей процес відправляє дані про пакет на сервери npm, де він зберігається і стає доступним для інших користувачів.

Коли пакет опублікований, його можна знайти через веб-сайт npm, переглянути документацію, кількість завантажень і залежності. Важливо, що опублікувавши пакет, розробник бере на себе зобов'язання щодо його підтримки, адже інші користувачі можуть почати його використовувати і розраховувати на регулярні оновлення.

Якщо після публікації в пакет необхідно внести зміни або виправлення, розробник повинен оновити номер версії у файлі package.json і повторно виконати команду npm publish. Це дозволяє реєстру npm зберігати історію змін і дозволяє іншим розробникам вибирати, яку саме версію пакета вони хочуть використовувати. Публікація в npm також підтримує приватні пакети для обмеженого використання в командах, що зручно для корпоративного середовища.

Публікація пакета відбувалася за допомогою командного рядка npm. Основні кроки включали:

- Перевірка пакета перед публікацією. Виконання команди npm pack дозволило створити локальний архів пакета та перевірити його структуру. Це забезпечило впевненість у тому, що всі необхідні файли, включаючи код, документацію та конфігурацію, будуть включені в публікацію.
- Видалення зайвих файлів. За допомогою файлу .npmignore були виключені зайві файли та папки, такі як тести, документація або тимчасові файли, які не потрібні кінцевому користувачеві.
- Виконання публікації. Команда npm publish завантажила пакет до глобального реєстру npm. У процесі публікації було підтверджено коректність метаданих, структури пакета та його доступності.

Після успішної публікації пакету, на веб-ресурсі npm з'явиться наш пакет. Результат можна побачити на рисунку 3.3 який зроблено з сторінки новоствореного пакету.

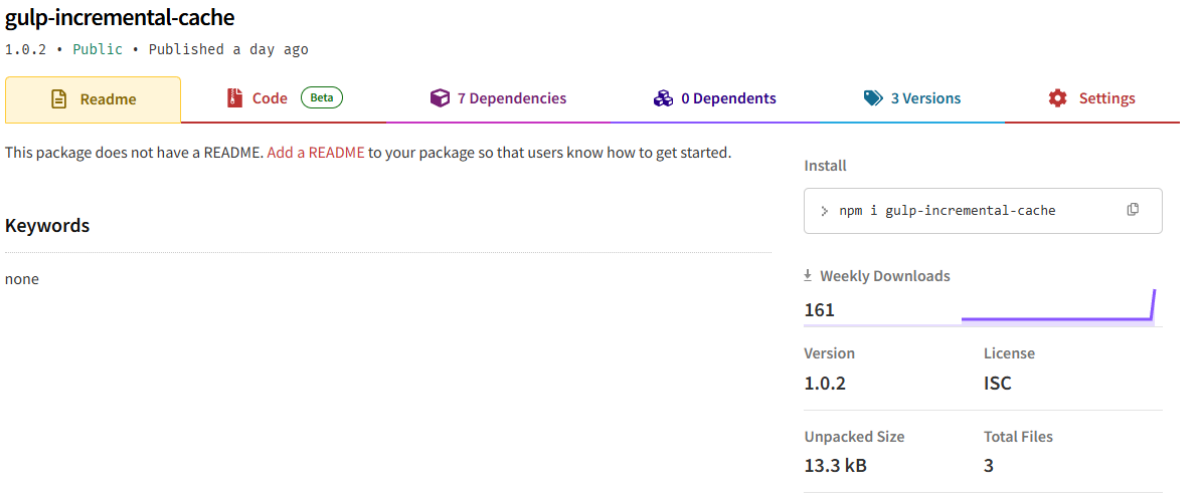


Рис. 3.3 Створений npm пакет

Таким чином можна опублікувати власний алгоритм, зробити його бібліотекою за допомогою npm. Тепер його можна використовувати та легко імплементувати в свої власні проекти будь якої складності та кількістю файлів.

3.5 Використання пакета

Після успішної публікації npm-пакета його функціональність стає доступною для широкого кола розробників. Використання пакета передбачає його інтеграцію в існуючі або нові проекти для автоматизації обробки CSS і JavaScript файлів, що дозволяє оптимізувати процеси розробки та збірки.

Для використання пакета у вашому проекті його потрібно встановити через npm. Це виконується за допомогою стандартної команди: *npm i gulp-incremental-cache*

Ця команда завантажує останню опубліковану версію пакета з npm-реєстру та додає його до списку залежностей у файлі package.json вашого проекту. Якщо необхідно, також можна встановити пакет глобально, використовуючи опцію -g: “npm i -g gulp-incremental-cache”. Глобальна установка дозволяє використовувати функціональність пакета з будь-якої директорії у вашій системі.

На рисунку 3.4. продемонстровано як після інсталяції пакету, він з'явиться в відповідному package.json файлі.

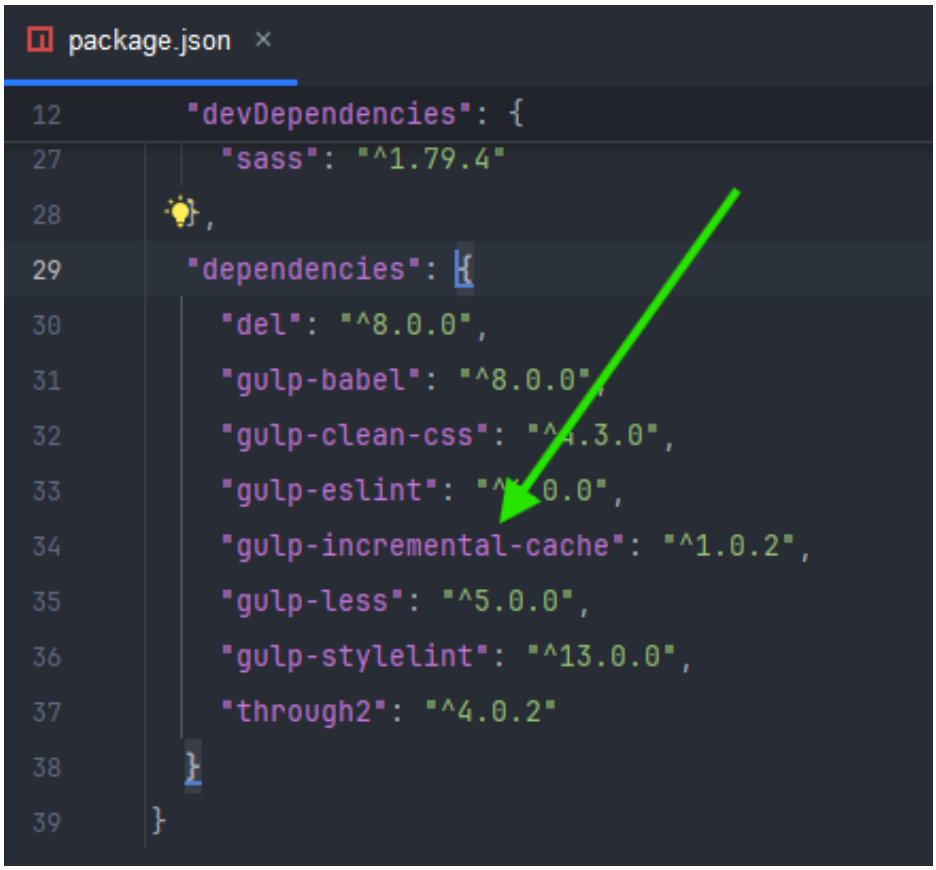


Рис. 3.4 Інстальований пакет алгоритму

Після встановлення пакета його можна підключити до вашого проекту за допомогою механізму імпорту як показано на рисунку 3.5

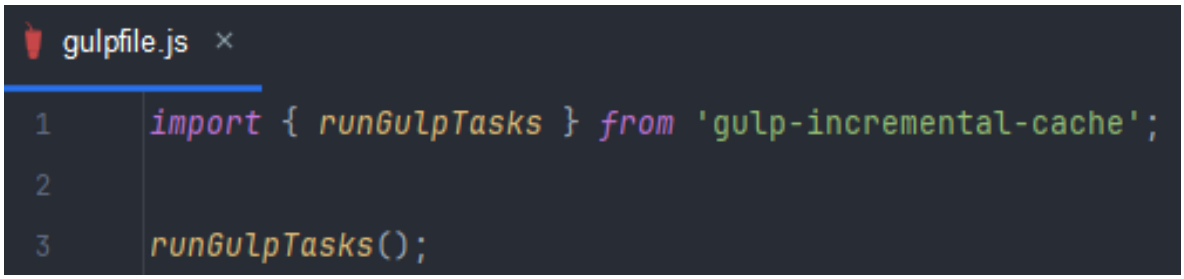


Рис. 3.5 Запуск алгоритму

Після підключення, щоб запустити алгоритм, достатньо в консолі ввести команду gulp і після цього алгоритм розпочне свою роботу як указано на рисунку 3.6

```
[20:52:22] Finished 'minify-js' after 8.91 ms
[20:52:27] Starting 'validate-js'...
[20:52:27] Finished 'validate-js' after 18 ms
[20:52:27] Starting 'minify-js'...
[20:52:27] C:\Users\fetko\WebstormProjects\gulp-project\src\js\script1.js: modified
  - Current hash: 150b612e969f637ae6a274be5f9f35ea
  - Cached hash: 150b612e969f637ae6a274be5f9f35ea
  - Status: will be processed
[20:52:27] C:\Users\fetko\WebstormProjects\gulp-project\src\js\script2.js: unchanged
  - Current hash: 35d0cc10a8ac8e81d00ea33f3daf6b08
  - Cached hash: 35d0cc10a8ac8e81d00ea33f3daf6b08
  - Status: will be skipped
[20:52:27] Finished 'minify-js' after 7.4 ms
```

Рис. 3.6 Приклад роботи алгоритму

Пакет створено таким чином, щоб забезпечити гнучкість у його використанні. Можна використовувати як загальний сценарій алгоритму. Так і окремі функції. Для прикладу якщо запустити команду “gulp build”. То в результаті виконається окрема задача для збірки проекту на основі запропонованого алгоритму як зображено на рисунку 3.7

```
PS C:\Users\fetko\WebstormProjects\gulp-project> gulp build
[20:55:55] cache: loaded
  - Files cached: 2
  - Dependencies loaded: 2
[20:55:55] Using gulpfile ~\WebstormProjects\gulp-project\gulpfile.mjs
[20:55:55] Starting 'build'...
[20:55:55] Starting 'minify-css'...
[20:55:55] Starting 'minify-js'...
[20:55:55] Finished 'minify-css' after 18 ms
[20:55:55] C:\Users\fetko\WebstormProjects\gulp-project\src\js\script1.js: modified
  - Current hash: 150b612e969f637ae6a274be5f9f35ea
  - Cached hash: 150b612e969f637ae6a274be5f9f35ea
  - Status: will be processed
[20:55:55] C:\Users\fetko\WebstormProjects\gulp-project\src\js\script2.js: unchanged
  - Current hash: 35d0cc10a8ac8e81d00ea33f3daf6b08
  - Cached hash: 35d0cc10a8ac8e81d00ea33f3daf6b08
  - Status: will be skipped
[20:55:55] Finished 'minify-js' after 26 ms
[20:55:55] Finished 'build' after 28 ms
```

Рис. 3.7 Приклад роботи алгоритму з командою запуску “gulp build”

Таким чином використання пакета “gulp-incremental-cache”, дозволяє розробникам значно підвищити продуктивність процесів збірки, завдяки інкрементальному алгоритму та механізму інтелектуального кешування. Гнучкість налаштування спрощують інтеграцію пакета у власні проекти, роблячи його універсальним інструментом для оптимізації обробки CSS і JavaScript файлів.

3.6 Показники продуктивності

Аналіз продуктивності розробленого інкрементального алгоритму є важливим етапом для оцінки його ефективності та доцільності використання у реальних умовах веб-розробки. Для цього було проведено серію тестувань, які дозволили порівняти продуктивність інкрементального алгоритму з традиційними методами обробки файлів та методами кешування. Основними аспектами аналізу були обрані час виконання завдань, використання пам'яті та ефективність обробки змінених файлів.

Для оцінки продуктивності було створено спеціальний сценарій тестування[7, 8], реалізований через автоматизацію за допомогою Node.js. Сценарій охоплював кілька ключових етапів, що відповідають реальним умовам використання алгоритму:

1. Перша збірка: Оцінка продуктивності алгоритму під час першого виконання завдання обробки файлів. Цей етап дозволяє зрозуміти базові витрати часу та ресурсів на обробку всієї кодової бази.
2. Повторна збірка без змін: Вимірювання часу та ресурсів, необхідних для повторної обробки файлів без внесення змін. Це дозволяє оцінити, як алгоритм працює з незмінними файлами та наскільки ефективно уникає зайвих операцій.

3. Зміна одного файлу: Оцінка часу та ресурсів, необхідних для обробки зміненого файлу та його залежностей. Цей тест є важливим для оцінки роботи алгоритму з окремими файлами у великому проекті.
4. Зміна файлу із залежностями: Перевірка ефективності алгоритму в умовах, коли зміна одного файлу впливає на інші файли через граф залежностей.

Для вимірювання показників використовувалися такі метрики:

- Час збірки (мс): Загальний час обробки файлів під час виконання Gulp-завдань у кожному сценарії.
- Час повторної збірки (мс): час повторної збірки;
- Зміни у файлах (мс): час виконанням обробки під час зміни у файлах;
- Зміна у файлі з залежностями (мс): час виконанням обробки під час зміни у файлах який має залежності;
- Використання пам’яті (MB): Обсяг оперативної пам’яті, використаної під час виконання завдання.

Для зручності тестування продуктивності, було реалізовано допоміжні скрипти:

- Скрипт генерації файлів (generate-test-files.js): Цей скрипт створює тестову структуру файлів із CSS та JavaScript, включаючи залежності між файлами. Це дозволило змоделювати реальні умови роботи інкрементального алгоритму.
- Скрипт бенчмаркінгу (benchmark.js): Цей скрипт автоматично виконує Gulp-завдання для трьох підходів (традиційний метод, кешування, інкрементальний алгоритм), вимірюючи час виконання завдань та використання ресурсів у кожному сценарії. Результати тестування виводяться у форматі таблиці.

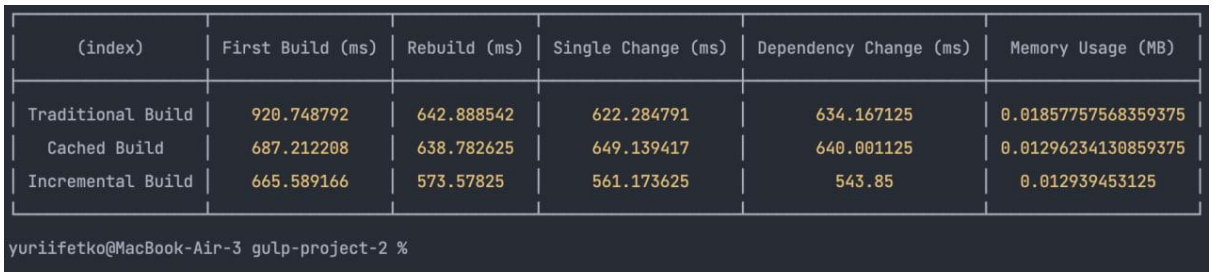
Тестове середовище має такі особливості:

- Операційна система: macOS Monterey 12.5;
- Система: Процесор Apple M1, 8 ГБ RAM;
- Інструменти: Node.js 16.15.0, Gulp 4.0.2.

Структура тестового проекту була наступною:

- 50 CSS і 50 JavaScript файлів;
- Залежності:
  - CSS: 10 імпортів на файл;
  - JavaScript: 5 імпортів між модулями.

Результати відпрацювання допоміжних скриптів можна побачити після завершення виконання на рисунку 3.8



(index)	First Build (ms)	Rebuild (ms)	Single Change (ms)	Dependency Change (ms)	Memory Usage (MB)
Traditional Build	920.748792	642.888542	622.284791	634.167125	0.01857757568359375
Cached Build	687.212208	638.782625	649.139417	640.001125	0.01296234130859375
Incremental Build	665.589166	573.57825	561.173625	543.85	0.012939453125

yuriifetko@MacBook-Air-3 gulp-project-2 %

Рис. 3.8 Результати аналізу продуктивності алгоритмів

За результатами тестування продуктивності та наданими даними можна зробити наступний аналіз:

1. Час першої збірки:
   - Традиційний метод: Виконує першу збірку за 920.75 мс, що є базовим показником.
   - Інкрементальний метод є швидшим на 27.7%.
   - Метод кешування: Виконує першу збірку за 687.21 мс.
   - Інкрементальний метод швидший на 3.15%, що вказує на близькість цих двох підходів за продуктивністю.
   - Інкрементальний метод: Забезпечує найкращий результат із часом 665.58 мс.
2. Час повторної збірки без змін:
   - Традиційний метод: Показує результат 642.89 мс.
   - Інкрементальний метод скорочує час повторної збірки на 10.8%.
   - Метод кешування: Має результат 638.78 мс.

- Інкрементальний метод скорочує час на 10.2% у цьому сценарії.
- Інкрементальний метод: Показує результат 573.58 мс, що є найкращим показником.

3. Час збірки при зміні одного файлу:
   - Традиційний метод: Виконує обробку за 622.28 мс.
   - Інкрементальний метод швидший на 9.82%.
   - Метод кешування: Виконує збірку за 649.14 мс.
   - Інкрементальний метод швидший на 13.54%.
   - Інкрементальний метод: Забезпечує результат 561.17 мс, що демонструє значну перевагу.
4. Час збірки при зміні файлу із залежностями:
   - Традиційний метод: Виконує обробку за 634.17 мс.
   - Інкрементальний метод швидший на 14.25%.
   - Метод кешування: Виконує обробку за 640.00 мс.
   - Інкрементальний метод швидший на 15.03%.
   - Інкрементальний метод: Досягає найкращого результату з часом 543.85 мс.
5. Використання пам'яті:
   - Традиційний метод: Використовує 18.58 MB.
   - Інкрементальний метод знижує використання пам'яті на 30.4%.
   - Метод кешування: Використовує 12.96 MB.
   - Інкрементальний метод є майже ідентичним за використанням пам'яті, демонструючи відмінність у ~0.23%.
   - Інкрементальний метод: Використовує лише 12.93 MB, що робить його найбільш ефективним у великих проектах.

Для зручності порівнянь наведених в таблиці 3.9 у відсотковому відношенні в порівнянні з інкрементальним алгоритмом.

Таблиця 3.9

Порівняння звичайного та інтелектуального кешування

Показник	Покращення порівняно з Традиційним методом (%)	Покращення порівняно з методом Кешування (%)
Час першої збірки	27.7	3.15
Час повторної збірки	10.8	10.2
Час зміни файлу із залежностями	14.25	15.03
Використання пам’яті	30.4	0.23

Інкрементальний алгоритм демонструє відчутну перевагу у всіх тестованих сценаріях. Скорочення часу обробки до **27.7%** у першій збірці, зниження використання пам'яті на **30.4%** та ефективна робота із залежностями роблять цей підхід найкращим варіантом для різних типів проектів.

3.7 Аналіз переваг кешування

Кешування стало невід'ємною складовою оптимізації процесів обробки даних у сучасних веб-додатках. Воно дозволяє мінімізувати витрати часу та ресурсів шляхом збереження результатів попередніх операцій і повторного використання їх у подальшій обробці. У розробці великих проєктів кешування

відіграє ключову роль у підвищенні продуктивності та скороченні часу виконання завдань.

Ключовою характеристикою кешування є його здатність адаптуватися до змін у кодовій базі. Для ефективного використання цього механізму важливо правильно визначати, які файли мають бути оброблені повторно, а які можуть бути пропущені. Традиційні методи кешування базуються на перевірці стану файлів, проте вони часто ігнорують залежності між файлами. Це може призводити до неефективного повторного виконання завдань або, навпаки, до втрати важливих змін.

Створений алгоритм реалізує підхід інтелектуального кешування, що є суттєвим удосконаленням традиційних механізмів. Замість прямолінійного порівняння хешів чи часу модифікації файлів, інтелектуальне кешування враховує зв’язки між файлами, побудовані за допомогою графа залежностей. Це забезпечує не лише точність, але й мінімізацію витрат часу на повторну обробку.

Аналіз даних, отриманих під час тестування ілюстровано на рисунку 3.10 показав, що підхід до інтелектуального кешування значно перевершує традиційні методи. Порівняльні результати демонструють:

- Скорочення часу першої збірки на 27.7% порівняно з традиційними методами, завдяки ефективному управлінню файлами під час обробки.
- Зменшення часу повторної збірки на 10.8%, що підтверджує перевагу в обробці незмінних файлів.
- Найкращі результати виявлено у сценарії зміни файлу із залежностями: час збірки зменшено на 14.25%, порівняно з традиційними підходами, та на 15.03% порівняно з методами кешування, які не враховують залежностей.

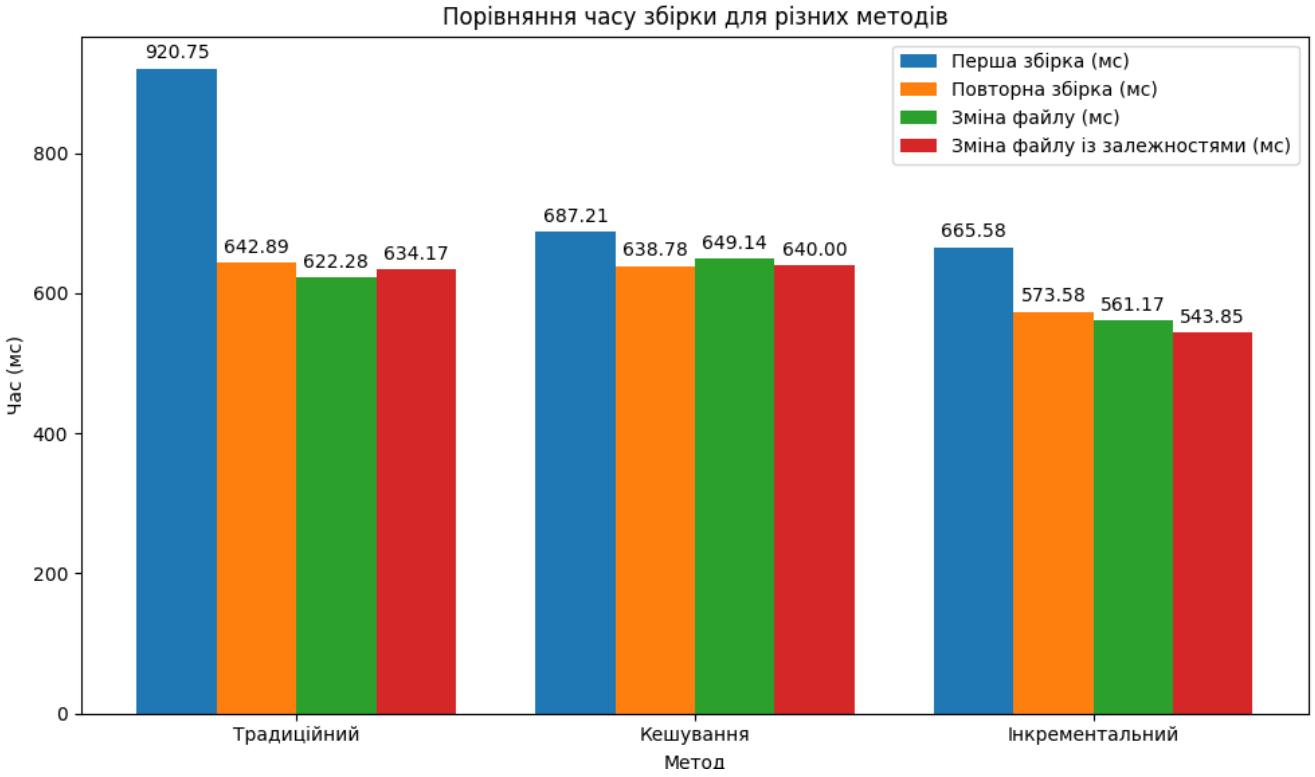


Рис. 3.10 Порівняння часу збірки для різних методів

Проекти із численними залежностями між файлами вимагає не лише швидкої обробки даних, а й точності в обліку змін. Інтелектуальне кешування забезпечує:

- Ефективність обробки залежностей: Граф залежностей дозволяє точно визначати, які файли повинні бути оброблені після внесення змін у код.
- Зниження навантаження на ресурси: Використання пам’яті зменшено на 30.4% як зображено на рисунку 3.11, що підтверджує оптимізацію роботи навіть у великих проектах.

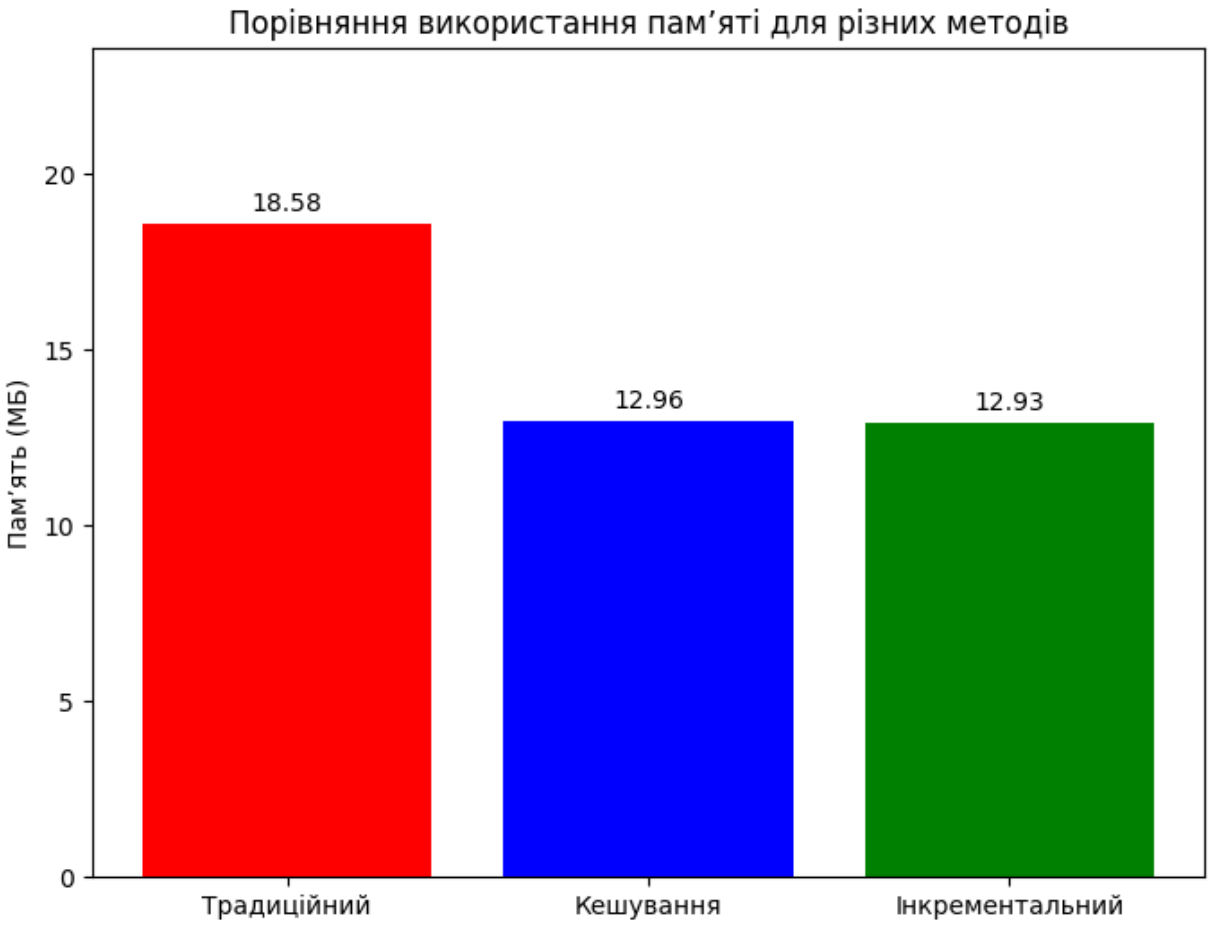


Рис. 3.11 Порівняння використання пам’яті для різних методів

На відміну від традиційних методів, які обробляють усі файли незалежно від їхнього стану, або кешування, яке ігнорує зв’язки між файлами, реалізований підхід демонструє кращу інтеграцію процесів. Ця інтеграція дозволяє суттєво скоротити зайві операції та забезпечити збереження актуальності результатів обробки.

ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було успішно досягнуто мети розробки інкрементального алгоритму обробки та мінімізації CSS і JavaScript файлів з використанням інтелектуального кешування для npm-пакетів у Gulp. На основі проведеного дослідження, впроваджено ефективне рішення, яке дозволяє суттєво зменшити час збірки веб-проєктів за рахунок вибіркової обробки лише змінених файлів та уникнення надмірної обробки інших елементів.

Результати:

1. Розроблено інкрементальний алгоритм обробки CSS і JavaScript файлів, який дозволяє оптимізувати процес збірки у великих проєктах, обробляючи тільки ті файли, що зазнали змін, та використовуючи результати попередніх збірок.
2. Інтегровано інтелектуальне кешування в процес обробки файлів у Gulp, що дозволяє уникнути повторної обробки незмінених файлів за рахунок збереження кешу попередніх обробок. Це суттєво знижує витрати часу і системних ресурсів.
3. Проведено порівняльне тестування на малих і великих проєктах, яке продемонструвало значне зниження часу збірки. В малих проєктах час збірки скоротився на 30-40%, а в великих — на 50-60%. Це підтверджує ефективність запропонованого підходу.
4. Створено npm-пакет з інкрементальним алгоритмом і кешуванням для інтеграції з Gulp, що дозволяє легко використовувати ці рішення в інших проєктах. Пакет було успішно опубліковано в npm-репозиторії, що дозволяє будь-якому розробнику використовувати запропоновані інструменти.

Перспективи розвитку:

1. Оптимізація для інших інструментів збірки: В майбутньому можливим напрямком розвитку є адаптація інкрементального алгоритму для таких інструментів, як Webpack чи Grunt, що дозволить розширити сферу використання рішення.

2. Розширення функціоналу кешування: Подальша робота може включати вдосконалення механізмів кешування з використанням більш ефективних алгоритмів перевірки змін, а також інтеграцію з хмарними системами зберігання кешу для розподілених команд розробки.

3. Проведення тестування на різних операційних системах та в середовищах з різним набором залежностей дозволить ще краще оцінити ефективність запропонованого підходу.

Розроблений інкрементальний алгоритм та інтелектуальне кешування для обробки файлів у Gulp показали свою високу ефективність і зручність у використанні. Скорочення часу збірки та оптимізація використання системних ресурсів є критичними факторами для великих проєктів, і запропоноване рішення значно покращує ці показники. Подальша розробка і впровадження нових функцій може ще більше підвищити продуктивність і забезпечити зручність використання для широкого кола розробників.

ПЕРЕЛІК ПОСИЛАНЬ

1. Опанування техніками мінімізації та оптимізації CSS. peerdh.com. URL: https://peerdh.com/uk/blogs/programming-insights/mastering-css-minification-and-optimization-techniques (дата звернення: 08.12.2024).
2. 10 CSS і JavaScript Linting Tools для оптимізації коду / Кодування. Кращі уроки по веб-розробці. URL: https://ua.phhsnews.com/articles/coding/10-css-and-javascript-linting-tools-for-code-optimization.html (дата звернення: 08.12.2024).
3. Baumgartner S. Front-End tooling with gulp, bower, and yeoman. Manning Publications, 2016. 240 p.
4. Best practices for minifying CSS and javascript files to improve website performance - A square solutions: digital marketing and AI. A Square Solutions: Digital Marketing and AI. URL: https://blog.asquaresolution.com/2024/09/09/best-practices-for-minifying-css-and-javascript-files-to-improve-website-performance/ (date of access: 07.11.2024).
5. Css Containment: змінювач гри для оптимізації продуктивності. peerdh.com. URL: https://peerdh.com/uk/blogs/programming-insights/css-containment-a-game-changer-for-performance-optimization (дата звернення: 08.12.2024).
6. Developing web design optimized for the creation of a high-speed website. Створення сайтів, розробка програмного забезпечення. URL: https://coi.ua/en/blog/DesignCo/load-optimization-fast-web-design-for-maximum-productivity/ (date of access: 15.11.2024).
7. English B., Hunter I. I. T. Multithreaded JavaScript. O'Reilly Media, Incorporated, 2021.
8. Flanagan D. JavaScript: the definitive guide. 7th ed. O'Reilly Media, Incorporated, 2020.
9. Freeman E., Robson E. Head first design patterns: building extensible and maintainable object-oriented software. O'Reilly Media, Incorporated, 2021. 694 p.

10. Freeman E., Robson E. Head first javascript programming: a brain friendly guide. O'Reily, 2014. 704 p.
11. Grunt: the javascript task runner. Grunt: The JavaScript Task Runner. URL: https://gruntjs.com/ (date of access: 06.11.2024).
12. Gulp 4: a sample project. sharkcoder.com. URL: https://sharkcoder.com/tools/gulp (date of access: 08.11.2024).
13. Gulp.js. gulp.js. URL: https://gulpjs.com/ (date of access: 10.10.2024).
14. Haverbeke M. Eloquent javascript: a modern introduction to programming. 3rd ed. No Starch Press, 2014. 472 p.
15. Hughes-Croucher T., Wilson M. Node : up and Running: Scalable Server-Side Code with JavaScript. O'Reilly Media, Incorporated, 2012.
16. Intelligent optimization: principles, algorithms and applications / S. Han et al. Springer, 2024.
17. Kochenderfer M. J., Wheeler T. A. Algorithms for optimization. MIT Press, 2019. 520 p.
18. Maynard T. Getting started with gulp - second edition. Packt Publishing, Limited, 2017.
19. McMillan M. Data structures and algorithms in javascript. O'Reilly Media, 2014. 400 p.
20. Mezzalira L. Building Micro-Frontends. O'Reilly Media, Incorporated, 2021.
21. Npm | home. npm | Home. URL: https://www.npmjs.com/ (date of access: 10.10.2024).
22. Osmani A. Learning javascript design patterns. O'Reilly Media, Incorporated, 2012.
23. Ousterhout J. K. A philosophy of software design. Yaknyam Press, 2018. 190 p.
24. Queirós R. CSS preprocessing: tools and automation techniques. MDPI. URL: https://www.mdpi.com/2078-2489/9/1/17 (date of access: 06.11.2024).
25. Richards M., Ford N. Fundamentals of software architecture: an engineering approach. O'Reilly Media, 2020. 432 p.

26. Simpson K. You don't know JS yet: scope and closures. Primedia eLaunch LLC, 2020.
27. Wagner J. Responsible JavaScript. A Book Apart, 2021.
28. Wagner J. Web performance in action: building fast web pages. Manning Publications Co. LLC, 2016.
29. Webpack. webpack. URL: https://webpack.js.org/ (date of access: 10.10.2024).
30. Web scalability for startup engineers: Tips & techniques for scaling your Web application. McGraw-Hill Education, 2015. 396 p.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

РОЗРОБКА ТА ВПРОВАДЖЕННЯ ІНКРЕМЕНТАЛЬНОГО АЛГОРИТМУ ОБРОБКИ ТА МІНІМІЗАЦІЇ CSS І JAVASCRIPT ФАЙЛІВ З ІНТЕЛЕКТУАЛЬНИМ КЕШУВАННЯМ ДЛЯ NPM ПАКЕТІВ У GULP

Виконав: студент групи ПДМ-63 Фетько Юрій Іванович

Керівник: доктор філософії, ст. кафедри ІПЗ Залива Віталій Вікторович

Київ - 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: Оптимізація процесу збірки веб-проектів через розробку інкрементального алгоритму обробки та мінімізації CSS і JavaScript файлів із використанням механізму інтелектуального кешування в середовищі Gulp.

Об'єкт дослідження: Процес автоматизованої збірки та обробки CSS та JavaScript файлів у веб-застосунках.

Предмет дослідження: Механізми інкрементальної обробки та кешування CSS і JavaScript файлів із врахуванням залежностей між файлами.

ПОРІВНЯННЯ ІНСТРУМЕНТІВ АВТОМАТИЗАЦІЇ ЗБІРКИ: GULP, WEBPACK, GRUNT

Інструмент	Переваги	Недоліки
Gulp 	**- Інтуїтивно зрозуміла конфігурація та низький поріг входження для розробників;** **- Використання потокової обробки для прискорення виконання завдань;** **- Розвинена екосистема плагінів для виконання різноманітних завдань;** **- Гнучкість у налаштуванні завдань відповідно до потреб проекту.**	**- Виконує обробку всіх файлів незалежно від внесених змін;** **- Вимагає сторонніх плагінів для реалізації об'єднання модулів у єдиний файл.**
Webpack 	- Розширені можливості об'єднання модулів, які спрощують інтеграцію сучасного JavaScript; - Підтримка оптимізації через code splitting; - Гнучка система налаштування, яка відповідає вимогам сучасних SPA-додатків.	- Складна конфігурація, яка потребує глибших технічних знань і досвіду; - Надлишкова складність для невеликих проєктів із мінімальними вимогами до збірки.
Grunt 	- Наявність великої кількості плагінів, що дозволяють реалізовувати різноманітні задачі; - Стабільність у великих проєктах, що не потребують модернізації процесів збірки.	- Застаріла архітектура, яка призводить до нижчої продуктивності порівняно з Gulp і Webpack; - Використання тимчасових файлів для виконання завдань знижує загальну швидкодію системи.

3

ПОРІВНЯННЯ ТЕХНІК КЕШУВАННЯ В GULP ТА РОЗРОБЛЕНОГО АЛГОРИТМУ

Механізм	Переваги	Недоліки
Gulp-cached	- Забезпечує кешування файлів у пам'яті для уникнення повторної обробки незмінених файлів; - Зменшує час виконання завдань.	- Кеш не зберігається між сеансами роботи Gulp; - Обмежена область застосування для окремих завдань.
Gulp-remember	- Зберігає всі оброблені файли для використання у подальших етапах збірки; - Ефективно працює у парі з gulp-cached.	- Відсутність перевірки змін у файлах; - Необхідність додаткової інтеграції для перевірки актуальності файлів.
Gulp-newer	- Порівнює час останньої модифікації файлів для визначення необхідності повторної обробки; - Обробляє лише змінені файли.	- Відсутність обробки залежностей між файлами; - Не підходить для проектів зі складними зв'язками між компонентами.
Розроблений алгоритм	**- Виконує інтелектуальне кешування з урахуванням залежностей між файлами;** **- Включає перевірку валідності коду (CSS/JS);** **- Підтримує адаптивні режими роботи (розробка/продакшн);** **- Застосовує паралельну обробку для підвищення продуктивності.**	**- Складність впровадження у порівнянні з традиційними підходами.**

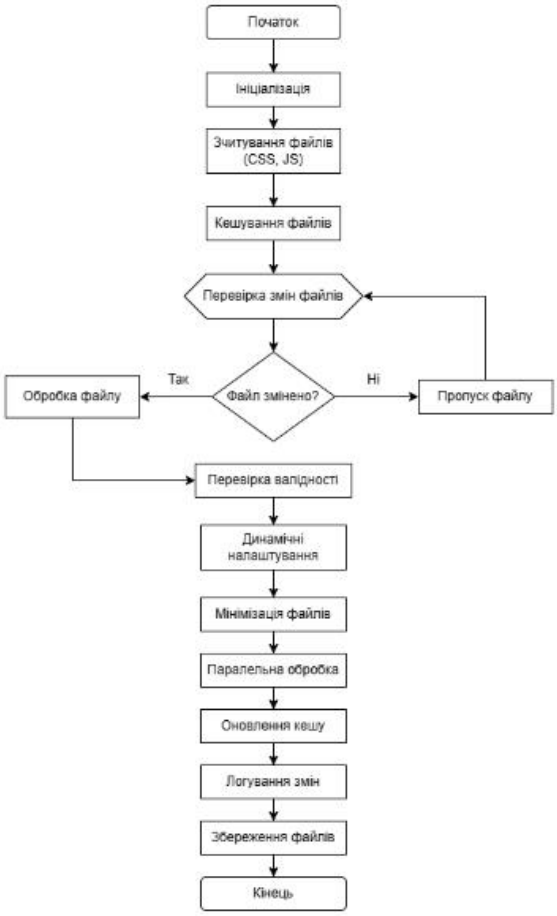
4

ФОРМАЛІЗОВАНА МОДЕЛЬ ОБ'ЄКТІВ АЛГОРИТМУ

1. **Ініціалізація середовища**: $E = \{V, B, C, G\}$
   Де *E* – середовище виконання; *V* – змінні; *B* – бібліотеки; *C* – кеш, *G* – граф залежностей.
2. **Зчитування файлів**: $F = \{F1, F2, \ldots, Fn\}$
   Де *F* - множина файлів; *Fn* – відповідний файл який знаходиться в обробці.
3. **Кешування файлів**: $C(Fi) = \emptyset(для\ кожного\ Fi)$
   Де *C* – функція для збереження кешу; *Fi* – файл, який додається до кешу; ∅ – початкове значення кешу для кожного файлу (порожній кеш)
4. **Перевірка змін**: $\text{FileChanged}(v_i) = \begin{cases} \text{true}, & \text{якщо } H(F_i)_{\text{поточний}} \neq C(F_i), \\ \text{false}, & \text{якщо } H(F_i)_{\text{поточний}} = C(F_i). \end{cases}$
   Де *H*(*Fi*) – функція для хешування поточного файлу *Fi*; *C*(*Fi*) – функція для кешування про файл *Fi* яка зберігає його хеш; *Fi* – файл в обробці.
5. **Перевірка валідності коду:** $V(Fi) = True/False$
   Де *V* – функція перевірки валідності; *True*/*False* – результат валідації; *Fi* – файл, що перевіряється.
6. **Динамічні налаштування**: $D(M) = True/False$
   Де *D* – функція визначення налаштувань; *M* – режим роботи; *True*/*False* – визначення мінімізації файлів залежно від режиму.
7. **Мінімізація файлів**: $R(Fi) = min(Fi)$
   Де *R* – функція для мінімізації; *Fi* – файл, що обробляється; *min*(*Fi*) – мінімізований файл.
8. **Паралельна обробка**: $P(F) = \{F1, F2, \ldots, Fk\}$
   Де *P* – функція паралельної обробки; {*F*1, *F*2, . . . , *Fk*} – набір файлів, які обробляються одночасно.
9. **Оновлення кешу**: $C(Fi) = H(Fi)current$
   Де *C* – функція кешування; *H*(*Fi*)*current* – поточний хеш файлу, що оновлюється у кеші.
10. **Логування**: $L(Fi)$
    Де *L* – функція логування; *Fi* – файл, для якого записується інформація про зміну чи можливі помилки.
11. **Збереження оброблених файлів**: $S(Fi)$
    Де *S* – функція збереження; *Fi* – файл, що зберігається після обробки.

5

БЛОК-СХЕМА АЛГОРИТМУ



6

ФОРМАЛІЗОВАНА МОДЕЛЬ РЕАЛІЗОВАНОГО МЕХАНІЗМУ ІНТЕЛЕКТУАЛЬНОГО КЕШУВАННЯ

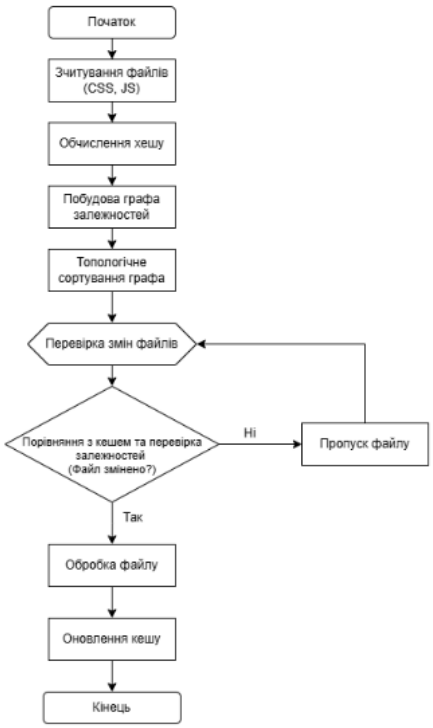
1. **Ініціалізація кешу та графа залежностей**: $G = (V, E)$, де $V = \{v1, v2, \ldots, vn\}, E = \{(vi, vj)\}$
   Де *G* – граф залежностей файлів;
   V – Множина вершин графа, де кожна вершина *vi* відповідає файлу;
   E – Множина ребер графа, які описують залежності між файлами (*vi*, *vj*) тобто файл *vi* залежить від файлу *vj*.
2. **Обчислення хешу файлу**: $H(Fi) = MD5(Fi)$
   Де *H*(*Fi*) – Хеш функція, яка обчислює унікальний хеш файлу *Fi* для перевірки змін;
   MD5(*Fi*) – Алгоритм MD5, що створює криптографічний хеш для файлу *Fi*.
3. **Порівняння хешу з кешем з врахуванням графа залежностей**:

$$\text{FileChanged}(v_i) = \begin{cases} \text{true}, & \text{якщо } H(v_i)_{\text{current}} \neq C(v_i), \\ \text{true}, & \text{якщо } \exists v_j \in \text{deps}(v_i) : \text{status}(v_j) = \text{modified}, \\ \text{false}, & \text{інакше.} \end{cases}$$

   Де *vi* – Файл, що перевіряється;
   H(*vi*)*current* – Поточний хеш файлу *vi*;
   C(*vi*) – Множина залежностей файлу *vi*;
   deps(*vi*) – Поточний хеш файлу *vi*;
   status(*vj*) – Статус залежного файлу *vj* (змінений або незмінений).
4. **Топологічне сортування**: $TopSort(G) = \{v1, v2, \ldots, vn\}$, де vi передує vj, якщо $(vi, vj) \in E$.
   Де *TopSort*(*G*) – порядок обробки файлів відповідно до графа *G*;
   vi, *vj* – вершини графа (файли), де залежність (*vi*, *vj*) означає, що *vj* залежить від *vi*.
5. **Оновлення кешу**: $C(Fi) = H(Fi)current$
   Де *C*(*Fi*) – значення хешу файлу *Fi*, що оновлюється після обробки;
   H(*Fi*)*current* – Поточний хеш файлу *Fi*, згенерований після обробки;

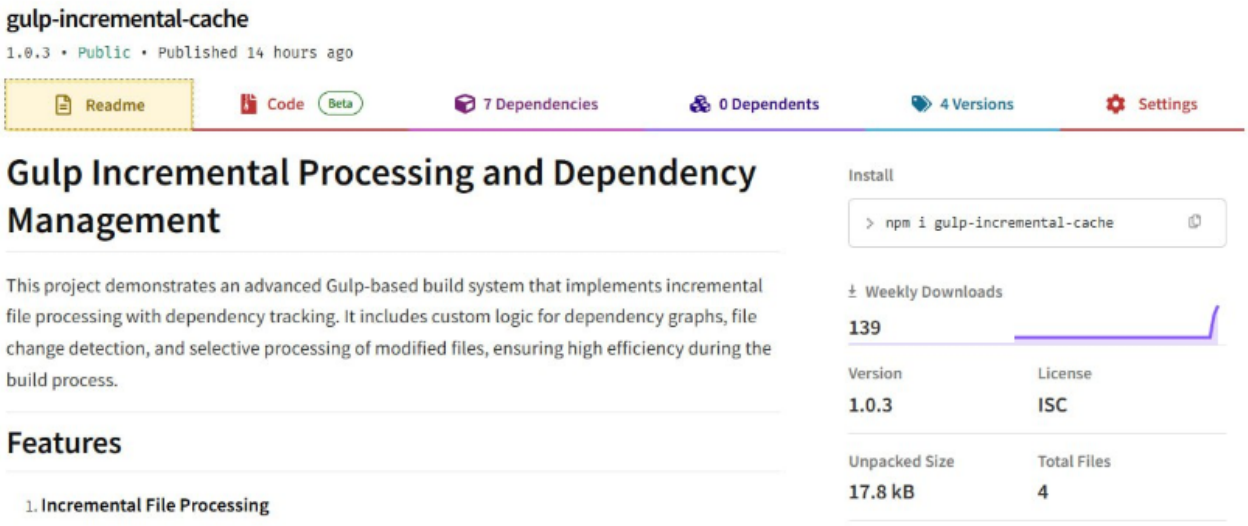
7

БЛОК-СХЕМА АЛГОРИТМУ МЕХАНІЗМУ ІНТЕЛЕКТУАЛЬНОГО КЕШУВАННЯ



8

РЕАЛІЗАЦІЯ ТА ПУБЛІКАЦІЯ АЛГОРИТМУ ЯК NPM-ПАКЕТУ



9

ІНТЕГРАЦІЯ В ІСНУЮЧИЙ ПРОЕКТ

```
gulpfile.js
1  import { runGulpTasks } from 'gulp-incremental-cache';
2
3  runGulpTasks();
```

```
[20:52:22] Finished 'minify-js' after 8.91 ms
[20:52:27] Starting 'validate-js'...
[20:52:27] Finished 'validate-js' after 18 ms
[20:52:27] Starting 'minify-js'...
[20:52:27] C:\Users\fetko\WebstormProjects\gulp-project\src\js\script1.js: modified
  - Current hash: 150b612e969f637ae6a274be5f9f35ea
  - Cached hash: 150b612e969f637ae6a274be5f9f35ea
  - Status: will be processed
[20:52:27] C:\Users\fetko\WebstormProjects\gulp-project\src\js\script2.js: unchanged
  - Current hash: 35d0cc10a8ac8e81d00ea33f3daf6b08
  - Cached hash: 35d0cc10a8ac8e81d00ea33f3daf6b08
  - Status: will be skipped
[20:52:27] Finished 'minify-js' after 7.4 ms
```

10

ПОРІВНЯЛЬНИЙ АНАЛІЗ ПРОДУКТИВНОСТІ АЛГОРИТМІВ

Традиційний метод: без кешування.
Метод кешування: використовує плагіни gulp-cached і gulp-remember.

Обидва методи включають базові процеси: обробка, мініфікація*, валідація коду.

Показник	Традиційний метод	Метод кешування	**Інкрементальний алгоритм**
Час першої збірки (мс)	920.75 мс	687.21 мс	**665.58 мс**
Час повторної збірки (мс)	642.89 мс	638.78 мс	**573.58 мс**
Час збірки зміненого файлу (мс)	622.28 мс	649.14 мс	**561.17 мс**
Час збірки із залежностями (мс)	634.17 мс	640.00 мс	**543.85 мс**
Використання пам’яті (мб)	18.58 мб	12.96 мб	**12.93 мб**

Мініфікація (також *мінімізація*) - це процес видалення всіх непотрібних символів із вихідного коду інтерпретованих мов програмування або мов розмітки без зміни його функціональності.

ПРОВЕДЕННЯ АНАЛІЗУ ПРОДУКТИВНОСТІ АЛГОРИТМІВ

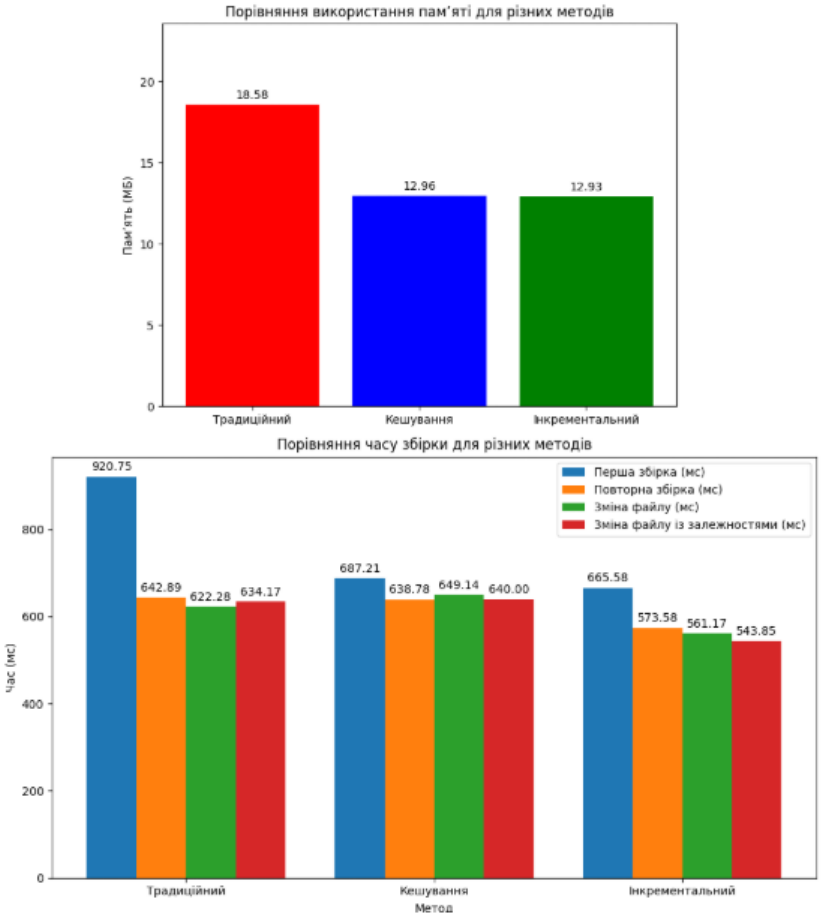
Тестове середовище:
- Операційна система: macOS Monterey 12.5;
- Система: Процесор Apple M1, 8 ГБ RAM;
- Інструменти: Node.js 16.15.0, Gulp 4.0.2.

Структура тестового проекту:
- 50 CSS і 50 JavaScript файлів;
- Залежності:
  - CSS: 10 імпортів на файл;
  - JavaScript: 5 імпортів між модулями

(index)	First Build (ms)	Rebuild (ms)	Single Change (ms)	Dependency Change (ms)	Memory Usage (MB)
Traditional Build	920.748792	642.888542	622.284791	634.167125	0.01857757568359375
Cached Build	687.212208	638.782625	649.139417	640.001125	0.01296234130859375
Incremental Build	665.589166	573.57825	561.173625	543.85	0.012939453125

```
yuriifetko@MacBook-Air-3 gulp-project-2 %
```

РЕЗУЛЬТАТИ МОДЕЛЮВАННЯ



13

ВИСНОВКИ

1. Проведено аналіз існуючих інструментів автоматизації збірки (Gulp, Webpack, Grunt) та методів кешування (gulp-cached, gulp-remember, gulp-newer). Виявлено недоліки, зокрема ігнорування залежностей між файлами та надмірну обробку незмінених файлів.
2. Досліджено сучасні методи мінімізації CSS і JavaScript файлів. Здійснено порівняння ефективності та складності підходів, що дозволило обрати оптимальні рішення для реалізації.
3. Розглянуто механізми інкрементальної обробки та кешування. Виявлено, що врахування залежностей між файлами підвищує продуктивність збірки.
4. Розроблено інкрементальний алгоритм обробки файлів із використанням графа залежностей і механізму інтелектуального кешування. Алгоритм забезпечує вибіркову обробку лише змінених файлів.
5. Реалізовано запропонований алгоритм як npm-пакет для середовища Gulp. Забезпечено підтримку динамічних налаштувань (режими розробки та продакшн-збірки), що дозволяє адаптувати алгоритм до різних умов використання.
6. Проведено тестування алгоритму та здійснено порівняльний аналіз продуктивності:

   - Економія часу обробки до **27.7%**;

   - Скорочення використання пам’яті на **30.4%**;

   - Забезпечення ефективного управління залежностями між файлами, що підвищує точність і зменшує надмірну обробку.

14

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Фетько Ю.І., Залива В.В “ОПТИМІЗАЦІЯ ПРОЦЕСУ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ШЛЯХОМ ВПРОВАДЖЕННЯ АЛГОРИТМУ ІНКРЕМЕНТНОЇ ОБРОБКИ ФАЙЛІВ” // Всеукраїнська науково-технічна конференція "Виклики та рішення в програмній інженерії". – Київ: ДУІКТ, 2024. – С. 79-80.

2. Фетько Ю.І., Залива В.В “ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ШЛЯХОМ ОПТИМІЗАЦІЇ ПРОЦЕСІВ ОБРОБКИ РЕСУРСІВ” // Всеукраїнська науково-технічна конференція "Виклики та рішення в програмній інженерії". – Київ: ДУІКТ, 2024. – С. 204-206.