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

ПОРІВНЯЛЬНИЙ АНАЛІЗ ТЕХНОЛОГІЇ HOTWIRE ТА ТРАДИЦІЙНОГО REST API + REACT SPA ДЛЯ РОЗРОБКИ ВЕБ-ЗАСТОСУНКІВ.

Текстова частина до курсової роботи
за спеціальністю 121 «Інженерія програмного забезпечення»

Керівник курсової роботи
ст.в. Захоженко П.О.
(прізвище та ініціали)
_____
(підпис)
“___” _____ 2025 р.

Виконав студент Хоменко М.О.
(прізвище та ініціали)
“____” _____ 2025 р.

Київ 2025

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

ЗАТВЕРДЖУЮ
Викладач кафедри мультимедійних
систем, старший викладач
_____ Захоженко П.О.
(підпис)
„____”_____2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту 4 курсу факультету інформатики
Хоменко Максиму Олександровичу
ТЕМА: Порівняльний аналіз технології Hotwire та традиційного REST API + React SPA для розробки веб застосунків.

Зміст ТЧ до курсової роботи:

Зміст
Анотація
Розділ 1. Вступ
Розділ 2. Теоретичне підґрунтя та огляд технологій.
Розділ 3. Реалізація на практиці
Розділ 4. Порівняльний аналіз
Розділ 5. Висновки
Список використаних джерел

Дата видачі “___” _____ 2025 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: <u>Порівняльний аналіз технології Hotwire та традиційного REST API + React SPA для розробки веб застосунків.</u>

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1	Отримання завдання на курсову роботу	26.11.2024	Завдання отримано, початкове планування розпочато
2	Дослідження Hotwire та REST API + React SPA: принципи, архітектура, порівняння	26.12.2024	Аналіз документації, вивчення підходів
3	Розробка першого прототипу React SPA + REST API	01.03.2025	Базова логіка, серверна частина, Turbo Streams, Turbo Frames
4	Розробка першого прототипу Hotwire-застосунку	10.04.2025	Налаштування бекенду, frontend-компоненти, API-запити
5	Тестування та оптимізація обох застосунків	20.04.2025	Порівняння продуктивності, юзабіліті, рефакторинг
6	Аналіз результатів, формування висновків	05.05.2025	Обґрунтування вибору технології, плюси/мінуси
7	Оформлення теоретичної частини курсової роботи	10.05.2025	Написання розділів, технічної документації
8	Фінальні правки, підготовка до здачі	18.05.2025	Остаточне редагування, форматування, вичитка
9	Захист курсової роботи	21.05.2025	
10	Підготовка до презентації (слайди, виступ)	01.06.2025	
11	Фінальний захист курсової роботи	02.06.2025	

Хоменко М.О. _____

Захоженко П.О. _____

“_____”_____ 2025 р.

Зміст

Анотація 1
РОЗДІЛ 1. ВСТУП. 2
1.1. Актуальність та контекст дослідження 2
1.2. Проблема та завдання дослідження 3
1.3. Масштаб та обмеження 4
1.3.1. Короткий огляд джерел 5
1.3.2. Методологія дослідження 5
1.4. Структура роботи 6
РОЗДІЛ 2. ТЕОРЕТИЧНЕ ПІДГРУНТЯ ТА ОГЛЯД ТЕХНОЛОГІЙ. 8
2.1. Огляд сучасних підходів до веб-розробки 8
2.1.1. Еволюція клієнт-серверних архітектур 8
2.1.2. Односторінкові додатки (SPA) та SSR 9
2.1.3. Сучасні тренди (2020–2025) 10
2.2. REST API + React SPA: поняття та принципи 12
2.2.1. Як працюють REST API 12
2.2.2. Роль React у фронтенд розробці 13
2.3. Hotwire: Поняття та принципи 14
2.3.1. Походження та цілі Hotwire 14
2.3.2. Turbo Drive, Turbo Streams, Stimulus 15
2.4. Ключові відмінності між двома підходами 17
2.4.1. Архітектура 17
2.4.2. Взаємодія з бекендом 17
2.4.3. Рендеринг: клієнт проти серверу 18
2.4.4. Передача даних і затримки 20
РОЗДІЛ 3. РЕАЛІЗАЦІЯ НА ПРАКТИЦІ. 23
3.1. Мета експериментальної реалізації та базова функціональність 23
3.2. Розробка версії React SPA + REST API 24
3.2.1. Бекенд 24
3.2.2. Фронтенд 25
3.2.3. Інтеграція 27
3.3. Розробка Hotwire-версії 28
3.3.1. Налаштування бекенду 28
3.3.2. Turbo Streams / Frames 28
3.3.3. Динаміка без Stimulus 29
3.4. Проблеми, які виникли під час розробки 30
РОЗДІЛ 4. ПОРІВНЯЛЬНИЙ АНАЛІЗ. 32

4.1. Порівняння продуктивності 32
4.2 Користувацький досвід та інтерактивність 35
4.2.1. React SPA + Spring Boot 36
4.2.2. Hotwire + Rails 7 38
4.2.3. Порівняльний аналіз продуктивності 41
4.3. Продуктивність розробника 44
4.3.1. Залежності та розмір коду 44
4.3.2. Час запуску та редагування 46
4.3.3. Крива навчання та документація 48
РОЗДІЛ 5. ВИСНОВКИ. 51
5.1. Підсумок результатів 51
5.2. Варіанти використання та найкращі сценарії 51
5.3. Остаточний вердикт: Який підхід є кращим? 52
5.4. Майбутні перспективи та напрямки досліджень 53
Список використаних джерел 56

Анотація

У дипломній роботі проведено порівняльний аналіз архітектурних підходів до побудови веб-застосунків на прикладі React (SPA) та Hotwire (SSR). Актуальність дослідження зумовлена зростаючими вимогами до продуктивності, зручності використання (UX) та ефективності процесу розробки (DX) у сучасних веб-системах.

Метою дослідження є оцінка переваг і недоліків двох підходів шляхом реалізації і тестування двох еквівалентних веб-застосунків із однаковою бізнес-логікою (kanban-дошка). Для досягнення мети використано експериментальний метод, що включає профілювання продуктивності (Lighthouse, Chrome DevTools), тестування API-латентності та аналіз коду.

У ході дослідження встановлено, що React SPA забезпечує вищу продуктивність у складних інтерфейсах і кращу масштабованість, тоді як Hotwire дозволяє значно швидше створювати MVP завдяки спрощеній архітектурі та мінімальній залежності від JavaScript.

Результати можуть бути корисними для команд розробників при виборі технологій залежно від вимог проєкту: продуктивність, час розробки, складність підтримки. Робота містить практичні рекомендації щодо застосування SPA та SSR-архітектур у різних контекстах.

РОЗДІЛ 1. ВСТУП.

1.1. Актуальність та контекст дослідження

Значення веб-розробки у створенні сучасних інформаційних систем постійно зростає - від продуктивності до інтерактивності та доступності. Із постійним зростанням вимог до швидкодії, зручності користування та гнучкості, вибір архітектурного підходу до побудови веб-застосунків набуває вирішального значення. Сьогодні широко використовуються як односторінкові додатки (SPA - *Single Page Application*), побудовані за допомогою таких бібліотек, як React, так і серверні архітектури рендерингу (SSR − *Server-Side Rendering*), серед яких особливу увагу привертає підхід Hotwire.

React залишається найпоширенішим веб-фреймворком - його використовують 38.9% учасників опитування Stack Overflow 2024 [3]. Водночас стрімко зростає популярність SSR-орієнтованих рішень, зокрема Next.js: 16.5% респондентів використовують цей фреймворк, а близько 30% з них готові працювати з ним і надалі [3]. Це свідчить про підвищений інтерес до SSR-архітектур як ефективного засобу забезпечення продуктивності, SEO та швидкого часу до першого рендеру, що, у свою чергу, підкреслює актуальність порівняння між React (SPA) і Hotwire (SSR).

Хоча Hotwire ще не представлений у ключових технологічних рейтингах, він є логічним продовженням традиційної архітектури SSR, вбудованої в Ruby on Rails [4]. Незважаючи на обмежене поширення за межами екосистеми Rails, впровадження Hotwire значно розширює можливості фронтенд-інтерактивності без залучення складних фреймворків [5]. Таким чином, дослідження Hotwire в контексті порівняння з React має не лише технічне, але й концептуальне значення як приклад альтернативної парадигми побудови сучасних веб-додатків.

Тема даної кваліфікаційної роботи є актуальною, оскільки дозволяє провести поглиблений порівняльний аналіз двох сучасних архітектурних підходів, а також

оцінити їх ефективність на практиці шляхом реалізації двох повнофункціональних прототипів з ідентичною бізнес-логікою. Такий підхід дозволяє об'єктивно проаналізувати технічні особливості, простоту розробки та потенційне застосування кожної технології і сформулювати обґрунтовані рекомендації щодо вибору архітектурного підходу в залежності від вимог проекту.

1.2. Проблема та завдання дослідження

Хоча SPA і SSR архітектури широко використовуються в сучасній веб-розробці, більшість існуючих досліджень зосереджені в основному на питаннях продуктивності, таких як час рендерингу, завантаження сторінки і навантаження на сервер. Це обмежує глибину аналізу і ускладнює обґрунтований вибір архітектури для конкретних проектів, особливо в проектах зі складною логікою інтерфейсу або багатокористувацькою взаємодією.

Деякі сучасні дослідження [1, 2] розглядають інші аспекти, такі як досвід розробника, зручність підтримки коду та взаємодія з інтерфейсом. Однак такі роботи зазвичай залишаються оглядовими або зосереджуються на окремих фреймворках без практичної реалізації та повного функціонального аналізу. Це створює основу для дослідження, яке поєднує технічний аналіз з практичною оцінкою взаємодії інтерфейсу та досвіду розробників.

У межах цієї роботи розглядається проблема вибору ефективної архітектури для побудови системи управління завданнями (канбан-дошки), що включає в себе

- багатокористувацьку підтримку;
- управління завданнями (створення, редагування, видалення)
- інтеграція глобальної статистики;
- система оповіщення (email);
- виявлення та відстеження прострочених завдань.

Метою дослідження є порівняння підходів Hotwire (SSR) та React з REST API (SPA) в контексті побудови одного і того ж додатку для визначення їх придатності в залежності від вимог проекту.

Для досягнення цієї мети поставлено такі завдання:

1. Вивчити архітектурні принципи, переваги та недоліки обох підходів (Hotwire і React).
2. Реалізувати два повнофункціональні веб-додатки з ідентичною логікою.
3. Провести експериментальне тестування продуктивності, зручності використання (UX) та досвіду розробника (DX).
4. Узагальнити результати та сформулювати практичні рекомендації щодо вибору архітектури.

1.3. Масштаб та обмеження

Об’єкт дослідження - веб-застосунок для керування завданнями, що реалізує типову прикладну логіку: створення, редагування, сортування завдань, базову аналітику та підтримку багатокористувацької взаємодії. Такий тип системи широко застосовується в сучасних інформаційних платформах, зокрема корпоративного та освітнього спрямування, і є придатним середовищем для порівняння архітектурних підходів.

Предмет дослідження - вплив обраної архітектури веб-застосунку (Hotwire (SSR) та React з REST API (SPA)) на продуктивність системи, досвід кінцевого користувача та ефективність процесу розробки.

Обсяг дослідження свідомо обмежено з метою зосередження на архітектурних аспектах, а не на масштабуванні або безпекових механізмах. Зокрема:

- не моделюється повноцінне горизонтальне масштабування для великої кількості одночасних користувачів;
- авторизація реалізована на базовому рівні із застосуванням JSON Web Token (JWT);

- дослідження здійснюються в керованих (лабораторних) умовах із використанням локального або обмеженого серверного середовища.

1.3.1. Короткий огляд джерел

Теоретичну основу дослідження становить аналіз офіційної документації, практичних кейсів та наукових джерел, які висвітлюють переваги, недоліки та архітектурні принципи SPA та SSR.

До ключових джерел належать:

- наукові дослідження, зокрема дипломні роботи A. Meredova (2023) [1] та E. Celander, A. Möllestål (2024) [2], що містять актуальні огляди технологічних рішень і обґрунтовують потребу у глибшому практичному аналізі.
- документація фреймворку Hotwire (Hotwire.dev) [5], зокрема його компонентів Turbo та Stimulus;
- офіційні матеріали React та Ruby on Rails;

Використані джерела дозволяють комплексно оцінити не лише технічні характеристики, але й концептуальні відмінності у підходах до побудови веб-застосунків.

1.3.2. Методологія дослідження

У межах дослідження застосовано емпіричний підхід з елементами експериментального аналізу. Для цього реалізовано два прототипи системи управління завданнями з однаковим функціональним обсягом: один - із використанням React у парадигмі SPA, інший - на основі Hotwire як SSR-рішення.

Оцінювання продуктивності проводиться за допомогою браузерних інструментів Chrome DevTools та Lighthouse. Вимірюються показники:

- часу першого рендеру (First Contentful Paint);

- часу до взаємодії (Time to Interactive);
- обсягу переданих ресурсів;
- продуктивності при взаємодії з інтерфейсом.

Досвід кінцевого користувача (UX) оцінюється на основі часу реакції інтерфейсу, кількості кроків до виконання базових дій та візуальної стабільності елементів під час навігації.

Досвід розробника (DX) аналізується через:

- складність конфігурації середовища;
- обсяг коду, необхідного для реалізації основної функціональності;
- кількість зовнішніх залежностей;
- зручність інструментів дебагінгу та тестування.

Отримані результати подаються у вигляді порівняльних таблиць, графіків та підсумків, що дозволяють обґрунтувати доцільність застосування кожного підходу залежно від типу задачі, вимог до продуктивності й організаційного контексту.

1.4. Структура роботи

Кваліфікаційна робота складається з п'яти розділів, кожен з яких виконує окрему функцію в досягненні мети дослідження.

Розділ 1 присвячено вступу, в якому обґрунтовано актуальність теми, сформульовано проблему, визначено мету та завдання дослідження, а також коротко окреслено його контекст.

Розділ 2 охоплює теоретичну базу дослідження. У ньому розглядаються архітектурні особливості SPA (на прикладі React) та SSR (на прикладі Hotwire), їхні принципи роботи, переваги, недоліки та сучасні тенденції використання. Окрему увагу приділено аналізу наукових та прикладних публікацій.

У розділі 3 описано практичну частину роботи, зокрема, процес реалізації двох прототипів системи управління завданнями на основі архітектур SPA та SSR. Висвітлено вибір технологій, структуру додатків, особливості реалізації функціональності та труднощі, що виникли в процесі розробки.

Розділ 4 присвячено порівняльному аналізу реалізованих рішень. Представлено результати дослідження, що охоплюють показники продуктивності, досвід кінцевого користувача (UX) та досвід розробників (DX). Порівняння представлено у вигляді таблиць, графіків та узагальнень.

Розділ 5 містить загальні висновки за результатами дослідження та рекомендації щодо вибору архітектурного підходу відповідно до функціональних вимог та контексту використання.

РОЗДІЛ 2. ТЕОРЕТИЧНЕ ПІДГРУНТЯ ТА ОГЛЯД ТЕХНОЛОГІЙ.

2.1. Огляд сучасних підходів до веб-розробки

2.1.1. Еволюція клієнт-серверних архітектур

Історично веб-розробка базувалася на класичній клієнт-серверній моделі, у якій рендеринг HTML виконувався виключно на стороні сервера. З кожним запитом сервер обробляв вхідні дані, формував повноцінну сторінку HTML та повертав її браузеру. Такий підхід, відомий як серверний рендеринг (SSR), домінував у перші десятиліття інтернету [6].

У 2010-х роках відбулася революція завдяки появі JavaScript-фреймворків, таких як Angular, React і Vue. Це дало змогу реалізовувати односторінкові застосунки (SPA), в яких логіка інтерфейсу та рендеринг перенеслися на сторону клієнта. Сервер став виконувати роль API-шлюзу, що передає лише структуровані дані, переважно у форматі JSON [6, 7].

Ця зміна дозволила створювати динамічні, інтуїтивно зрозумілі інтерфейси без перезавантаження сторінки, однак призвела до нових викликів, які вимагали переосмислення підходів до рендерингу.

З 2019 року почалося активне повернення до ідеї попереднього рендеру. У відповідь на обмеження SPA зросла популярність статичної генерації сайтів (SSG) та оновленого серверного рендерингу (SSR), вже в контексті сучасних фреймворків. Паралельно виникли ізоморфні (універсальні) фреймворки, зокрема Next.js, які дозволяють виконувати той самий JavaScript-код як на сервері, так і на клієнті, поєднуючи переваги обох підходів [6].

Еволюція архітектур веб-додатків - від SSR до CSR (Client-Side Rendering) і назад до гібридних рішень - демонструє, як розробники прагнуть поєднати найкращі риси обох парадигм - продуктивність і доступність SSR із гнучкістю та інтерактивністю SPA.

2.1.2. Односторінкові додатки (SPA) та SSR

Архітектура веб-застосунків упродовж останнього десятиліття набула значної гнучкості завдяки розвитку як клієнтських, так і серверних технологій. Два основні підходи - односторінкові застосунки (SPA) та рендеринг на сервері (SSR) - по-різному реалізують взаємодію між браузером, сервером і даними, і ці відмінності будуть коротко викладені у цьому підрозділі.

SPA та підхід Client-Side Rendering (CSR)

Односторінковий застосунок (SPA) - це веб-застосунок, який завантажує одну HTML-сторінку з мінімальним шаблоном, а подальші навігація, оновлення даних, рендер інтерфейсу і т.д. здійснюються повністю на клієнті за допомогою JavaScript. Сервер у цьому випадку виступає лише постачальником даних через API-запити (зазвичай у форматі JSON), а вся логіка побудови інтерфейсу, маршрутизації та оновлення DOM виконується безпосередньо у браузері користувача [6, 7].

До ключових фреймворків, які визначили створення архітектури SPA, можна віднести Angular (2010), React (2013) та Vue.js (2014). React, зокрема, запровадив концепцію віртуального DOM і компонентної структури, що значно спростило розробку складних інтерфейсів з високим рівнем повторного використання коду.

Серед основних переваг SPA:

- Плавна взаємодія без перезавантаження сторінки (single-page routing).
- Зменшення кількості запитів до сервера після первинного завантаження.
- Гнучка архітектура та швидка побудова динамічних елементів.

Однак підхід має й свої недоліки:

- Повільний перший рендеринг (First Contentful Paint), особливо на повільних мережах.
- Проблеми з SEO (вміст не індексується без додаткових налаштувань).

- Великий обсяг файлів JS, що впливає на час завантаження та взаємодію.
- Високі вимоги до потужності клієнтських пристроїв [6].

SSR та повернення до серверного рендерингу

Server-Side Rendering (SSR) - це підхід, за якого HTML сторінки генерується на сервері у відповідь на кожен запит. У такому випадку користувач одразу отримує повноцінну сторінку, яка не потребує додаткової обробки для відображення основного контенту. Така модель традиційно використовувалася у фреймворках типу Ruby on Rails або Django.

Сучасні реалізації SSR - такі як Next.js (для React), Nuxt (для Vue), або Astro - комбінують SSR з можливістю "гідрації" (hydration), коли клієнтська логіка «підключається» до вже згенерованої на сервері структури HTML. Це дозволяє досягти високої продуктивності на першому етапі та зберегти інтерактивність клієнта після завантаження [6].

Серед переваг SSR:

- Швидкий початковий рендеринг (менше часу до першого пікселя).
- Покращене SEO (весь контент доступний під час індексації).
- Краща продуктивність на слабших пристроях.

Недоліки підходу:

- Затримки при відображенні динамічних компонентів, які потребують клієнтської логіки.
- Ускладнення при реалізації інтерактивності на стороні користувача.

2.1.3. Сучасні тренди (2020–2025)

У період з 2020 по 2025 роки веб-розробка зазнала суттєвих змін під впливом зростаючих вимог до продуктивності, оптимізації під пошукові системи (SEO) та якості взаємодії з користувачем (UX). Впровадження Google Core Web Vitals - набору метрик, до якого належать Largest Contentful Paint (LCP), First Input Delay

(FID) і Cumulative Layout Shift (CLS), - стало каталізатором змін у підходах до рендерингу веб-сторінок.

Класичні SPA-додатки з рендерингом на клієнті демонструють слабші показники за цими метриками, особливо на повільних пристроях і при нестабільному інтернет-з’єднанні. CSR-додатки часто мають високі значення LCP та FID, що погіршує UX, тоді як SSR і SSG дозволяють стабільно знижувати ці показники завдяки попередньому рендерингу HTML і поступовій гідрації [8].

У відповідь на ці виклики дедалі більше фреймворків орієнтуються на SSR, SSG або гібридні рішення. Наприклад, Next.js активно впроваджує Server Components і Streaming SSR, які дозволяють скоротити TTFB (Time To First Byte) завдяки потоковій передачі фрагментів HTML та мінімізувати гідрацію JavaScript, покращуючи тим самим TTI (Time To Interactive) та CLS (Cumulative Layout Shift). У документації Next.js [9] зазначено, що ці підходи спрямовані на покращення Core Web Vitals без шкоди для динаміки інтерфейсу.

За результатами опитування State of JS 2022 [10], Next.js утримує лідерство серед SSR-фреймворків: 49% розробників повідомили про його використання, і 90% з них залишилися задоволеними. Інтерес до нових фреймворків також зростає: Astro демонструє рівень зацікавленості 67%, а SvelteKit - 66%, що вказує на загальну тенденцію до пошуку продуктивніших і гнучкіших альтернатив класичним SPA-рішенням.

Поряд із масштабними фреймворками виникла й протилежна тенденція - до спрощення клієнтської частини через мінімалістичні підходи, зорієнтовані на HTML. До таких і належить Hotwire - фреймворк, що прагне зменшити обсяг JavaScript та покладається на сервер для формування більшої частини інтерфейсу.

Hotwire, представлений компанією Basecamp [4, 5], складається з трьох компонентів: Turbo Drive (перехоплення навігації), Turbo Streams (оновлення DOM через WebSocket або HTTP), і Stimulus (мінімалістичний JavaScript-фреймворк). З інтеграцією в Ruby on Rails 7 Hotwire позиціонується як альтернатива SPA у випадках, коли в пріоритеті швидкість інтерфейсу й мінімізація складної логіки на фронтенді.

Попри зростаючий інтерес у професійних спільнотах, академічна література щодо Hotwire наразі обмежена, що ускладнює систематичну оцінку їхніх переваг порівняно з усталеними архітектурами. Утім, він активно використовується у внутрішніх CRM-системах, адміністративних панелях та B2B-інструментах, де критична простота підтримки, продуктивність та контроль над усім стеком [8].

У наступних підрозділах розглянемо два конкретних підходи на практиці - на прикладі архітектури REST + React SPA та Hotwire. Це дозволить детальніше проаналізувати сильні і слабкі сторони кожного з них у контексті сучасної веб-розробки.

2.2. REST API + React SPA: поняття та принципи

2.2.1. Як працюють REST API

Архітектура REST (Representational State Transfer), сформульована Роем Філдінгом у своїй докторській дисертації (2000), є одним із найпоширеніших підходів до організації клієнт-серверної взаємодії у веб-застосунках [11]. REST базується на принципі уніфікованого інтерфейсу, згідно з яким кожен ресурс (наприклад, користувач, продукт або повідомлення) має власний ідентифікатор у вигляді URI, а доступ до нього здійснюється за допомогою стандартних HTTP-методів: GET, POST, PUT, DELETE тощо.

Одним із фундаментальних принципів REST є безстанова взаємодія (statelessness): кожен запит до сервера повинен містити всю необхідну інформацію для його обробки, без збереження контексту на сервері. Це спрощує

масштабування, дозволяє ефективніше кешувати відповіді та покращує відмовостійкість.

Типовий REST API відповідає на запити у форматі JSON (згідно з RFC 8259), що дозволяє легко обробляти дані у браузері. Наприклад, запит GET /api/products/42 повертає об'єкт з даними конкретного товару, які потім відображаються інтерфейсом на основі React. Браузер, без повного перезавантаження сторінки, оновлює лише ті компоненти, що залежать від нового стану [12].

У SPA на базі React дані зазвичай отримуються з бекенду через REST API. React-компоненти надсилають запити до API, отримують відповіді у форматі JSON та оновлюють DOM без перезавантаження сторінки, забезпечуючи динамічну взаємодію користувача з інтерфейсом.

2.2.2. Роль React у фронтенд розробці

React - популярна бібліотека для створення інтерфейсів, розроблена компанією Facebook у 2013 році. Вона базується на компонентному підході: інтерфейс розбивається на багаторазові блоки з власною логікою та станом.

JSX - синтаксичне розширення JavaScript - дозволяє описувати структуру UI у вигляді HTML-подібного коду, що транслюється у виклики React.createElement. Віртуальний DOM, який порівнюється з поточним станом, дає змогу ефективно оновлювати лише змінені частини реального DOM, підвищуючи продуктивність [13].

З 2019 року основним способом роботи зі станом стали React Hooks - зокрема useState для локального стану та useEffect для побічних ефектів. Це спростило написання функціональних компонентів та відмову від класів [13].

Оскільки React не має вбудованої маршрутизації, у SPA-застосунках використовують сторонні рішення на кшталт React Router [14]. Більші

застосунки також інтегрують бібліотеки для керування станом (наприклад, Redux [15] або Zustand [16]) та обробки форм.

Завдяки своїй модульності, React легко масштабується як у малих, так і у великих проєктах. Його популярність пояснюється гнучкістю, активною спільнотою та багатою екосистемою супутніх інструментів.

2.3. Hotwire: Поняття та принципи

2.3.1. Походження та цілі Hotwire

Hotwire (HTML Over The Wire) - це набір фреймворків, представлений компанією Basecamp у 2020 році під час розробки поштового сервісу HEY [5]. Ініціатором підходу виступив Девід Гайнемайєр Ханссон (DHH), автор фреймворку Ruby on Rails. Основна ідея Hotwire полягає у мінімізації використання JavaScript на користь передавання вже зрендереного HTML із сервера до клієнта, що значно спрощує фронтенд-архітектуру.

На відміну від класичних SPA-архітектур, де більшість логіки побудови інтерфейсу та обробки стану перенесена на клієнтську сторону, Hotwire дотримується філософії серверно-орієнтованого підходу. Взаємодія користувача (натискання кнопок, переходи між сторінками тощо) ініціює HTTP-запити, у відповідь на які сервер повертає готові фрагменти HTML. Ці фрагменти потім динамічно вбудовуються у DOM за допомогою JavaScript-бібліотеки Turbo.

До складу Hotwire входять три ключові компоненти, які будуть описані детальніше у наступному підрозділі: Turbo Drive, Turbo Frames / Turbo Streams, та Stimulus.

Із релізом Rails 7 у 2021 році, Hotwire було включено до офіційного стеку як рекомендований спосіб створення динамічних інтерфейсів без залучення повноцінних SPA-фреймворків. Його інтеграція в екосистему Rails надала бекенд-орієнтованим командам інструменти для:

- зменшення JavaScript-залежностей,
- підтримки SEO-дружнього SSR за замовчуванням,
- спрощення структури застосунків,
- покращення часу рендерингу без втрати динаміки.

Такий підхід виявився особливо ефективним у сферах, де переважають внутрішні інтерфейси - адміністративні панелі, CRM, CMS або B2B-рішення - де критичною є стабільність, передбачуваність поведінки та простота обслуговування [4, 5].

2.3.2. Turbo Drive, Turbo Streams, Stimulus

Hotwire реалізує концепцію "HTML-over-the-wire" через три основні інструменти: Turbo Drive, Turbo Streams і Stimulus. У взаємодії вони забезпечують динамічну поведінку інтерфейсу без повноцінної клієнтської логіки, властивої SPA-фреймворкам.

Turbo Drive

Turbo Drive автоматично перехоплює кліки по посиланнях та надсилання форм, щоб оновити лише вміст елемента <body>, зберігаючи <head>; незмінним. Завдяки цьому ресурси, як-от CSS або скрипти, не завантажуються повторно, що зменшує навантаження на мережу та пришвидшує навігацію [17].

Цей підхід зберігає стан змінних JavaScript, історію переглядів (pushState) і дозволяє досягти ефекту "плавної" навігації між сторінками без перезавантаження. Концептуально це схоже на PJAX, однак Turbo Drive тісніше інтегрований у сучасну Rails-архітектуру та підтримує більшу гнучкість [17].

Turbo Streams

Turbo Streams дають змогу серверу надсилати спеціальні фрагменти HTML з тегом <turbo-stream>, у яких описано дію, яку слід виконати над DOM. Turbo підтримує дії append, prepend, replace, update та remove, що дозволяє

ефективно оновлювати частини сторінки в реальному часі без повного ререндеру [18].

Ці дії можуть передаватися як частина HTTP-відповіді або через WebSocket-з'єднання за допомогою ActionCable - вбудованої WebSocket-технології у Rails. У такий спосіб можна досягти асинхронного оновлення інтерфейсу, подібного до SPA, не переносячи логіку рендерингу на клієнт.

Stimulus

Stimulus - це мінімалістичний JavaScript-фреймворк із прогресивною інтеграцією, що дозволяє додати динамічну поведінку до існуючої структури HTML. Контролери Stimulus зв’язуються з DOM через `data-controller` і `data-action` атрибути [19].

На відміну від фреймворків типу React або Vue, Stimulus не керує станом централізовано. Він орієнтований на "поведінкову прив’язку", дозволяючи створювати компактні контролери, які реагують на DOM-події (click, input, focus тощо). Наприклад, валідація форм, кнопки "показати більше" і т.д. можуть бути реалізовані лише з контролером Stimulus.

У комплексі ці три компоненти - Turbo Drive для навігації, Turbo Streams для оновлення вмісту й Stimulus для поведінки - забезпечують простий і зрозумілий підхід до побудови динамічних інтерфейсів без повноцінного SPA-стека. Вони ідеально підходять для проєктів, де сервер є джерелом логіки та рендерингу.

Ці особливості роблять Hotwire привабливим для певних типів проєктів, але також породжують низку архітектурних обмежень, які розглядаються у порівняльному аналізі далі.

2.4. Ключові відмінності між двома підходами

2.4.1. Архітектура

React + REST API та Hotwire демонструють два принципово різні архітектурні підходи до побудови веб-застосунків.

React (SPA) орієнтований на клієнтське середовище: HTML-інтерфейс формується в браузері за допомогою JavaScript, маршрутизація реалізується через React Router, а стан - через useState або глобальні менеджери типу Redux чи Zustand. Сервер постачає лише структуровані дані через REST або GraphQL API.

Hotwire (HTML-over-the-wire) реалізує server-driven UI: HTML генерується на сервері під час початкового запиту та динамічно оновлюється через Turbo Streams. Клієнт виконує лише відображення й мінімальну логіку взаємодії через Stimulus, без централізованого стану або складних компонентів JavaScript.

Це фундаментально впливає на архітектурну філософію: у випадку React перевагу має front-heavy логіка, тоді як у Hotwire контроль збережений за сервером.

2.4.2. Взаємодія з бекендом

Підходи React і Hotwire відрізняються принципами взаємодії з сервером і розподілу відповідальності між клієнтом і бекендом.

React у типовій архітектурі SPA використовує REST API (або у деяких випадках GraphQL) для отримання даних. Кожен компонент або view ініціює запити, обробляє відповіді у форматі JSON та самостійно оновлює інтерфейс. Для збереження та синхронізації стану (наприклад, кешу, помилок, завантаження) розробники часто використовують сторонні бібліотеки - такі як React Query, SWR або Redux Toolkit.

Ускладнення виникають при:

- реалізації optimistic UI (наперед оновлений інтерфейс до відповіді),
- обробці помилок (fallback UI, retry),
- кешуванні та інвалідації даних.

Усе це підвищує гнучкість, але також ускладнює архітектуру та вимагає ручного контролю.

Hotwire використовує іншу парадигму: взаємодія з бекендом здійснюється не через API, а через форми HTTP, переходи або події WebSocket. Сервер відповідає вже згенерованими фрагментами HTML (через Turbo Streams), які клієнт вбудовує в DOM. Керування станом (валідація, ререндер, редиректи) відбувається на сервері, тому клієнт не займається оновленням кешу або синхронізацією.

Наприклад, при створенні нового об'єкта:

- У React: компонент робить POST-запит, очікує відповідь, оновлює локальний або глобальний state.
- У Hotwire: форма надсилається на сервер, який у відповіді повертає оновлений фрагмент DOM, що автоматично вставляється на сторінку.

Ключова різниця полягає у тому, що React делегує більшу частину логіки на клієнт і потребує ретельного проєктування взаємодій, тоді як Hotwire централізує поведінку на сервері, спрощуючи розробку UI, але обмежуючи автономність клієнта.

2.4.3. Рендеринг: клієнт проти серверу

React SPA виконує початковий рендеринг лише після завантаження та виконання JavaScript-бандлу. Це означає, що браузер не має повноцінного контенту до завершення парсингу JS, що збільшує метрики FCP (First Contentful Paint) і TTI

(Time to Interactive), особливо на повільних мережах чи старих пристроях. Щоб компенсувати це, часто використовують SSR (Next.js), SSG або гібридні підходи (ISR) [9, 20].

Hotwire забезпечує серверний рендеринг за замовчуванням: розмітка HTML формується на сервері та надсилається у відповідь на запит, без необхідності завантаження JavaScript до появи основного вмісту. Це покращує першу візуальну реакцію сторінки та забезпечує кращі показники Core Web Vitals «із коробки» [18].

Можна навести кілька ключових відмінностей:

- React дозволяє тонко керувати рендерингом компонентів (через useMemo, lazy loading, suspense), але потребує складної оптимізації.
- Hotwire централізує рендеринг на сервері, що спрощує початкову завантажуваність, але обмежує кастомізацію складних інтерфейсів або анімацій.

Наслідки для SEO та доступності, відповідно, наступні:

- У React SPA без SSR основний вміст недоступний для crawler-ів без додаткового пререндеру.
- У Hotwire зміст доступний одразу, навіть з вимкненим JavaScript, що підвищує доступність і SEO за замовчуванням.

React дає більше контролю над клієнтським рендерингом, але вимагає додаткових зусиль для досягнення високої продуктивності. Hotwire, натомість, забезпечує стабільно швидке початкове завантаження та просту доступність, але менш гнучкий у кастомних сценаріях.

2.4.4. Передача даних і можливі затримки

Підходи React та Hotwire суттєво відрізняються у способі передачі даних між клієнтом і сервером, що впливає на затримки, ефективність та складність розробки.

React традиційно використовує формат JSON для обміну даними через REST API або GraphQL. Компоненти надсилають запити (через fetch, Axios тощо), отримують відповіді у форматі JSON, а потім парсять їх і оновлюють локальний стан. Цей підхід потребує ручного керування:

- завантаженням (loading states),
- помилками (error boundaries, retries),
- кешем (через React Query, SWR, Apollo Cache).

Проблеми over-fetching (отримання зайвих даних) і under-fetching (недостатність даних) у REST вирішуються частково через GraphQL, який дозволяє клієнту точно вказати структуру потрібної відповіді. Проте це збільшує складність бекенду та потребує побудови централізованої схеми, конфігурації кешу (наприклад, Apollo InMemoryCache) і генерації типів, що може бути надмірним для невеликих застосунків [11].

У великих застосунках, як-от Facebook, REST-архітектура виявилася недостатньо гнучкою - зокрема, через дублювання логіки та over-fetching. Це й стало підставою для створення GraphQL, який дозволяє клієнту самостійно визначати структуру відповіді й уникати передачі зайвих даних [21].

Hotwire, навпаки, не оперує окремими запитами: сервер повертає готові фрагменти HTML, які безпосередньо вбудовуються в DOM. Це значно спрощує реалізацію інтерфейсу, проте обмежує клієнтську гнучкість у повторному використанні даних, локальному збереженні або логіці post-processing.

У таблиці 2.4 наведено порівняння технологій оновлення даних у реальному часі, які пропонують React і Hotwire.

Таблиця 2.4. Порівняння технологій оновлення даних у реальному часі

Фреймворк	Технології	Затримка	Переваги	Недоліки
React	`setInterval`, polling, Apollo subscriptions, SWR revalidation, WebSocket через socket.io	Висока при polling (500–3000мс); середня при WS	Працює без серверної підтримки WebSocket	Потрібна ручна реалізація push-логіки
Hotwire	Turbo Streams + ActionCable (WebSocket)	<100–300мс	Push-оновлення «з коробки» через нативну WebSocket-інтеграцію	Залежність від Rails/ActionCable; слабке кешування

Примітка: затримка Turbo Streams вказана орієнтовно і залежить від реалізації ActionCable, навантаження на сервер та кількості WebSocket-підключень.

Приклад:

- React: щоб оновити список повідомлень, компонент періодично запитує `/api/messages`, перевіряє зміни, оновлює state.
- Hotwire: сервер відправляє `turbo-stream` через WebSocket, і повідомлення автоматично з’являється в DOM.

Загалом, React надає повний контроль над логікою оновлення даних, але потребує додаткових інструментів для реалізації реального часу (наприклад, socket.io або custom polling). Це дає гнучкість, але ускладнює архітектуру.

Hotwire натомість забезпечує вбудовану підтримку push-моделі через ActionCable, спрощуючи реалізацію живих оновлень. Однак цей підхід менш універсальний: складні сценарії кешування, offline-first або декілька каналів WebSocket потребують додаткової реалізації на сервері.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ НА ПРАКТИЦІ.

3.1. Мета експериментальної реалізації та базова функціональність

Метою практичної частини було створення двох функціонально еквівалентних веб-застосунків на основі різних архітектурних підходів:

- React SPA + REST API із використанням Spring Boot для бекенду;
- Hotwire + Rails 7 з повним серверним рендерингом.

Це дозволило провести порівняльний аналіз продуктивності, зручності розробки (DX) та поведінки інтерфейсу з урахуванням особливостей кожного стеку.

Функціональність, реалізована в обох версіях:

1. Базова логіка:
   - Операції CRUD для дошок та завдань;
   - Kanban-інтерфейс з поділом на статуси: *"очікує"*, *"в процесі"*, *"виконано"*, *"прострочено"*;
   - Пріоритети завдань (*низький*, *середній*, *високий*).
2. Інтерфейс та UX:
   - Зміна статусу завдань;
   - Фільтрація та сортування за пріоритетом та статусом;
   - Статистичні графіки реалізовані через Recharts / Chartkick (на основі Chart.js).
3. Додаткові функції:
   - Інтернаціоналізація (i18n) для української та англійської мов;
   - Сповіщення користувача на електронну пошту про додавання нового завдання (через SMTP);
   - Автоматичне оновлення статусу завдання до "прострочено" (фоновий таймер через TaskSchedulerService/SolidQueue).

Обсяг проєкту:

React + Spring Boot:

- Frontend: 34 вручну створених файли (React-компоненти, стилі, API-сервіси);
- Backend: 36 класів Java (контролери, DTO, сервіси, конфігурації);
- Загалом: близько 4000 рядків власного коду, без урахування залежностей (npm, Maven).

Hotwire + Rails 7:

- ~65 файлів, зокрема: controllers, views, turbo_streams, Stimulus-контролери;
- Інтеграція з ActionMailer (email) та SolidQueue (таймер);
- Загалом: 1,096 рядків власного коду (Ruby, HTML/ERB, Stimulus).

3.2. Розробка версії React SPA + REST API

3.2.1. Бекенд

Для реалізації серверної частини застосунку використано фреймворк Spring Boot з модулями Spring Security та Spring Data JPA. База даних: PostgreSQL.

Інтегровані сервіси:

- Автентифікація через JWT - захист приватних ендпоїнтів;
- повідомлення на електронну пошту - через JavaMailSender;
- фонові задачі - за допомогою TaskScheduler для оновлення статусів.

У таблиці 3.1 наведено головні ендпоїнти API, які були використані у застосунку React + REST API. Рисунок 3.2 зображує приклад відповіді JSON, яку фронтенд отримує від бекенду.

Таблиця 3.1. Головні ендпоїнти API у додатку React + REST API.

Метод	**Роут**	**Призначення**
POST	/api/auth/login	Вхід користувача
POST	/api/auth/register	Реєстрація

GET	/api/users/{id}	Отримання профілю користувача
GET	/api/boards	Отримати всі дошки
POST	/api/boards	Створити нову дошку
PUT	/api/boards/{id}	Оновити дошку
DELETE	/api/boards/{id}	Видалити дошку
GET	/api/tasks/board/{boardId}	Завдання для дошки
POST	/api/tasks	Створити нове завдання
PUT	/api/tasks/{id}	Редагувати завдання
DELETE	/api/tasks/{id}	Видалити завдання
PUT	/api/tasks/{id}/assign	Призначити користувача
GET	/api/stats/global	Глобальна статистика
GET	/api/stats/board/{id}	Статистика дошки

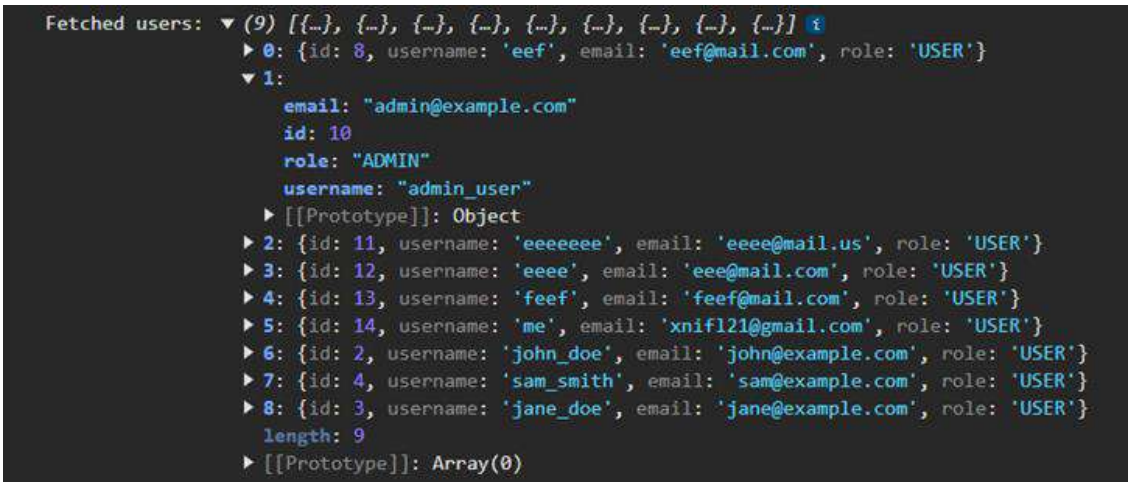

Рисунок 3.2. Приклад відповіді JSON (Spring Boot / REST API).

3.2.2. Фронтенд

Фронтенд реалізовано на базі React (Create React App), з використанням useState, useEffect для керування локальним станом та бібліотеки axios для запитів . Таблиця 3.3 надає базову структуру компонентів у фронтенді, а рисунки 3.4 і 3.5 демонструють приклад виклику API та використання useEffect.

Таблиця 3.3. Базова структура компонентів у React-застосунку.

Компонент	**Опис**
Dashboard.js	Основна сторінка з дошками
TaskBoard.js	Окрема дошка (Kanban)
Task.js	Компонент одного завдання
CreateBoardModal.js	Модальне вікно для створення дошки
EditBoardModal.js	Модальне вікно для редагування дошки
BoardStatsModal.js	Модальне вікно зі статистикою
Navbar.js	Верхня панель з локалізацією
Login.js / Register.js	Аутентифікація
api.js	Обгортка над axios із запитами до бекенду

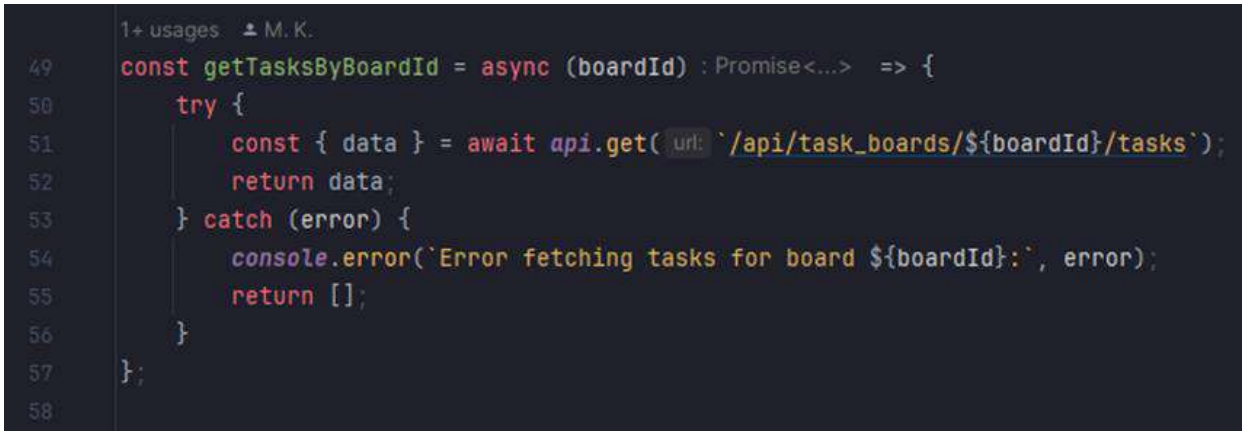

Рисунок 3.4. Приклад виклику API (api.js).

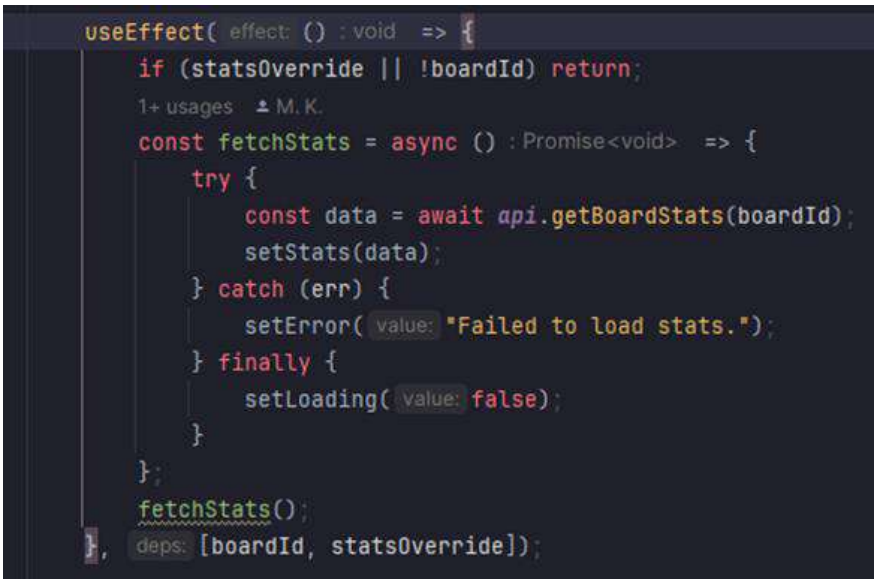

Рисунок 3.5. Приклад використання useEffect.

3.2.3. Інтеграція

Компоненти React звертаються до api.js, який виконує всі запити через axios. Усі запити додають токен JWT до хедера для захисту. На рисунку 3.6 показано приклад обробки помилки у застосунку.

Рисунок 3.6. Обробка помилки при неправильному логіні користувача.

При створенні завдання адміністратор може призначити виконавця, і на бекенді запускається email-повідомлення.

Інтернаціоналізація реалізована через react-i18next, вибір мови - у Navbar.

3.3. Розробка Hotwire-версії

3.3.1. Налаштування бекенду

Застосунок реалізовано на базі Ruby on Rails 7, який має нативну інтеграцію з Hotwire. Для реалізації функціональності було використано такі ключові бібліотеки (gems):

- hotwire-rails - інтеграція Turbo та Stimulus
- devise - автентифікація користувачів
- chartkick + groupdate - побудова графіків статистики
- solid_queue - фонові задачі через ActiveJob
- letter_opener - відлагодження email-розсилок у development-режимі

Хоча на старті частково використовувалися scaffold (наприклад, rails g scaffold Task title:string), переважна більшість логіки (контролери TaskBoardsController, TasksController, в'юшки index.html.erb, _task.html.erb) була реалізована вручну з урахуванням відповідей turbo_stream.

3.3.2. Turbo Streams / Frames

У застосунку реалізовано динамічні оновлення інтерфейсу через Turbo:

- створення, редагування чи видалення завдання не спричиняє повного оновлення сторінки;
- лише фрагмент DOM у відповіді оновлюється через turbo-frame або turbo-stream.

Рисунки 3.7 і 3.8 демонструють приклад використання Turbo Frame і Turbo Stream.

```
<!-- Boards Section -->
<%= turbo_frame_tag "boards" do %>
  <div class="flex gap-6 overflow-x-auto pb-4"
    <%= render partial: "task_boards/boards", locals: { task_boards: @task_boards } %>
  </div>
<% end %>
```

Рисунок 3.7. Приклад використання Turbo Frame (вставляється в HTML як turbo-frame id="task_42").

```
<%= turbo_stream.replace "board_#{@task.task_board_id}" do %>
  <%= render partial: "task_boards/board", locals: { board: @task.task_board } %>
<% end %>
<%= turbo_stream.append "flash", partial: "shared/flash", locals: { notice: "Task created." } %>
```

Рисунок 3.8. Приклад turbo_stream.erb для динамічного створення завдання.

Rails автоматично рендерить формат відповіді (turbo_stream.erb) на основі запиту, якщо клієнт підтримує Turbo. Усі CRUD-операції мають свої шаблони (create.turbo_stream.erb, update.turbo_stream.erb, тощо).

3.3.3. Динаміка без Stimulus

Попри те, що Stimulus є рекомендованим інструментом у складі Hotwire для організації клієнтської поведінки, у цьому застосунку Stimulus не використовувався.

Натомість динамічні дії - відкриття/закриття модальних вікон, обробка форм тощо - були реалізовані через ванільний JavaScript із використанням подій:

- turbo:load
- turbo:render
- turbo:frame-load

Цей підхід дозволив зберегти мінімалізм архітектури та уникнути додаткових залежностей, одночасно забезпечивши потрібну клієнтську взаємодію.

Примітка: Відмова від Stimulus не суперечить концепції Hotwire, оскільки сам Turbo не вимагає Stimulus як обов’язкової умови. У масштабних проєктах, де взаємодія між компонентами інтерфейсу опирається на численні події, Stimulus здатен суттєво покращити структуру поведінкової логіки.

3.4. Проблеми, які виникли під час розробки

У процесі створення обох реалізацій було виявлено низку технічних та архітектурних проблем, включаючи конфігурацію середовища, інтеграцію компонентів та підтримку динаміки інтерфейсу користувача.

React SPA + Spring Boot:

- Складність конфігурації: При розробці бекенду в Spring Boot необхідно було вручну налаштовувати DTO, класи Service, валідацію (через `@Valid`), а також реалізовувати кастомну логіку поштовиків і фонову обробку завдань (`@Scheduled`), що вимагало розширеного шаблонного коду.
- Інтеграція з фронтендом: На фронтенді React вимагав ретельної реалізації обробки помилок (error boundaries, try/catch), синхронізації токенів авторизації та валідації сесій.
- Дебагінг запитів: Відсутність централізованого моніторингу викликів API ускладнювала налагодження при помилках авторизації або відповідях 401.

Hotwire + Rails 7:

- Налаштування Devise: Інтеграція Devise із Hotwire вимагала ручного налаштування поведінки після логіну/реєстрації, особливо при перенаправленнях, які не працювали з коробки.
- Stimulus vs. Vanilla JS: Відмова від Stimulus на користь звичайного JavaScript була вимушеним рішенням через труднощі з налаштуванням фреймворку.
- Solid Queue: Використання SolidQueue для фонових задач потребувало окремого налаштування міграцій БД та виведення логіки у окремі worker-класи.
- Тестування було нетривіальним через «магію» Rails, коли помилки й виняткові ситуації виникали з незрозумілих причин. Повідомлення при

помилках часто були мінімальними, без деталей або інформації, які б допомогли відладці коду.

Попри виявлені труднощі, обидва застосунки були доведені до повноцінного функціонального паритету. Це дозволило провести об'єктивне архітектурне порівняння в наступних розділах, засноване на власній реалізації, а не лише на теоретичних припущеннях.

РОЗДІЛ 4. ПОРІВНЯЛЬНИЙ АНАЛІЗ.

4.1. Порівняння продуктивності

Для оцінки продуктивності було проведено серію тестів у контрольованому середовищі як для Hotwire + Rails 7, так і для React + Spring Boot. Тестування охоплювало як метрики фронтенду через Lighthouse, так і бенчмарки на бекенді за допомогою wrk та curl.

Тестове середовище

Hotwire + Rails 7:

- ОС: WSL (Ubuntu), запуск із RubyMine.
- Запуск Rails: у production-режимі (RAILS_ENV=production rails s).
- Threads: RAILS_MAX_THREADS=1, WEB_CONCURRENCY=1 для рівності з Spring Boot.
- HTTP: версія 1.1.
- Стиснення: Gzip увімкнено через Rack::Deflater.
- Збірка фронтенду: через esbuild (cssbundling-rails).

React + Spring Boot:

- ОС: WSL (Ubuntu), запуск із IntelliJ IDEA.
- Запуск Spring Boot API з JWT-аутентифікацією.
- Фронтенд зібрано через npm run build (vite/esbuild).
- Тестовий режим: теж HTTP/1.1, gzip активовано.
- Стиснення: використано server.compression у application.properties
- Threads: використано Tomcat для обмеження потоків (server.tomcat.threads.max=5, server.tomcat.threads.min-spare=1)

Метрики продуктивності фронтенду (Lighthouse)

Було виконано по 10 вимірювань на сторінках входу в обох застосунках. Наведено середні, мінімальні та максимальні значення основних метрик. Таблиця 4.1 демонструє результати цього експерименту.

Таблиця 4.1. Порівняння метрик React SPA / Hotwire.

Метрика	React SPA (Spring Boot)	Hotwire (Rails 7)
First Contentful Paint	0.20 с (мін: 0.2, макс: 0.2)	0.66 с (мін: 0.5, макс: 0.9)
Largest Contentful Paint	1.00 с (мін: 1.0, макс: 1.0)	0.66 с (мін: 0.5, макс: 0.9)
Speed Index	0.40 с (мін: 0.4, макс: 0.4)	0.69 с (мін: 0.5, макс: 1.0)

React SPA показує кращу продуктивність за FCP, оскільки весь HTML формується на клієнті після швидкого стартового завантаження. Проте Hotwire демонструє стабільні LCP/Speed Index після оптимізації - навіть при серверному рендерингу.

Латентність API (curl, 100 запусків)

Було виконано 100 команд curl на обох додатках (Ендпоінт Rails - /task_boards.json; Spring Boot - /api/task_boards (JWT додавався до запиту)), щоб порівняти латентність API. Таблиця 4.2 демонструє результати цього експерименту, де наведені середні значення основних метрик.

Таблиця 4.2. Порівняльна таблиця API-латентності Spring Boot / Rails 7.

Метрика	React + Spring Boot	Hotwire + Rails 7
DNS Lookup (середнє)	0.000023 с	0.000025 с
Connect Time (середнє)	0.000157 с	0.000214 с
Start Transfer (середнє)	0.005757 с	0.008019 с
Total Time (середнє)	0.005944 с	0.008075 с

API Spring Boot демонструє меншу латентність на всіх етапах - особливо у

start_transfer, завдяки кращій багатопоточності JVM і легкості JSON-відповідей. Rails показує помітно більший total time, особливо через серверний рендер HTML.

Стрес-тест бекенду (wrk)

Була проведена симуляція навантаження на бекенд обох застосунків з використанням команди wrk.

Параметри запуску:

- Тривалість: 30 секунд
- Threads: 2, 4, 8
- Connections: 10, 50, 100
- Ендпоінти:
    - Rails: /task_boards.json
    - Spring Boot: /api/task_boards (JWT додавався до запиту)

У таблицях 4.3 і 4.4 наведені результати цього дослідження.

Таблиця 4.3. Результати симуляції навантаження Hotwire (Rails 7).

Threads	Connections	Req/sec	Avg Latency	Max Latency	Transfer/sec	Timeouts
2	10	541.55	19.7ms	50.4ms	704.44 KB	10
4	10	530.84	16.1ms	65.3ms	690.51 KB	8
8	10	553.99	15.4ms	48.3ms	720.63 KB	8
2	50	436.99	230.9ms	1.21s	569.23 KB	0
4	50	470.09	215.9ms	917.8ms	612.34 KB	48
8	50	472.23	215.5ms	928.6ms	615.13 KB	48
2	100	468.51	466.9ms	2.00s	610.29 KB	256
4	100	471.56	479.7ms	2.00s	614.26 KB	216
8	100	468.39	493.5ms	2.00s	610.13 KB	173

Таблиця 4.4. Результати симуляції навантаження React + Spring Boot.

Threads	Connections	Req/sec	Avg Latency	Max Latency	Transfer/sec	Timeouts

2	10	472.91	22.0ms	72ms	263.79 KB	10
4	10	470.24	17.8ms	95ms	262.34 KB	8
8	10	472.11	17.6ms	52ms	263.36 KB	8
2	50	471.92	110ms	359ms	263.26 KB	50
4	50	452.72	105ms	240ms	252.50 KB	-
8	50	474.83	105ms	172ms	264.85 KB	48
2	100	470.97	220ms	403ms	262.68 KB	100
4	100	471.38	219ms	418ms	262.93 KB	100
8	100	474.28	209ms	433ms	264.53 KB	96

Загальні висновки

- React + Spring Boot демонструє кращу стійкість до навантажень і нижчу латентність, з вищою стабільною швидкістю обробки запитів (~470–475 req/sec).
- Hotwire + Rails 7 обмежується продуктивністю CRuby і серверним рендером, але після оптимізації фронтенду показує конкурентні результати за FCP і LCP.
- Spring Boot використовує переваги JVM (G1GC, многопоточність) та легші відповіді .
- Rails відповідає повними сторінками HTML, що збільшує Transfer/sec, але вимагає більше серверного часу.

4.2 Користувацький досвід та інтерактивність

Проведено емпіричне тестування інтерфейсної частини обох застосунків у реальних сценаріях користування. Основна увага зосереджена на швидкості рендерингу, перемиканні між сторінками, оновленні елементів, споживанні пам’яті та реакції системи на користувацькі дії.

4.2.1. React SPA + Spring Boot

У реалізації на основі React SPA додаток продемонстрував стійку продуктивність навіть після масштабування канбан-дошки з 8 до 20+ завдань на 4 дошках. Створення нових завдань відбувалося із затримкою до 1 с (переважно через серверну обробку).

Аналогічна затримка була виявлено й на великих платформах. Наприклад, інженери Airbnb зазначили, що використання чистого CSR у їхньому інтерфейсі спричиняло повільний рендеринг сторінок із динамічним контентом. У відповідь вони почали впроваджувати SSR-оптимізації через React.lazy, пріоритетизацію ресурсів, серверні підказки (server hints) та попереднє завантаження шрифтів, що дозволило скоротити TTFB та покращити Core Web Vitals [22, 23].

Водночас більша частина інших дій у React SPA - переміщення, редагування, перегляд - виконувалися практично миттєво.

Примітка: Незважаючи на те, що реалізований додаток демонструє ключові аспекти взаємодії між React і REST API, для повноцінної оцінки архітектурного підходу до взаємодії фронтенду й бекенду доцільно було б розглянути альтернативний варіант із використанням Rails як серверної частини замість Spring Boot. Такий підхід дозволив би порівняти рівень інтеграції, продуктивності та гнучкості між більшою кількістю архітектурних рішень, забезпечуючи кращий аналітичний паритет.

Профілювання у Chrome DevTools показало:

- Heap: зростання з 7.3 до 8.0 MB при активному редагуванні
- Scripting: ~374 мс
- Rendering: ~119 мс
- Повний сценарій (вхід + декілька змін): ≤ 15 с
- DOM: 2,200 → 3,400 вузлів

Метрики Web Vitals:

- LCP: 0.29 с
- CLS: 0.01
- INP: 24 мс (нижче порогового значення у 200 мс)
- FCP: 58.4 мс

Для тестування початкового завантаження (first visit) застосовано Chrome DevTools із очищенням кешу та увімкненою опцією "Disable cache". Сценарій охоплював вхід та перехід на дашборд.

Основні результати профілювання:

- FCP: 1.49 с
- LCP: 1.32 с
- INP: 16 мс
- CLS: 0.0
- Rendering: 1,168 мс
- Heap: ~932 КБ → 6.7 MB
- DOM: 12 → 921

На рисунку 4.5 видно результати цього профілювання.

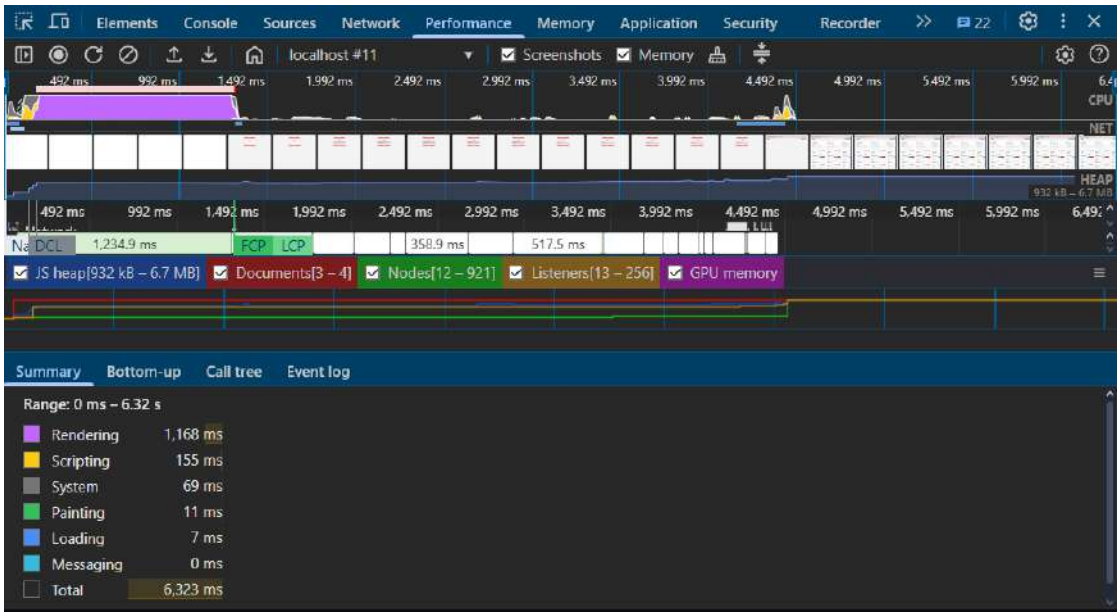

Рисунок 4.5. Результати профілювання завантаження застосунку «з нуля» (React SPA / Spring Boot)

У React SPA статистика (як глобальна, так і для окремих дошок) відображається через модальні вікна, без повного перезавантаження сторінки.

Chrome DevTools показав:

- Загальний час сценарію (вхід + відкриття модального вікна): ~2.14 с
- Scripting: 549 мс
- Rendering: 76 мс
- DOM-вузли: 924 → 2,517
- Heap: 9.7 → 23.6 MB

Навігація до статистики не викликає повного рендеру сторінки, що позитивно позначається на швидкодії та зниженні навантаження на систему.

4.2.2. Hotwire + Rails 7

Застосунок, реалізований з використанням стеку Hotwire + Rails 7, було протестовано у тих самих умовах, що і SPA на React, з акцентом на реальну взаємодію користувача: вхід у систему, видалення/редагування задач, перегляд

дошок. Архітектура реалізована через Turbo Frames та Turbo Streams із серверним рендерингом HTML.

Під час тестування поведінки інтерфейсу на складній структурі (4 дошки, понад 20 завдань) було виявлено, що створення та оновлення задач виконується з незначними затримками (до 1 с), зумовленими серверною обробкою.

Примітка: деякі проблеми продуктивності можуть бути пов’язані не зі стеком Hotwire як таким, а з конкретними особливостями реалізації застосунку. Наприклад, відсутність компонентного підходу (напр. ViewComponent), повторне рендерення всього DOM без часткових шаблонів, і базова обробка через JavaScript могли вплинути на продуктивність.

Профілювання Chrome DevTools

Було записано кілька сценаріїв:

- Сценарій 1: вхід у систему → видалення кількох задач.
- Сценарій 2: вхід у систему → редагування задачі.
- Сценарій 3: повне завантаження сторінки після очищення кешу (перше відвідування).

Профілювання сценаріїв 1–2 (вхід + взаємодії):

- Загальний час сценарію: ~4 с
- Scripting: 2,965 мс (вище, ніж у React, через повну реконструкцію DOM)
- System: 205 мс
- Rendering: 42 мс
- Painting: 35 мс
- DOM-вузли: зросли з ~3700 до понад 9300 під час сценарію
- JS heap: 7.3 → 15.4 MB
- INP: 8 мс (ідеальний показник)
- CLS: 0.0

Сценарій 3 (перший візит):

- First Contentful Paint (FCP): 1.2 с
- Largest Contentful Paint (LCP): 1.18 с
- Interaction to Next Paint (INP): 8 мс
- CLS: 0.0
- Rendering (CPU time): ~1,200 мс
- JS heap: 7.0 → 15.6 MB
- DOM-вузли: 2,941 → 6,090

На рисунку 4.6 видно результати профілювання сценарію 1.

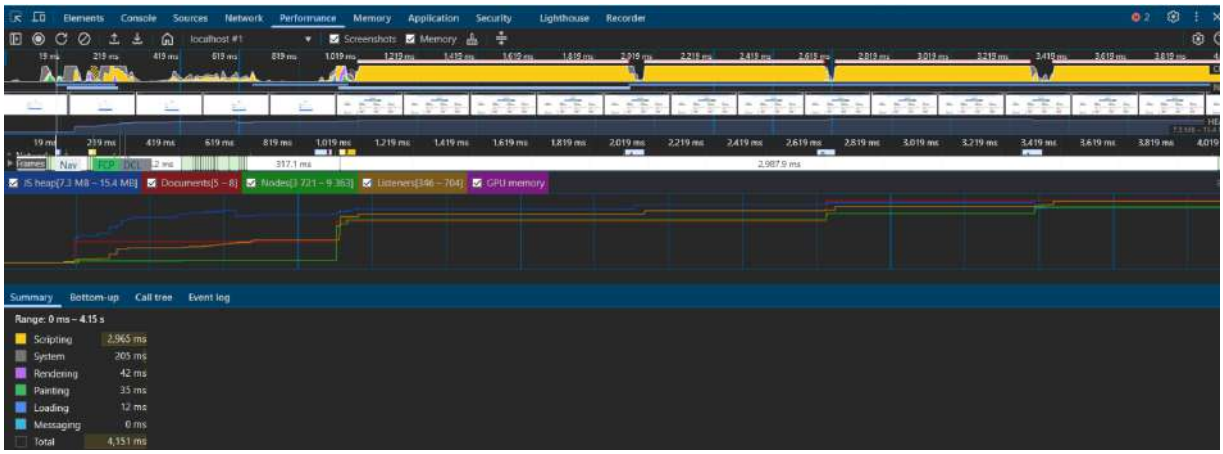

Рисунок 4.6. Профілювання сценарію входу та видалення кількох завдань у Hotwire.

У реалізації на Hotwire відкриття статистики здійснюється через повну навігацію на окрему сторінку. Навіть при використанні Turbo Drive це спричиняє повторне завантаження та рендеринг інтерфейсу, що впливає на час виконання та споживання ресурсів.

Chrome DevTools показав:

- Загальний час сценарію (вхід + повна навігація): ~3.77 с
- Scripting: 721 мс

- Rendering: 178 мс
- DOM-вузли: 27,347 → 37,145
- Heap: 20.4 → 34.6 MB

Повне оновлення DOM та збільшення кількості слухачів подій (>3,000) свідчать про відсутність компонентної декомпозиції та ефективного кешування при переходах.

4.2.3. Порівняльний аналіз продуктивності

Зіставлення ключових показників інтерфейсної продуктивності обох реалізацій свідчить про суттєві відмінності в архітектурному підході.

React SPA забезпечує стабільно кращі показники рендерингу, особливо в частині обробки DOM та JavaScript. Завдяки використанню Virtual DOM та компонентної архітектури, React демонструє нижчі показники Scripting Time (223 мс проти 2965 мс у Hotwire) та менше навантаження на DOM (до 3400 елементів проти понад 9300).

Hotwire, натомість, досягає чудових результатів за метрикою INP (<10 мс), зберігаючи високу інтерактивність інтерфейсу після завантаження. Проте загальна продуктивність під час складних маніпуляцій з DOM виявилась нижчою, що пов’язано з повним перерендеренням HTML через Turbo Streams без локального оновлення.

У сценаріях "першого візиту" обидві архітектури мають порівняні значення LCP (1.18–1.32 с) і FCP (1.2–1.49 с), але Hotwire значно активніше використовує CPU та пам’ять через повну ініціалізацію всього HTML.

CLS = 0 в обох випадках свідчить про стабільність візуального макету, без зміщень або "стрибків" під час завантаження.

Загалом, React SPA виявляється ефективнішим у складних та динамічних взаємодіях, тоді як Hotwire добре показує себе при простих сценаріях, зберігаючи дуже низьку латентність після початкового рендерингу. На рисунку 4.7 наведено порівняння основних метрик при завантаженні застосунків «з нуля».

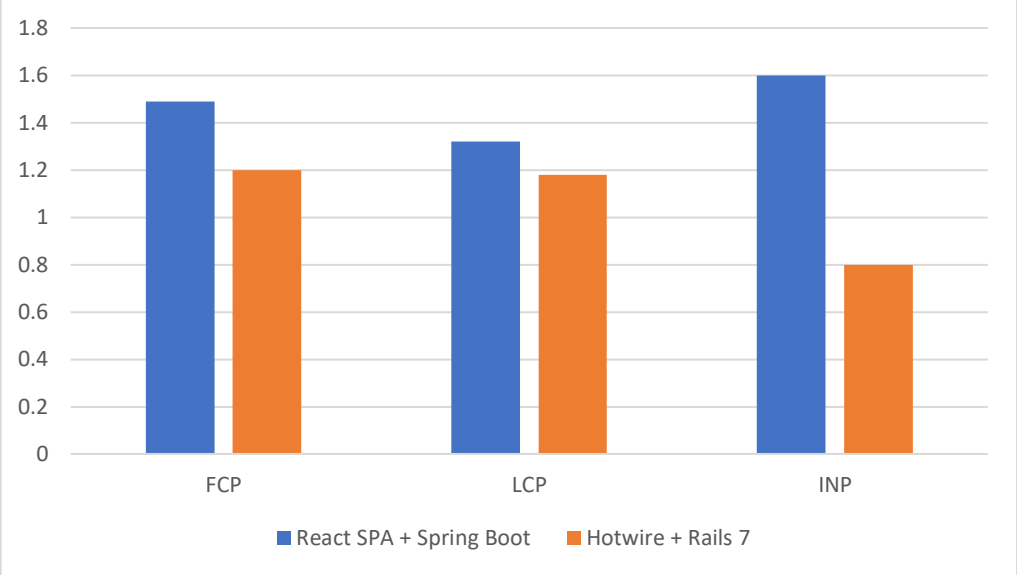

Рисунок 4.7. Порівняння результатів завантаження застосунків з очищенням кешу браузера та активованою опцією "Disable cache"

Окреме тестування було присвячене взаємодії зі сторінкою статистики: після входу користувач переходить до перегляду статистики (як глобальної, так і на кожну дошку). У React SPA перегляд реалізовано через модальне вікно, тоді як у Hotwire - шляхом повного переходу на окрему сторінку з повторним рендерингом.

Основні показники:

- Загальний час сценарію: Hotwire - 3.77 с, React - 2.14 с
- Scripting: 721 мс проти 549 мс
- Rendering: 178 мс проти 76 мс
- DOM-вузли: 37,145 проти 2,517
- JS Heap приріст: +14 MB (в обох випадках)

На рисунку 4.8 зображено порівняння метрик при профілюванні на обох застосунках.

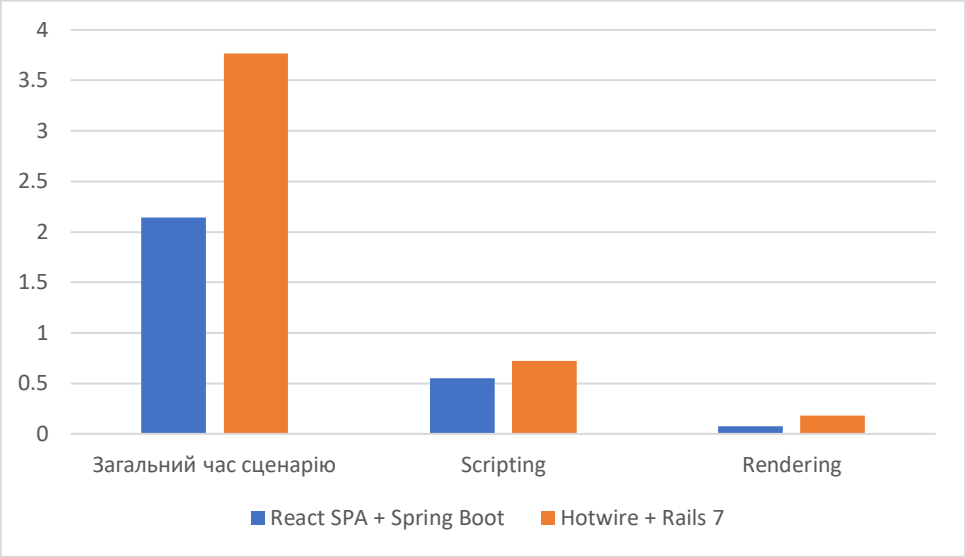

Рисунок 4.8. Порівняння часу взаємодії та рендерингу для сценаріїв "Вхід -> Перегляд статистики"

Ці результати підтверджують загальну тенденцію: архітектура SPA на React краще справляється з частковими оновленнями даних та має нижчий загальний час виконання дій, тоді як Hotwire потребує повного рендерингу, що призводить до вищого навантаження на DOM та тривалішого scripting.

Варто зазначити, що частина спостережених проблем з продуктивністю в Hotwire-реалізації, зокрема відсутність миттєвого оновлення DOM після видалення задачі, могла бути спричинена запуском застосунку у production-режимі без додаткового відлагодження механізмів Turbo Stream. У development-середовищі інтерфейс працював набагато плавніше, хоча враження від використання все ж залишалося трохи відмінним від React SPA: перемикання між сторінками відбувалося з короткими затримками, обумовленими повним перезавантаженням фреймів.

У React SPA кожна дія - від редагування задач до перегляду статистики - виконувалась миттєво, завдяки клієнтській маршрутизації та локальному оновленню стану. Видалення задачі відбувалося одразу після запиту, без перезавантаження, тоді як у Hotwire інтерфейс у production вимагав ручного оновлення сторінки, що створювало враження меншої плавності.

Загалом, React SPA виявився ефективнішим у складних та динамічних взаємодіях, тоді як Hotwire добре показує себе при простих сценаріях, зберігаючи дуже низьку латентність після початкового рендерингу.

Однак у деяких сферах, зокрема у малих e-commerce проєктах, client-side rendering в цілому часто демонструє слабкі метрики продуктивності (високі LCP, CLS, TTI), що негативно впливає на SEO та конверсію. Це підтверджено в дослідженні Kroon Celander & Möllestål (2024) [2], і саме тому для таких проєктів зростає інтерес до SSR або SSG-підходів.

4.3. Продуктивність розробника

Цей розділ аналізує ефективність, зручність та масштабованість роботи розробника в межах двох підходів: React SPA + Spring Boot та Hotwire + Rails 7. Порівняння базується на чотирьох аспектах: залежності, швидкість змін, навчальна крива, і архітектурна прозорість.

4.3.1. Залежності та розмір коду

Кількість і складність зовнішніх залежностей безпосередньо впливає на підтримуваність, безпеку та швидкість розробки застосунку. У цьому підрозділі проведено аналіз залежностей і структури коду в обох реалізаціях.

React + Spring Boot

Frontend-частина React має на перший погляд невелику кількість top-level залежностей у package.json - лише 4 основні бібліотеки (react-i18next, i18next, ghc, i18next-browser-languagedetector). Проте реальна картина залежностей

розкривається через node_modules, яка займає понад 461 MB - типовий обсяг для сучасного JavaScript-додатку навіть із мінімальним стеком.

Backend-частина (Spring Boot) виявилась значно складнішою за обсягом та структурою:

- Команда ./mvnw dependency:tree показала 742 залежності, включно з транзитивними.
- Власний код Java налічує понад 1,460 рядків (cloc src/).
- Проєкт має типову для Spring конфігурацію через application.properties, а також pom.xml, що сам по собі містить десятки залежностей, плагінів build і профілів.

Таким чином, повна збірка застосунку передбачає запуск декількох окремих компіляційних етапів (frontend build, Maven package), а загальна кількість стороннього коду обчислюється сотнями тисяч рядків. Подібна глибина створює ризики "dependency hell" - ситуацій, де оновлення однієї бібліотеки призводить до збоїв через конфлікти або вразливості в інших.

Це підтверджують реальні випадки, як-от видалення бібліотеки npm left-pad у 2016 році, що спричинило масовий збій тисяч пакетів у екосистемі JavaScript [24]. Така централізована залежність від великої кількості нестабільних сторонніх пакетів створює технічний борг і знижує передбачуваність середовища, а також вимагає від розробників постійно стежити за оновленнями та змінами у використовуваних залежностях і бібліотеках, щоб жодні оновлення не порушували процес розробки.

Hotwire + Rails 7

Застосунок на Rails має лаконічну структуру з мінімальною кількістю нестабільних залежностей. Було виявлено понад 120 гемів (за gem list), більшість

із яких є частиною офіційного стеку Rails або добре підтримуваними рішеннями open source (наприклад, devise, stimulus-rails, hotwire-rails, pg).

За результатами команди rails stats, загальна кількість рядків коду в основних компонентах (моделі, контролери, представлення, тощо) становить 1,096 LOC (стрічок коду). Основне навантаження припадає на шаблони (661 LOC), що очікувано у серверно-орієнтованій архітектурі з рендерингом HTML.

Кодова структура:

- Контролери: 224 LOC, 7 класів, 29 методів
- Моделі: 33 LOC, 5 класів, 2 методи
- В'юшки: 661 LOC (більшість у шаблонах ERB)
- Код JavaScript: лише 26 LOC у каталозі app/javascript, що ілюструє залежність Hotwire від серверної логіки

Архітектура відповідає класичній структурі Rails: app/controllers, app/models, app/views, із мінімальною кастомізацією. Більшість логіки реалізована у межах стандартного фреймворку без складної конфігурації або webpack-based білдів.

Високий ступінь інтеграції Hotwire + Rails дозволяє реалізувати динамічні інтерфейси без складних залежностей у фронтенді. Наприклад, графіки реалізовані через chartkick, а StimulusJS контролери розташовані локально в app/javascript/controllers. Загальна кількість рядків JavaScript становила ~29,000 - переважно через бібліотеку візуалізації.

4.3.2. Час запуску та редагування

Час між редагуванням коду та відображенням змін у застосунку критично впливає на ефективність розробки - особливо на ранніх етапах створення інтерфейсу та логіки.

Spring Boot у поєднанні з React вимагає додаткової координації під час розробки, оскільки клієнт і сервер функціонують як окремі застосунки. Зазвичай необхідно запускати два сервери - один для React, інший для Spring Boot, і для обох потрібно окремо налаштовувати порти (localhost:3000 vs localhost:8080). Це ускладнює інтеграційне тестування і подовжує цикл “редагування - перевірка”.

Було проведено заміри запуску Spring Boot за допомогою команди time ./mvnw spring-boot:run:

- Початковий запуск: ~32.8 с
- Повторний запуск після редагування коду: ~28.4 с

Хоча існують інструменти гарячого перезавантаження (наприклад, Spring DevTools або JRebel), їх конфігурація потребує окремих кроків, і не всі зміни (зокрема зміни у структурі класів) застосовуються миттєво.

У порівнянні, Hotwire + Rails 7 надає значно швидший цикл зворотного зв’язку. Заміри запуску сервера у режимі розробки (time bin/rails s) засвідчили:

- Перший запуск: ~9.88 с
- Редагування шаблону та повторний запуск: ~8.4 с

У більшості випадків Rails не потребує навіть цього: зміни в шаблонах ERB, Turbo Stream або контролерах, і загалом будь-які зміни крім у config/ і додавання нових гемів застосовуються автоматично, без перезапуску сервера. Це дозволяє практично в реальному часі бачити результати змін у браузері.

Також варто врахувати загальний розробницький цикл:

- Rails не вимагає збірки фронтенду - уся інтеграція виконується через importmap або jsbundling-rails (у цьому випадку - лише для CSS).
- Spring Boot у поєднанні з React потребує повної збірки фронтенду через Vite або Webpack та окремого запуску бекенду.

Rails 7 із Hotwire значно швидше реагує на зміни в коді, завдяки hot-reload шаблонів, моделей і логіки. Spring Boot з React продуктивний у масштабних проєктах, але ускладнює швидке прототипування через довгий час збирання і запуску.

4.3.3. Крива навчання та документація

Оцінка кривої навчання охоплює не лише перші кроки у використанні фреймворку, але й час, необхідний для глибшого розуміння, налагодження складних сценаріїв, масштабування проєкту та ефективної роботи з екосистемою. Важливими критеріями є доступність якісної документації, зрозумілість інструментів та передбачуваність поведінки фреймворку.

React + Spring Boot

React має одну з наймасштабніших і найактивніших екосистем у серед фреймворків на фронтенд. Репозиторій GitHub фреймворку має понад 236 тис. зірок і 1,600+ дописувачів [25] а Spring Boot - 77.2 тис. зірок, 1,156 дописувачів [26], що свідчить про зрілість, активну підтримку та велику кількість навчальних ресурсів.

Однак повноцінне використання React вимагає засвоєння великого обсягу знань:

- JSX, компонентна модель, props/state, React-хуки (useEffect, useState).
- Бібліотеки маршрутизації (React Router), управління станом (Redux, Zustand), форм (Formik, React Hook Form).
- Tooling: Vite/Webpack, Babel, ESLint, npm/yarn, конфігурації CI/CD.

Окремий виклик становила інтеграція з бекендом - автентифікація через JWT, CORS, проксі, кешування, налаштування середовищ.

React є надзвичайно потужним та гнучким інструментом, але його повне освоєння може зайняти місяці або навіть роки, особливо для розробників без досвіду в SPA-архітектурах.

Hotwire + Rails 7

Hotwire пропонує мінімалістичний підхід до створення динамічного інтерфейсу без використання SPA. Інтерфейс формується на сервері й оновлюється за допомогою передачі фрагментів HTML через Turbo.

Ключові переваги:

- Значно нижчий поріг входу: немає необхідності вивчати фреймворки JavaScript чи збирати frontend окремо.
- Базова інтеграція Stimulus дозволяє реалізовувати окремі сценарії без складної логіки.
- Офіційна документація зрозуміла та орієнтована на швидкий старт.

Однак на глибшому рівні з’являється "магія Rails", яка ускладнює розробку. Turbo автоматично оновлює частини DOM без явного коду, що спрощує старт, але ускладнює діагностику. Потім, побічні ефекти, як-от кешування, подвійне надсилання форм, або незрозуміла поведінка при вкладеності Turbo Frame, потребують дослідження GitHub-дискусій або навіть вихідного коду.

Репозиторії Hotwire на GitHub набагато менш активні:

- Turbo - 7,000 зірок, ~100 дописувачів [27]
- Stimulus - 12,900 зірок, ~90 дописувачів [28]

… що свідчить про меншу кількість матеріалів, обговорень та рішень для граничних сценаріїв.

Також важливим чинником є зручність пошуку та виправлення помилок.

У React помилки зазвичай добре видно у консолі браузера або DevTools, часто з повним стеком викликів, що дозволяє швидко локалізувати проблему. Інструменти на кшталт React Developer Tools чи ESLint додатково полегшують аналіз помилкових станів або помилок у логіці рендеру.

У Hotwire та Rails Turbo налагодження часто є непростим - особливо у складних сценаріях рендеру. Однією з проблем, виявлених під час розробки, стала спроба переадресувати користувача після входу на сторінку з дошками. У певних конфігураціях Turbo помилково намагався інтерпретувати об’єкт, який відповідав за користувача, як відповідь HTML, що спричиняло помилку без чіткого опису. Це сталося під час спроби перенаправлення користувача після входу - типова поведінка, яка в React реалізовується явно і без подібних труднощів.

React виявився більш передбачуваним у роботі та налагодженні, особливо завдяки прозорій маршрутизації, зрозумілим помилкам та великій кількості інструментів аналізу. У Hotwire навіть базова функціональність, як-от переадресація або обробка вкладених Turbo Frame, іноді поводиться неочікувано, що потребує глибшого розуміння внутрішніх механізмів фреймворку. Така "магія" знижує прозорість і вимагає більше часу на тестування.

У результаті, хоча Hotwire швидше дозволяє реалізувати базову функціональність, React забезпечує більш передбачувану, керовану і діагностовану розробку в масштабних сценаріях.

РОЗДІЛ 5. ВИСНОВКИ.

5.1. Підсумок результатів

У межах цього дослідження було створено два повнофункціональні веб-застосунки - один із використанням React + Spring Boot, інший на основі Hotwire + Rails 7 - з ідентичною функціональністю (канбан-дошки, створення/редагування задач, перегляд статистики). Обидва застосунки пройшли профілювання в реальних сценаріях використання, включно з тестуванням продуктивності (Lighthouse, wrk, curl), аналізом поведінки UX (Chrome DevTools), та оцінкою продуктивності розробника.

Основні результати:

- React + Spring Boot стабільно демонструє кращу продуктивність у складних інтерфейсах, завдяки Virtual DOM, гнучкому управлінню станом і клієнтській маршрутизації.
- Hotwire + Rails 7 забезпечує швидкий старт розробки, просту архітектуру та високу продуктивність у простих сценаріях, але має певні обмеження в складних інтерактивних кейсах.
- Час запуску та оновлення змін у Hotwire у 2–3 рази коротший, ніж у Spring Boot, що суттєпо пришвидшує цикл розробки / відладки / тестування застосунку.
- Кількість залежностей та технічний борг значно вищі у React/Spring Boot через npm + Maven, що знижує стабільність і передбачуваність середовища.
- React має вищий поріг входу, але і значно ширший спектр архітектурних можливостей та інструментів для діагностики.

5.2. Варіанти використання та найкращі сценарії

В залежності від цілей і масштабу проєкту, обидва підходи мають свої переваги.

Hotwire + Rails 7 найкраще підходить для:

- Малих і середніх команд, які бажають швидко створити MVP або внутрішній інструмент (напр. при роботі з більшою системою).
- Розробників з досвідом у Ruby або Rails, яким важлива висока продуктивність без складного стеку JavaScript.
- Проєктів, де в пріоритеті швидкий старт, мінімум шаблонного коду та мінімум технічного боргу.
- Випадків із простою логікою фронтенду, де прийнятний повний перерендер DOM.

React + Spring Boot краще підійде для:

- Великих, динамічних або масштабованих веб-застосунків із складною структурою інтерфейсу.
- Команд, яким потрібна чітко декомпозована структура компонентів.
- Проєктів з великою кількістю взаємодій, асинхронною логікою, або складним управлінням станом.
- Застосунків, які будуть активно підтримуватись, масштабуватись, або передаватись іншим командам.

5.3. Остаточний вердикт: Який підхід є кращим?

Однозначної відповіді немає - все залежить від контексту. Але результати дослідження дозволяють зробити такі висновки:

- React + Spring Boot - це технології з глибокою екосистемою, активною підтримкою та високим рівнем гнучкості. Вони дозволяють створювати масштабовані, інтерактивні застосунки зі складною логікою, але вимагають більшого часу на розгортання, навчання й підтримку.
- Hotwire + Rails 7, хоча й менш популярні і ще недостатньо вивчені у великій кількості проєктів, продемонстрували високий рівень продуктивності в невеликих проєктах. Їхній підхід із мінімальним JavaScript, швидким циклом розробки та глибокою інтеграцією в

екосистему Rails робить їх чудовим варіантом для малих команд і MVP-проєктів.

Таким чином, Hotwire – це не альтернатива React, а окрема парадигма, яка за певних умов може перевершити класичний підхід SPA за зручністю та швидкістю розробки. Її головна слабкість - недостатня поширеність та невелика кількість навчального матеріалу - може зменшитися в майбутньому, враховуючи зростаючий інтерес до технології.

5.4. Майбутні перспективи та напрямки досліджень

У майбутньому доцільно провести подальші дослідження з акцентом на такі напрями:

- Масштабованість Hotwire у складних інтерфейсах (включно з проблемами при великій кількості Turbo Frame).
- Системне порівняння підходів SSR/SPA/MPA у продуктивності на хмарних платформах (напр. Heroku, Fly.io, Vercel).
- Порівняння технічного боргу через аналіз оновлюваності залежностей, уразливостей безпеки та стабільності білдів.
- UX-дослідження: суб'єктивне сприйняття взаємодії в Hotwire і React SPA, включаючи доступність, адаптивність, мобільну продуктивність.

5.5. Оцінка методології та обмежень дослідження

Хоча дослідження дало деякі практичні результати, що порівнюють архітектурні підходи (SPA на основі React + Spring Boot та серверний рендеринг за допомогою Hotwire + Rails 7), варто критично оцінити його методологічні обмеження, які потенційно можуть вплинути на глибину висновків.

5.5.1. Неповна архітектурна паритетність

Хоча обидві реалізації намагалися досягти однакової функціональності, було вирішено використовувати Spring Boot як бекенд для React SPA, тоді як Hotwire

базувався на повному стеку Rails. Це рішення дозволило дослідженню зосередитися на порівнянні рішень, які були популярними в їхніх середовищах, але воно завадило глибшому аналізу відмінностей між парадигмами фронтенду в одному серверному середовищі. Використання Rails як бекенду для React (що також є популярним рішенням серед розробників) зменшило б кількість змінних і чіткіше визначило б різницю між підходами до рендерингу.

5.5.2. Обмежений обсяг тестування

Дослідження зосереджувалося на технічній продуктивності (Lighthouse, wrk, curl), UX через DevTools та загальному досвіді розробника (DX). Однак, було б корисно включити розширене тестування під час фази розгортання, тестування мобільної поведінки, аналіз доступності та, якщо можливо, суб'єктивні оцінки користувачів. Це забезпечило б більш багатовимірне уявлення про те, як програми поводяться в реальному світі.

5.5.3. Вплив конкретних рішень щодо реалізації

Під час розробки програми на Hotwire Stimulus навмисно було не використано, що спростило кодову базу, але обмежило масштабованість динамічної логіки та, можливо, погіршило UX у складніших сценаріях. Аналогічно, React SPA не реалізував деякі додаткові оптимізації, такі як lazy-loading, розділення коду або SSR через Next.js, що потенційно могло вплинути на результати порівняння.

5.5.4. Оцінка UX одним користувачем

Оцінка користувацького досвіду проводилася суб'єктивно автором дослідження на основі технічних метрик (INP, вузли DOM, адаптивність). Тестування застосунків з реальними користувачами не проводилося, що обмежує можливість говорити про повноцінну оцінку UX.

Незважаючи на перелічені вище обмеження, дослідження досягло своєї головної мети - показати архітектурні відмінності в підході до створення веб-застосунків на основі двох радикально різних парадигм. Однак згадані аспекти можуть слугувати відправною точкою для майбутніх досліджень, які розширять

емпіричну базу та охоплять більше факторів, середовищ та сценаріїв використання.

Список використаних джерел

1. Meredova, A. *Comparison of Rendering Techniques in JavaScript Frameworks.* Theseus, 2023. [Електронний ресурс]. - Режим доступу: https://www.theseus.fi/bitstream/handle/10024/799997/Meredova_Aylar.pdf?sequence=2
2. Kroon Celander, E., Möllestål, A. *A Comparative Analysis of Next.js, SvelteKit, and Astro for E-commerce Web Development.* DIVA Portal, 2024. [Електронний ресурс]. - Режим доступу: https://www.diva-portal.org/smash/get/diva2:1869595/FULLTEXT01.pdf
3. Stack Overflow. *Stack Overflow Developer Survey 2024.* [Електронний ресурс]. - Режим доступу: https://survey.stackoverflow.co/2024/technology#most-popular-technologies
4. Hansson, D. H. *Rails 7: Fulfilling a Vision.* Ruby on Rails, 2021. [Електронний ресурс]. - Режим доступу: https://rubyonrails.org/2021/12/15/Rails-7-fulfilling-a-vision
5. Basecamp. *HEY and Hotwire: A New Frontier.* 2020. [Електронний ресурс]. - Режим доступу: https://www.hey.com
6. Wu, Y. *Research into Pre-rendering Technology for Supporting Modern Websites.* Metropolia UAS, 2021. [Електронний ресурс]. - Режим доступу: https://bpb-us-e1.wpmucdn.com/sites.uw.edu/dist/8/6647/files/2021/06/ChrisWu_Final_Report.pdf
7. Mukhamadiev, A. *Transitioning from server-side to client-side rendering of the web-based user interface: a performance perspective.* Theseus, 2018. [Електронний ресурс]. - Режим доступу: https://www.theseus.fi/bitstream/handle/10024/156808/Mukhamadiev_Aidar.pdf
8. Alekseeva, D. *Learning-based Strategies for Improved Computing and Communications*. Trepo Repository, 2024. [Електронний ресурс]. - Режим доступу: https://trepo.tuni.fi/bitstream/handle/10024/161137/978-952-03-3681-3.pdf

9. Vercel. (n.d.). *Streaming and Server Components in Next.js*. https://nextjs.org/docs/app/building-your-application/rendering/
10. State of JS 2022 Survey. https://2022.stateofjs.com/en-US/libraries/rendering-frameworks/
11. Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine. [Електронний ресурс]. Режип доступу - https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
12. Crockford, D. (2006). *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 8259. [Електронний ресурс]. Режип доступу - https://tools.ietf.org/html/rfc8259
13. Meta. (n.d.) *Virtual DOM and Internals - React legacy documtentation.* https://legacy.reactjs.org/docs/faq-internals.html
14. Remix Software Inc. (n.d.). *React Router documentation*. https://reactrouter.com
15. Redux.js. (n.d.). *Redux Essentials, Part 1*. https://redux.js.org/tutorials/essentials/part-1-overview-concepts
16. Poimandres. (n.d.). *Zustand documentation*. https://docs.pmnd.rs/zustand/
17. Hotwired.dev. (n.d.). *Turbo Drive - Navigation Without the JavaScript*. https://turbo.hotwired.dev/handbook/drive
18. Hotwired.dev. (n.d.). *Turbo Streams*. https://turbo.hotwired.dev/handbook/streams
19. Stimulus. (n.d.). *Stimulus Handbook*. https://stimulus.hotwired.dev/handbook/introduction
20. Meta. (n.d.). *Quick Start - React documentation*. https://react.dev/learn
21. Facebook Engineering. (2015). *GraphQL: A data query language.* https://engineering.fb.com/2015/09/14/core-infra/graphql-a-data-query-language/
22. Airbnb Engineering. (2017). *React Performance Fixes on Airbnb Listing Pages.* https://medium.com/airbnb-engineering/recent-web-performance-fixes-on-airbnb-listing-pages-6cd8d93df6f4

23. Airbnb Engineering. (2024). *How Airbnb Smoothly Upgrades React.* https://medium.com/airbnb-engineering/how-airbnb-smoothly-upgrades-react-b1d772a565fd
24. npm, Inc. (2016). *kik, left-pad, and npm*. [Електронний ресурс]. Режип доступу - https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm
25. GitHub. *facebook/react: The library for web and native user interfaces.* [Електронний ресурс]. Режим доступу - https://github.com/facebook/react
26. GitHub. *spring-projects/spring-boot: Spring Boot helps you to create Spring-powered, production-grade applications and services with absolute minimum fuss.* [Електронний ресурс]. Режим доступу - https://github.com/spring-projects/spring-boot
27. GitHub. *hotwired/turbo: The speed of a single-page web application without having to write any JavaScript*. [Електронний ресурс]. Режим доступу - https://github.com/hotwired/turbo
28. GitHub. *hotwired/stimulus: A modest JavaScript framework for the HTML you already have.* [Електронний ресурс]. Режим доступу - https://github.com/hotwired/stimulus