КВАЛІФІКАЦІЙНА РОБОТА

Група ІПЗс-2019

Манастирецький Л.В.

2023

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Манастирецький Любомир Васильович

УДК 004.378

Розробка веб-сайту агрегатора новин з елементами веб-скрапінгу засобами Ruby on Rails

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації бакалавра

Нормоконтроль

_____Стисло О.В.

(підпис, дата, розшифрування підпису)

Студент

_____Манастирецький Л.В.

(підпис, дата, розшифрування підпису)

Допускається до захисту

Завідувач кафедри

_____к.т.н., доц. Пашкевич О.П.

(підпис, дата, розшифрування підпису)

Керівник роботи

_____ к.т.н., доц. Ващишак С.П.

(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2023

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «бакалавр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

« ___ » _____ 2023 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Манастирецький Любомир Васильович

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Розробка веб-сайту агрегатора новин з елементами веб-скрапінгу засобами Ruby on Rails

керівник роботи:

Ващишак Сергій Петрович, кандидат технічних наук

затверджена наказом вищого навчального закладу від « 11» листопада 2022 року № 155/1НВ

2. Термін подання студентом роботи 14.06.2023

3. Вихідні дані роботи: Мова програмування Ruby, Ruby on Rails, Мова програмування JavaScript

4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)

1. Опис та порівняння існуючих аналогів

2. Проектування архітектури та план розробки додатку

3. Програмна реалізація веб-сайту

5. Дата видачі завдання 10.10.2022

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Розділ	Консультант (прізвище, ініціали та посада)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Дослідження існуючих аналогів	27.03.2023	Виконано
2.	Проектування архітектури та створення дизайну веб-сайту	25.04.2023	Виконано
3.	Програмна реалізація веб-сайту	8.05.2023	Виконано
4.	Розгортання та тестування веб-сайту	15.05.2023	Виконано
5.	Формування висновків	23.05.2023	Виконано
6.	Оформлення пояснювальної записки	29.05.2023	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	01.06.2023	Виконано

Студент _____ Манастирецький Л.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Ващишак С.П.
(підпис) (прізвище та ініціали)

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу
14	Головна сторінка RubyWeekly	25	Дизайн сторінки "Архів"
16	Головна сторінка LibHunt Ruby	26	Приклад генерації зображення з Midjourney
22	Архітектура rails.news	55	Сторінка зібраних постів в адмін панелі
24	Компонент Figma "карточка з новиною"	60	Інтерфейс Lookbook
24	Логотип веб-сайту на темному фоні	62	Головна сторінка готового сайту

АНОТАЦІЯ

Кваліфікаційна робота присвячена вивченню, аналізу та розробці веб-сайту агрегатора новин для спільноти Ruby і Ruby on Rails з метою оптимізації процесів відбору та публікації інформації.

В першому розділі проведено аналіз веб-сайтів агрегаторів новин, надано загальний опис майбутнього веб-сайту “rails.news” та проведено аналіз існуючих аналогів - RubyWeekly та LibHunt Ruby. Виявлено їхні переваги та недоліки.

У другому розділі розглянуто особливості сучасних інструментів веб-скрапінгу у Ruby, розроблено архітектуру для веб-сайту “rails.news”, а також макет. Проведено вибір технологій реалізації та розгортання.

В третьому розділі описано програмну реалізацію веб-сайту агрегатора новин, починаючи від створення базової аплікації та налаштування тестового середовища, і закінчуючи вводом аплікації в експлуатацію.

В результаті, було створено повноцінний веб-сайт, що забезпечує ефективний збір, організацію та відображення новин для спільноти Ruby і Ruby on Rails, з використанням веб-скрапінгу.

КЛЮЧОВІ СЛОВА: WEB-СКРАПІНГ, RUBY, RUBY ON RAILS, MRSK, VIEWCOMPONENT, АГРЕГАТОР НОВИН, WEB-ДИЗАЙН.

SUMMARY

This qualifying work is dedicated to the examination, analysis, and development of a news aggregator website intended for the Ruby and Ruby on Rails community. The aim is to optimize the processes of information selection and publication.

In the first section, an exploration of news aggregator websites is carried out, providing a general description of the future website "rails.news". There is also an analysis of existing counterparts - RubyWeekly and LibHunt Ruby - with their respective advantages and disadvantages identified.

The second section considers the features of modern web scraping tools in Ruby, and develops an architecture for the website "rails.news". It also explains the design, and decisions regarding the technologies used for implementation and deployment.

The third section describes the software implementation of the news aggregator website, from the creation of a basic application and the configuration of a test environment, through to the final deployment of the application.

As a result, a fully operational website was created, effectively collecting, organizing, and displaying news for the Ruby and Ruby on Rails community, leveraging web scraping techniques.

KEYWORDS: WEB SCRAPING, RUBY, RUBY ON RAILS, MRSK, VIEWCOMPONENT, NEWS AGGREGATOR, WEB DESIGN

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних

Продакшн – виробниче середовище

Гем – загальноприйнята назва для бібліотек написаних на Ruby

Хедлес браузер – це програма для веб-перегляду, яка працює без відображення графічного інтерфейсу користувача

Колбек – це метод, який автоматично викликається в певний момент життєвого циклу об'єкта моделі

Міграція – це процес зміни структури бази даних у Ruby on Rails, який дозволяє додавати, видаляти або змінювати таблиці і поля за допомогою написання коду замість безпосереднього виконання SQL-запитів

Екшн – у контексті Ruby on Rails відноситься до методів контролера, які реагують на запити HTTP

Паршиал – це файл виду, який містить частину коду HTML, який може бути використаний в кількох місцях

Моноліт – це структура веб-додатку, де всі компоненти системи (інтерфейс користувача, бізнес-логіка, обробка даних) інтегровані в одну єдину кодову базу

Мокап – в контексті веб-розробки це детальний проект або модель, який демонструє функціональність системи або продукту

ВСТУП

Актуальність теми: У сучасному світі, де інформаційні потоки набули неймовірної швидкості та обсягу, проблема ефективного вибору та представлення актуальної інформації стає особливо важливою. Інтернет-ресурси щодня генерують тисячі новин, що містять величезну кількість важливої та корисної інформації. Проте, для звичайного користувача часто важко впоратися з цим великим потоком інформації, відфільтрувати її та виокремити найважливіше.

Розробка веб-сайту агрегатора новин, який використовує технологію веб-скрапінгу, засобами Ruby on Rails вирішує цю проблему. Використовуючи веб-скрапінг, можна автоматично збирати інформацію з різних веб-ресурсів та агрегувати її в одному місці, що забезпечує зручність та ефективність використання.

Тема актуальна також через широке застосування Ruby on Rails в розробці веб-додатків. Це гнучкий та потужний інструмент, який дозволяє швидко розробляти надійні та безпечні веб-додатки.

Мета і завдання дослідження: розробити веб-сайт агрегатор новин з використанням веб-скрапінгу на основі Ruby on Rails.

Для досягнення поставленої мети вирішуються наступні завдання:

- провести аналіз сучасних методів та інструментів веб-скрапінгу;
- розробити архітектуру веб-сайту агрегатора новин;
- реалізувати сервіс веб-скрапінгу для автоматичного збору новин з різних веб-ресурсів;
- інтегрувати сервіс веб-скрапінгу в архітектуру веб-сайту;
- провести тестування розробленого веб-сайту.

Об’єкт дослідження: процес розробки веб-сайту агрегатора новин з використанням технології веб-скрапінгу на основі Ruby on Rails.

Предмет дослідження: методи та інструменти веб-скрапінгу, їх інтеграція в архітектуру веб-сайту, а також ефективність розробленого веб-сайту-агрегатора новин.

Методи дослідження: Для досягнення поставленої мети та вирішення завдань дослідження будуть використані наступні методи:

- метод аналізу - для проведення аналізу сучасних методів та інструментів веб-скрапінгу, а також аналізу результатів роботи веб-сайту та його ефективності;
- метод проектування - для розробки архітектури веб-сайту агрегатора новин;
- метод програмування на Ruby on Rails - для реалізації модулю веб-скрапінгу та інтеграції його в архітектуру веб-сайту;
- метод тестування - для перевірки коректності роботи розробленого веб-сайту.

Ці методи дослідження допоможуть ретельно вивчити обраний предмет, розробити ефективний веб-сайт агрегатор новин з використанням веб-скрапінгу на основі Ruby on Rails, а також оцінити його ефективність.

Практичне значення одержаних результатів: Результати даного дослідження, зокрема розроблений веб-сайт агрегатор новин з використанням веб-скрапінгу на основі Ruby on Rails, відкривають значні можливості для автоматизованого збору та агрегації новин з різних веб-ресурсів, спрощуючи доступ до актуальної інформації. Крім практичного застосування в роботі компаній, що займаються розробкою програмного забезпечення, дані результати можуть стати основою для подальших наукових робіт, навчання та підвищення кваліфікації в області веб-скрапінгу та розробки веб-додатків.

Структура: Розділи – 3. Загальний обсяг основної частини – 70 сторінок. Список використаних джерел – 25.

РОЗДІЛ 1. ОПИС ТА ПОРІВНЯННЯ ІСНУЮЧИХ АНАЛОГІВ

1.1 Аналіз веб-сайтів агрегаторів новин

Веб-сайти агрегатори новин виступають в ролі центральних вузлів для збору, класифікації, відбору і розповсюдження інформації з різноманітних джерел [1]. Вони працюють як потужні інструменти для автоматизованого збору, організації і представлення даних з численних веб-сайтів у вигляді, що зручний для використання та перегляду. Це значно полегшує доступ до потрібної інформації, оскільки вона зібрана і структурована в одному місці.

У контексті технології Ruby on Rails та мови програмування Ruby, веб-сайти агрегатори новин можуть стати незамінними інструментами для розробників. Вони можуть забезпечувати збір і надання актуальних новин, оновлень, інформації про нові версії бібліотек та фреймворків, а також інших релевантних ресурсів. Такі агрегатори можуть включати інформацію про майбутні події, такі як конференції, воркшопи, вебінари та інше, що стосується спільноти Ruby on Rails та Ruby.

Завдяки таким веб-сайтам, розробники отримують зручний доступ до найновіших новин та ресурсів, що дозволяє їм бути в курсі найсвіжіших трендів та оновлень в своїй сфері. Це може включати огляди нових версій бібліотек та фреймворків, статті та блоги від провідних експертів в галузі, навчальні матеріали та ресурси для самостійного вивчення, а також інформацію про майбутні події та конференції.

Однак, потрібно врахувати, що деякі з існуючих веб-сайтів агрегаторів новин можуть мати деякі недоліки, такі як нерегулярні оновлення, застарілий дизайн та незручний інтерфейс користувача. Отже, виникає потреба в розробці нового, сучасного веб-сайту агрегатора новин,

який би вирішив ці проблеми та відповідав би сучасним стандартам і вимогам користувачів.

1.2 Загальний опис веб-сайту “rails.news”

Веб-сайт "rails.news" планується як сучасний, добре структурований агрегатор новин, орієнтований на технологію Ruby on Rails та мову програмування Ruby. Він призначений для збору, категоризації та представлення актуальних новин та інформації з різних джерел в удобній і зрозумілій формі.

Основною функцією веб-сайту "rails.news" є представлення останніх новин. Він автоматично збирає останні новини, оновлення, анонси про нові версії бібліотек та фреймворків, інформацію про майбутні події та інше. Ця інформація відображається на головній сторінці сайту у вигляді добре організованого потоку новин.

Веб-сайт також має розділ "Архів", де користувачі можуть переглянути старіші новини та ресурси. Це може бути корисно для користувачів, які шукають специфічну інформацію або хочуть ознайомитися з історією певних подій, тенденцій або оновлень.

Щоб забезпечити прозорість та довіру користувачів, на веб-сайті також передбачено розділ "Про нас". Тут користувачі можуть знайти інформацію про мету сайту, його розробників та способи збору новин.

Нарешті, на веб-сайті є форма для зв'язку, яка дозволяє користувачам зв'язатися з адміністрацією сайту. Це може бути корисно для зворотного зв'язку, запитів щодо співпраці, або в разі виникнення питань або проблем.

Таким чином, "rails.news" розробляється як всеосяжний веб-ресурс для спільноти Ruby on Rails, який надає актуальну, цінну та своєчасну інформацію в зручному та легко доступному форматі. Веб-сайт відображає принципи користувацького дизайну, орієнтованого на користувача, інтуїтивно зрозумілого навігації та простого використання.

Він має бути створений з використанням найкращих практик в області веб-розробки, включаючи мобільну адаптацію, оптимізацію SEO та високу продуктивність. Крім того, веб-сайт "rails.news" розробляється з метою забезпечення високої ступені доступності та відповідності стандартам безпеки.

Використовуючи сучасні технології та методології розробки, "rails.news" прагне стати важливим ресурсом для всієї спільноти Ruby on Rails. Його метою є створення платформи, яка допоможе розробникам легко знаходити актуальну інформацію, навчальні матеріали, нові тенденції та багато іншого, що стосується Ruby on Rails та мови програмування Ruby.

1.3 Огляд та виявлення недоліків в існуючих аналогах

Існує кілька веб-сайтів, які вже займаються агрегацією новин, пов'язаних з Ruby on Rails та Ruby. До таких веб-сайтів належать: RubyWeekly та LibHunt Ruby.

RubyWeekly (https://rubyweekly.com/) – це веб-сайт та щотижнева розсилка, яка фокусується на новинах та оновленнях у світі Ruby. Хоча він має деякий функціонал розсилки, RubyWeekly все ж є агрегатором новин, оскільки він збирає та категоризує новини з різних джерел.

Головна сторінка веб-сайту RubyWeekly представляє собою інформативний та структурований ресурс, що включає різноманітні посилання та матеріали, які пов'язані з Ruby on Rails та мовою Ruby (рис. 1.1).

У верхній частині сторінки розташований блок з посиланням на найновішу розсилку. Це дозволяє відвідувачам сайту легко переглянути останні новини та оновлення у світі Ruby. Відвідувачам пропонується натиснути на посилання, щоб переглянути повний список новин та матеріалів, включених у найновішу розсилку.

Біля цього посилання знаходиться розділ з посиланнями на попередні розсилки. Він містить архів розсилок, що дозволяє користувачам прослідкувати та переглянути попередні випуски. Це забезпечує доступ до більшого обсягу новин та ресурсів, ніж тільки найновіша розсилка.

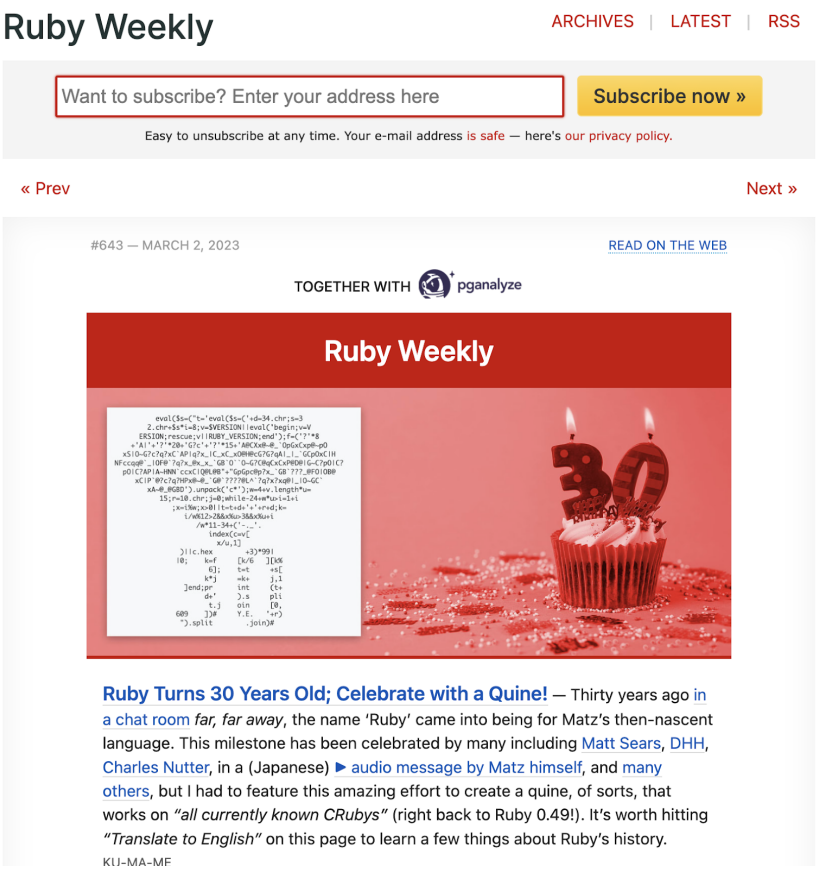

Рисунок 1.1 – Головна сторінка RubyWeekly

Відвідувачі сайту також мають можливість підписатися на розсилку RubyWeekly. На сторінці є форма підписки, де відвідувачі можуть ввести свою електронну адресу, щоб отримувати нові випуски розсилки прямо на свою електронну пошту. Це надає зручність та простоту доступу до оновлень для користувачів.

Попри важливість та внесок RubyWeekly в спільноту Ruby, існують деякі області, де цей веб-сайт може бути вдосконалений.

Один з очевидних недоліків – це дизайн сайту. Візуальний аспект сайту виглядає морально застарілим, що може не відповідати сучасним

стандартам та очікуванням користувачів. Оновлення дизайну та інтерфейсу користувача може значно поліпшити досвід користувачів і зробити сайт більш привабливим та легким у використанні.

Ще одним недоліком є відносно невелика кількість новин у кожній розсилці. Враховуючи швидкість змін та розвитку у сфері програмування, спільнота Ruby on Rails та Ruby заслуговує на більше актуального та цікавого контенту. Інтенсивніше оновлення новин може покращити цінність ресурсу для його читачів.

Відсутність інформації про розсилку, її авторів та процес відбору новин є одним з важливих недоліків RubyWeekly. Прозорість і ясність цих аспектів допомагають користувачам краще розуміти джерела та надійність новин, а також створюють більш прозорий та відкритий діалог між авторами розсилки та її читачами.

Процес відбору новин також є туманним. Для користувачів, що є авторами контенту, може бути незрозуміло, які критерії використовуються для вибору матеріалів для розсилки. Це може ускладнити їм підготовку та подання своїх матеріалів для розгляду. Прикладом може бути ситуація, коли автор новини або блогу хоче, щоб його контент потрапив до розсилки RubyWeekly, але не знає, які вимоги або процеси потрібно виконати для цього.

Отже, забезпечення більшої прозорості щодо авторства, процесу відбору новин та деталей розсилки може значно покращити досвід користувачів з RubyWeekly, а також допомогти авторам контенту краще адаптуватися до вимог розсилки.

LibHunt Ruby (https://ruby.libhunt.com/) – цей веб-сайт зосереджений на відстеженні та представленні нових бібліотек Ruby. Він забезпечує корисну інформацію про різні бібліотеки, включаючи їх рейтинги, оновлення та відгуки (рис. 1.2).

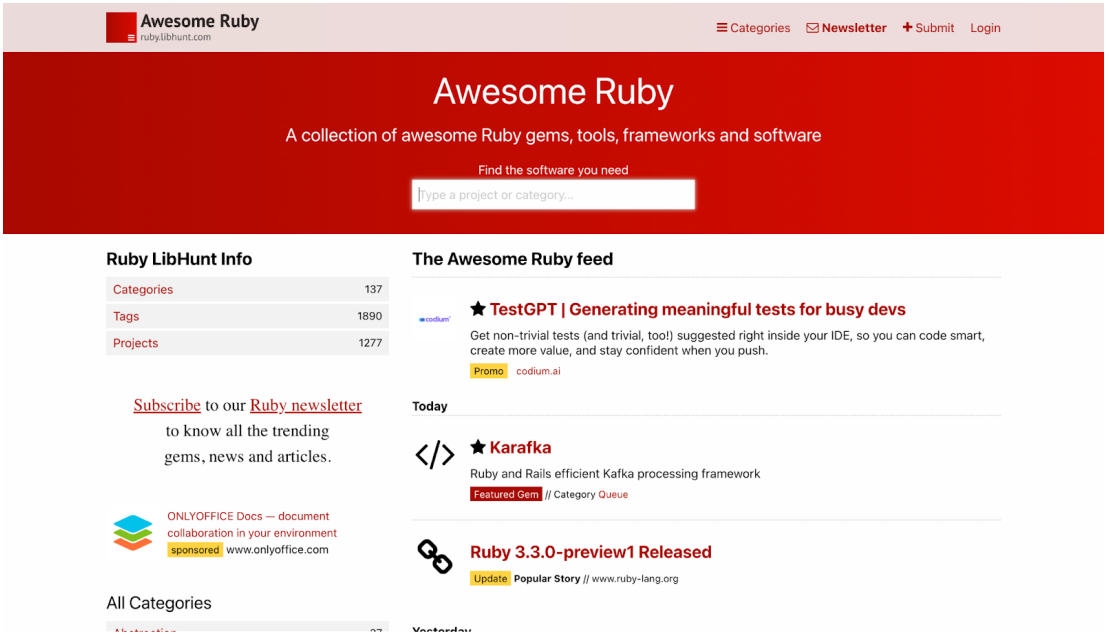

Рисунок 1.2 – Головна сторінка LibHunt Ruby

LibHunt Ruby, попри свою корисність як ресурс для спільноти Ruby, також має декілька областей, які можуть бути вдосконалені.

По-перше, дизайн сайту LibHunt Ruby є морально застарілим, подібно до RubyWeekly. Сучасні користувачі очікують від сайтів чіткості, сучасного дизайну та інтуїтивно зрозумілого інтерфейсу. Оновлення дизайну та інтерфейсу користувача може значно поліпшити загальний користувацький досвід на сайті.

Щодо частоти оновлень, LibHunt Ruby переважає RubyWeekly, оскільки новини з'являються практично кожен день, а не тільки раз на тиждень. Однак, незважаючи на більшу частоту оновлень, кількість новин все ще залишається обмеженою.

Крім того, на LibHunt Ruby можна переглянути новини тільки за останні 30 днів. Це обмеження може бути незручним для користувачів, які хочуть відстежувати або повернутися до старіших новин. Розширення архіву новин або надання можливості перегляду старіших новин може значно збільшити корисність сайту для його користувачів.

1.4 Постановка задачі

Основною задачею даної дипломної роботи є розробка веб-сайту агрегатору новин засобами Ruby on Rails, який включатиме в себе елементи веб-скрапінгу для збору новин з різних джерел [2]. Сайт має надати користувачам актуальні новини, інформацію про нові версії бібліотек та фреймворків, а також інформацію про майбутні події та конференції в сфері Ruby і Ruby on Rails.

Специфічні задачі, які потрібно вирішити в процесі виконання цієї роботи, включають:

- аналіз вимог до функціональності веб-сайту, включаючи огляд існуючих аналогів і визначення їхніх сильних і слабких сторін;
- проектування архітектури веб-сайту, включаючи структуру бази даних та систему навігації по сайту;
- реалізація веб-скрапінгу для автоматичного збору новин з відповідних джерел;
- розробка фронтенду і бекенду веб-сайту з використанням технології Ruby on Rails;
- тестування і налагодження веб-сайту для забезпечення його надійності та ефективності;
- розгортання веб-сайту та його підготовка до роботи в реальних умовах.

Результатом виконання цієї роботи має стати повноцінний веб-сайт, який забезпечує ефективний збір, організацію та представлення новин для спільноти Ruby і Ruby on Rails.

Висновки до розділу 1

У даному розділі було досліджено предметну область проекту, яка об'єднує аналіз новин та інформації зі світу Ruby і Ruby on Rails. Було

проведено детальний огляд двох аналогів - RubyWeekly та LibHunt Ruby, з оглядом їхніх сильних та слабких сторін. Було з'ясовано, що обидва ресурси мають певні недоліки, що можуть вплинути на загальний досвід користувача. Основними проблемами є відсутність прозорості в процесі відбору та публікації новин, обмеженість архіву новин, а також морально застарілий дизайн веб-сайтів.

На основі проведеного аналізу було сформульовано основну задачу дипломної роботи – розробку веб-сайту агрегатора новин за допомогою Ruby on Rails, який буде включати в себе елементи веб-скрапінгу для збору новин з різних джерел. Специфічні задачі, що випливають з основної, були детально описані. Результатом виконання цієї роботи має стати повноцінний веб-сайт, який забезпечує ефективний збір, організацію та представлення новин для спільноти Ruby і Ruby on Rails.

Цей розділ лягає в основу подальшого дослідження та розробки, надаючи чітке розуміння потреб користувачів та вимог до функціональності майбутнього веб-сайту.

РОЗДІЛ 2. ПРОЕКТУВАННЯ АРХІТЕКТУРИ ТА ПЛАН РОЗРОБКИ ДОДАТКУ

2.1 Особливості сучасних інструментів веб-скрапінгу у Ruby

У сфері Ruby для веб-скрапінгу найчастіше використовується бібліотека Nokogiri. Вона надає зручні інструменти для аналізу HTML та XML документів і дозволяє легко витягувати необхідну інформацію [3]. Однак, попри переваги, використання Nokogiri може бути не достатньо гнучким у деяких сценаріях.

Основною проблемою є те, що для кожного нового джерела даних, з якого потрібно здійснювати скрапінг, доведеться писати окремий Ruby код. Це може бути трудомістким і призводити до значного росту обсягу коду, особливо якщо джерела даних змінюються чи оновлюються.

Враховуючи ці обмеження, більш ефективним рішенням може бути використання окремого сервісу для веб-скрапінгу. Такий сервіс може приймати скрипт, що визначає, як здійснювати скрапінг конкретного джерела даних, і виконувати його автоматично.

Один з таких сервісів – Puppeteer, потужний інструмент для контролювання веб-браузера Google Chrome або Chromium за допомогою API [4]. Puppeteer може використовуватися для широкого спектра завдань, включаючи генерацію знімків екрану веб-сторінок, автоматизацію форм та, звичайно, веб-скрапінг.

Проте, в контексті даної роботи, буде використовуватися самостійно розміщений на сервері сервіс Browserless.io. Він пропонує всі переваги Puppeteer, та надає велику кількість додаткового функціоналу. Використання Browserless дозволить ефективно і гнучко виконувати веб-скрапінг з різних джерел, зменшуючи обсяг коду, необхідного для написання та підтримки [5]. Сервіс Browserless.io має вбудовані засоби для

управління браузерами, що спрощує задачу збору даних із веб-сторінок. Це також дозволяє легко адаптуватися до змін в структурі джерела даних, оскільки зміни можна легко внести в скрипт для скрапінгу, без необхідності внесення змін в основний код програми.

Окрім того, Browserless.io надає можливість паралельного запуску скрапінгу, що дозволяє швидко обробляти великі обсяги даних. Це особливо корисно для реалізації веб-сайту агрегатора новин, де потрібно збирати дані з великої кількості джерел у реальному часі.

Отже, узагальнюючи, враховуючи вищезазначені обмеження та потреби, найбільш оптимальним рішенням для даної задачі є використання сервісу Browserless.io для веб-скрапінгу.

2.2 Розробка архітектури веб-сайту

Архітектура веб-сайту – це структура його компонентів, їх взаємозв'язок, принципи взаємодії між ними та з користувачами. Під час розробки архітектури веб-сайту необхідно враховувати багато факторів: цілі та задачі проекту, технічні можливості, зручність користування, а також прогнозування майбутнього розвитку та масштабування проекту.

У випадку з проектом "rails.news", основна мета – це створення високопродуктивного, надійного та легко масштабованого веб-сайту агрегатора новин, який буде включати в себе інтеграцію з веб-скрапінгом. Для досягнення цієї мети було вирішено використовувати наступні компоненти:

- веб-сайт (моноліт Ruby on Rails): основний інтерфейс для користувачів. Ruby on Rails – це надійний та гнучкий фреймворк, який ідеально підходить для розробки веб-додатків. Він має потужну систему маршрутизації, вбудовані засоби для тестування та безпеки, а також велику спільноту розробників, яка забезпечує підтримку та постійне оновлення фреймворку [6];

- БД PostgreSQL: використовується для зберігання даних користувачів та новин. PostgreSQL – це потужна, високопродуктивна та гнучка БД, яка відмінно підходить для роботи з великими об'ємами даних [7];
- БД Redis: використовується для зберігання інформації, необхідної для роботи Sidekiq, а також для кешування даних. Redis - це високопродуктивна в пам'яті база даних, яка часто використовується для реалізації різноманітних систем кешування, черг задач, публікації/підписки на повідомлення і т.д [8];
- сервіс Browserless: використовується для реалізації скрапінгу веб-сторінок. Browserless це self-hosted сервіс, який базується на Puppeteer - бібліотеці для керування браузером Chrome або Chromium в хедлес режимі [9]. Він дозволяє виконувати скрапінг веб-сторінок, надсилаючи скрипт безпосередньо на сервер Browserless, та отримувати вже готовий JSON зі скрапленими даними;
- воркер Sidekiq: використовується для розпаралелювання та виконання важких задач у фоновому режимі [10]. Він відділений від основного процесу веб-додатку, що дозволяє покращити загальну продуктивність та відгук системи. Він здатний виконувати численні задачі паралельно, включаючи, але не обмежуючись, запитами до Browserless для скрапінгу веб-сторінок. Ця асинхронна модель роботи дозволяє нам забезпечити високу пропускну здатність та надійність системи, уникнути блокування основного потоку виконання програми, та забезпечити швидке відновлення в разі будь-яких збоїв;
- сховище S3: використовується для зберігання та постачання статичного контенту, такого як зображення, відео, PDF-файли та інше. Amazon S3 (Simple Storage Service) - це об'єктне сховище, що надає масштабовану, високопродуктивну, надійну та безпечну платформу для зберігання даних в інтернеті [11]. Він забезпечує гнучкість управління

даними, дозволяючи зберігати дані будь-якого типу і в будь-якому масштабі;

- reverse-proxy Traefik: високопродуктивний та гнучкий reverse-proxy сервер, що дозволяє маршрутизувати трафік до різних сервісів в межах даної інфраструктури. Він автоматично розподіляє навантаження, забезпечує підтримку HTTPS, дозволяє виконувати автоматичне оновлення сертифікатів SSL та інше [12].

Ця архітектура була обрана, оскільки вона дозволяє гнучко та ефективно розподіляти навантаження між різними компонентами системи, що дозволяє досягти високої продуктивності та масштабованості (рис. 2.1). Крім того, використання спеціалізованих сервісів для виконання конкретних задач (наприклад, скрапінг веб-сторінок) дозволяє зосередитися на основних задачах розробки та підтримки веб-сайту, замість витрачання ресурсів на розробку та підтримку додаткового функціоналу.

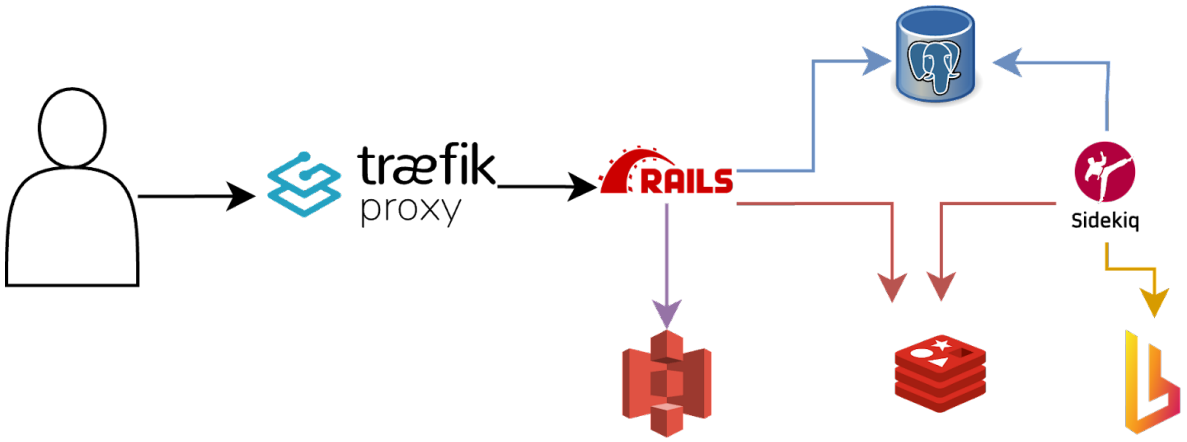

Рисунок 2.1 – Архітектура rails.news

Використання Ruby on Rails як основного фреймворку для розробки веб-сайту дозволяє використовувати багато вбудованих засобів для реалізації різноманітного функціоналу, а також отримати підтримку великої спільноти розробників.

PostgreSQL вибрана для зберігання основних даних проекту через свою надійність, високу продуктивність та гнучкість

Redis використовується для зберігання даних, що часто змінюються або потребують швидкого доступу (наприклад, дані для роботи Sidekiq або кеш).

Було вирішено використовувати самостійно розгорнутий на сервері сервіс Browserless для скрапінгу веб-сторінок, оскільки він надає гнучкі можливості для керування процесом скрапінгу, а також дозволяє отримувати вже готові дані в форматі JSON.

Sidekiq використовується для розпаралелювання та виконання важких задач у фоновому режимі, що дозволяє покращити загальну продуктивність та відгук системи.

Сховище S3 дозволяє просто та надійно зберігати статичні файли.

Traefik використовується як reverse-proxy для забезпечення ефективного розподілу навантаження та забезпечення безпеки.

Всі ці компоненти разом створюють архітектуру, яка спроможна забезпечити стабільну, надійну та високопродуктивну роботу веб-сайту "rails.news".

2.3 Створення мокапу для веб-сайту

Процес створення мокапу для веб-сайту агрегатора новин був виконаний за допомогою інструменту для дизайну та прототипування - Figma. Figma надає широкий набір інструментів для створення та редагування дизайну, включаючи компоненти, стилі, шаблони та інше [13].

Основним етапом у створенні мокапу було створення індивідуальних компонентів. Компоненти в Figma – це елементи дизайну, які можна повторно використовувати в різних частинах проекту. Це можуть бути кнопки, поля вводу, заголовки, карточки (рис. 2.2), списки та інші елементи

інтерфейсу. Всі ці компоненти були розроблені в Figma для майбутнього використання.

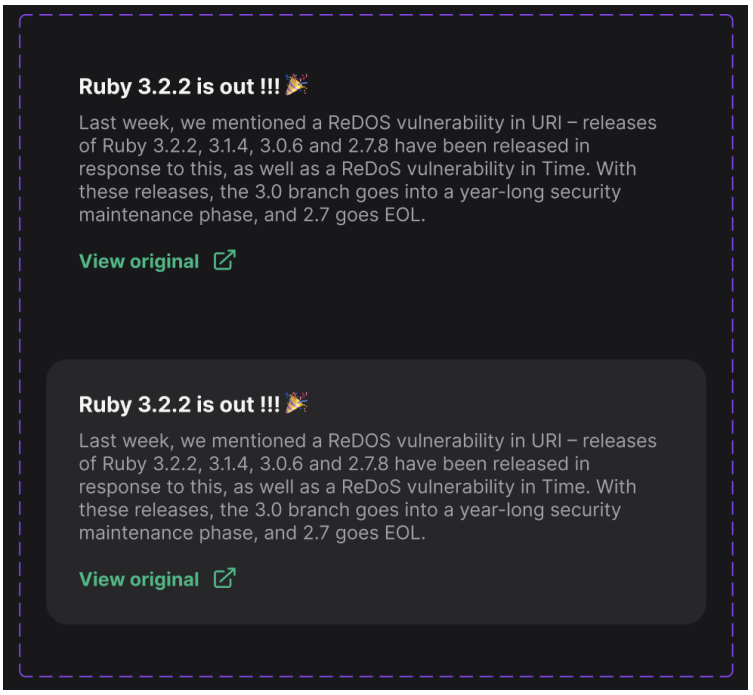

Рисунок 2.2 – Компонент Figma “карточка з новиною”

Окрім компонентів, було створено логотип для веб-сайту. Логотип є важливим елементом бренду, який допомагає користувачам легко впізнати та запам'ятати сайт. Після декількох ітерацій та обговорень було вирішено використати конкретний дизайн логотипу (рис. 2.3).

Рисунок 2.3 – Логотип веб-сайту на темному фоні

Наступним кроком було створення дизайну основних сторінок сайту: головної сторінки, сторінки архіву (рис. 2.4) та сторінки "About". Дизайн цих сторінок був розроблений з урахуванням потреб користувачів та

основних принципів UX і UI дизайну. Головна сторінка була розроблена таким чином, щоб надати користувачам швидкий доступ до найновіших новин, тоді як сторінка архіву надає можливість перегляду старіших публікацій. Сторінка "About" містить інформацію про проект, його місію та команду.

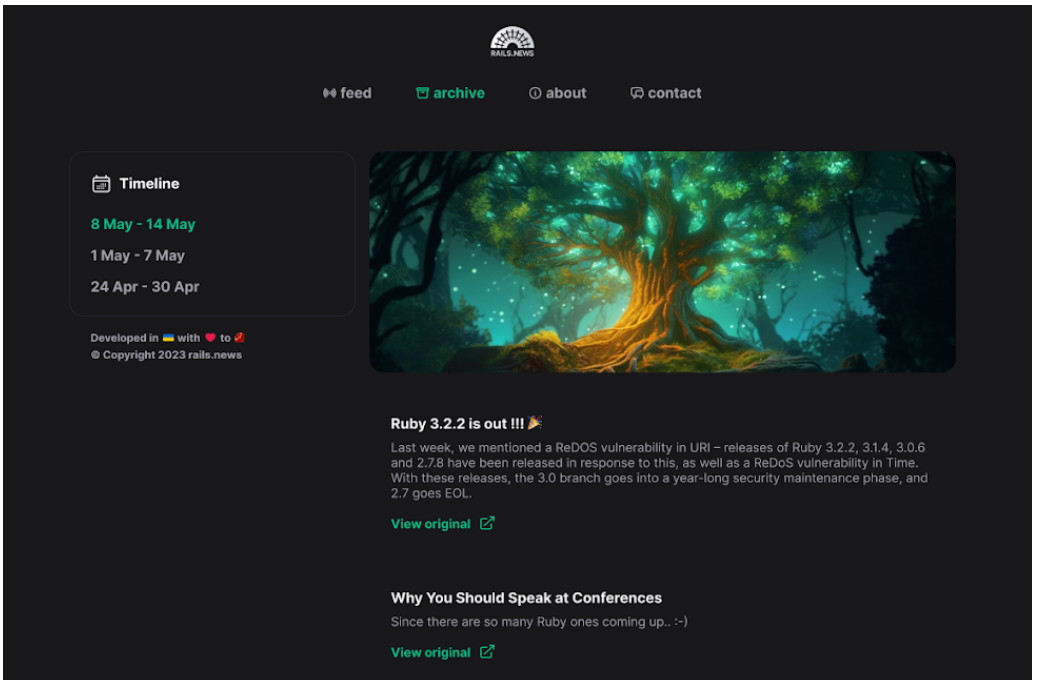

Рисунок 2.4 – Дизайн сторінки “Архів”

Також було створено дизайн для сторінок помилок 404, 422 та 500. Ці сторінки відображаються, коли виникає помилка при спробі навігації по сайту, помилці в даних під час надсилання форми та при помилці на сервері відповідно. Дизайн цих сторінок було виконано з урахуванням користувацького досвіду – вони не лише повідомляють користувачів про проблему, але також надають корисні рекомендації та посилання для дальшої навігації по сайту.

Загалом, створення мокапу веб-сайту було важливим кроком в процесі розробки. Він дозволив визначити загальний вигляд та структуру сайту, а також протестувати різні ідеї та концепції дизайну перед тим, як почати програмування.

Додатково, при створенні дизайну веб-сайту, було вирішено використовувати сервіс генерації ілюстрацій - midjourney (рис. 2.5). Цей інструмент дозволяє генерувати унікальні ілюстрації з текстового опису, які можуть бути використані для прикрашання веб-сайту та зробити його використання більш приємним для користувачів [14].

Використання Midjourney дозволило внести в дизайн веб-сайту більше кольору та візуального різноманіття, що сприяло створенню більш привабливого та захоплюючого користувацького досвіду. Особливо це стало актуальним для головної сторінки та сторінки архіву, де основний контент - це текстові новини. Ілюстрації Midjourney додали візуального контрасту та зробили перегляд контенту більш приємним і цікавим.

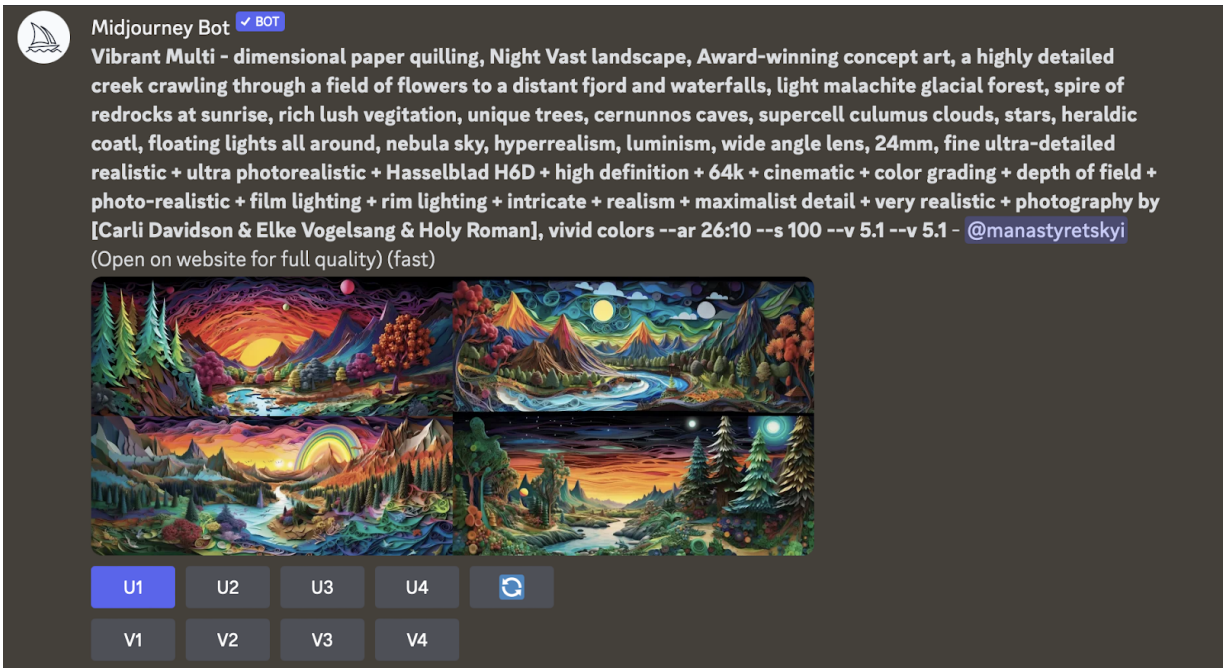

Рисунок 2.5 – Приклад генерації зображення з Midjourney

2.4 Вибір інструменту для розгортання веб-сайту в продакшені

Робота веб-сайту в продакшені включає в себе ряд кроків, які забезпечують стабільність, швидкість та безпеку веб-додатку. Одним із ключових етапів цього процесу є розгортання серверу, що вимагає

детального планування та глибоких знань про серверну інфраструктуру. Саме тут MRSK виявляється незамінним інструментом.

MRSK – це інструмент, що призначений для спрощення розгортання веб-додатків. Він дозволяє автоматизувати багато процесів, пов'язаних з розгортанням та управлінням серверами, що значно спрощує життя розробників та системних адміністраторів [15].

За допомогою MRSK, ми можемо легко розгортати наш веб-сайт на будь-яких серверах з операційною системою Linux, без будь-якої додаткової підготовки, окрім додавання SSH ключа. Все, що нам потрібно зробити - це надати MRSK файл конфігурації з переліком IP-адрес серверів, і він самостійно встановить необхідне середовище для роботи нашого додатку.

Цей підхід має ряд переваг порівняно з традиційними способами розгортання серверу:

- портабельність: MRSK дозволяє розгортати веб-додатки на будь-яких серверах, незалежно від того, де вони розташовані - в хмарі чи на власному обладнанні. Це дозволяє нам легко масштабувати інфраструктуру та переносити її між різними хмарними провайдерами;
- автоматизація: MRSK автоматично встановлює всі необхідні залежності, такі як Docker, на серверах, що вказані в файлі конфігурації. Це виключає необхідність ручного встановлення і налаштування середовища на кожному сервері, що значно прискорює процес розгортання і зменшує ймовірність помилок;
- незалежність від комерційних платформ: MRSK не прив'язаний до жодної комерційної платформи, що дозволяє нам використовувати будь-який хмарний провайдер або власне обладнання без будь-яких обмежень. Це надає нам більше контролю над нашою інфраструктурою та зменшує ризик залежності від одного провайдера.

Висновки до розділу 2

На основі проведеного аналізу, розробки архітектури та вибору інструментів, було визначено чіткий план розробки веб-сайту. Зокрема, процес створення мокапу допоміг візуалізувати кінцевий продукт і розробити зрозумілий і інтуїтивний дизайн користувальницького інтерфейсу, а вибір MRSK для розгортання додатку в продакшені дозволяє значно скоротити час необхідний для того щоб ввести програму в експлуатацію та забезпечує її гнучкість, автономність та ефективність.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-САЙТУ

3.1 Створення базової аплікації

Початкова ініціалізація проекту була здійснена за допомогою команди:

```
rails new rails.news --database=postgresql -a propshaft --main -T
```

Дані параметри виконують такі функції:

- rails new - ця команда створює нову базову структуру додатка Ruby on Rails;
- rails.news - це назва нового проекту;
- --database=postgresql - цей параметр вказує на те, що основною базою даних для проекту буде PostgreSQL. Rails автоматично включає необхідні геми для роботи з цією базою даних і генерує конфігураційний файл з налаштуваннями підключення;
- -a propshaft - цей параметр вказує на використання Propshaft замість Sprockets для компіляції статичних активів. Propshaft - це новіший інструмент, який пропонує більшу продуктивність та простоту використання;
- --main - цей параметр вказує, що цей додаток є головним в проекті. В останній версії Rails, яка зараз знаходиться в розробці;
- -T - цей параметр вказує на відмову від стандартного тестового фреймворку Minitest, який включається за замовчуванням. Замість цього для написання тестів буде використовуватися RSpec, що є одним з найпопулярніших та найпотужніших інструментів для тестування в Ruby.

Завдяки параметру --main, в процесі ініціалізації проекту було автоматично створено Dockerfile. Dockerfile - це текстовий документ, що містить всі команди, які користувач може викликати в командному рядку

для створення образу Docker [16]. У даному випадку, Dockerfile буде використовуватися для розгортання нашого веб-сервера.

Було використано версію Ruby on Rails, яка ще знаходиться в процесі розробки. Таке рішення може здатися дещо неочікуваним, але варто зазначити, що Ruby on Rails відомий своїми високими стандартами якості. Кожна нова версія проходить ретельну перевірку і тестування перед її випуском. Така стабільність і надійність роблять використання найновіших версій цього фреймворку цілком безпечним.

Важливо відмітити, що такі великі сайти, як GitHub, Hey, Basecamp, активно використовують останні версії Ruby on Rails в своїх додатках. Це свідчить про високу надійність і продуктивність фреймворку, а також дає мені впевненість у можливості отримати необхідну підтримку та документацію для роботи з найновішими технологіями.

3.2 Налаштування тестового середовища

Тестування коду є важливим аспектом розробки програмного забезпечення, який не можна ігнорувати. При розробці веб-додатку, особливо коли в команді відсутні спеціалізовані тестувальники, написання автоматизованих тестів стає критично важливим.

Автоматизовані тести дають впевненість в тому, що код працює так, як очікується. Вони допомагають виявити помилки на ранніх стадіях розробки, що в свою чергу зменшує час витрачений на їх виправлення. Більше того, тести є важливим інструментом рефакторингу, адже вони дозволяють нам без страху змінювати код, знаючи, що якщо щось піде не так, це виявиться на етапі тестування.

RSpec – це потужний фреймворк для тестування, який використовується у багатьох проектах Ruby. Його відмінність полягає в читабельності та виразності, а також в тому, що він дозволяє легко писати тести, які описують поведінку програми [17].

Для встановлення RSpec, було додано наступний гем в Gemfile: gem 'rspec-rails', '~> 5.0'. Після додавання гему в Gemfile, було виконано команду bundle install для встановлення гема, а потім rails generate rspec:install для генерації необхідних файлів та директорій RSpec.

Для спрощення написання тестів було вирішено використовувати такі геми як simplecov, factory_bot та faker.

- simplecov - це інструмент для вимірювання покриття коду тестами. Він дозволяє відстежувати, які частини коду виконуються під час тестування, і які не виконуються, що дозволяє визначити, які частини коду потребують додаткового тестування [18];
- factory_bot - це бібліотека, яка дозволяє створювати об'єкти для тестування. Замість того, щоб кожен раз ручним способом створювати об'єкти, можна використовувати "фабрики", які автоматично створюють об'єкти з заданими атрибутами [19];
- faker - це гем, який дозволяє генерувати випадкові дані. Це може бути корисним, наприклад, при написанні тестів, де потрібно створювати об'єкти з випадковими значеннями [20].

Ці три інструменти разом з RSpec дозволяють створювати чіткі, гнучкі та легко розширювальні тести. Вони полегшують процес написання тестів і допомагають забезпечувати високу якість коду.

simplecov встановлюється шляхом додавання gem 'simplecov', require: false, group: :test в Gemfile і виконання команди bundle install. Його використання розпочинається з додавання require 'simplecov'; SimpleCov.start на початку spec_helper.rb або rails_helper.rb. Це автоматично збирає інформацію про покриття коду тестами під час їх виконання, і генерує звіт, який можна переглянути в браузері.

factory_bot додається в Gemfile таким чином: gem 'factory_bot_rails'. Після встановлення за допомогою bundle install, в rails_helper.rb додається config.include FactoryBot::Syntax::Methods, що дозволяє використовувати методи фабрики безпосередньо в тестах.

faker додається в Gemfile за допомогою gem 'faker'. Цей гем не потребує додаткової конфігурації, і його можна використовувати в тестах або фабриках для генерації тестових даних.

Крім вищезгаданих інструментів, також було використано database_cleaner-active_record для забезпечення чистоти бази даних під час тестування. Це особливо важливо, коли тести вимагають певного стану бази даних або коли вони можуть вплинути на стан бази даних і спричинити побічні ефекти, які можуть вплинути на результати інших тестів.

Встановлення database_cleaner-active_record відбувається шляхом додавання gem 'database_cleaner-active_record' до Gemfile і виконання команди bundle install.

Потім, в rails_helper.rb налаштовується database_cleaner:

```
require "database_cleaner/active_record"

RSpec.configure do |config|
  config.before(:suite) do
    DatabaseCleaner.strategy = :transaction
    DatabaseCleaner.clean_with(:truncation)
  end

  config.around(:each) do |example|
    DatabaseCleaner.cleaning do
      example.run
    end
  end
end
```

Ця конфігурація гарантує, що перед кожним тестом база даних очищується, що забезпечує унікальність умов для кожного тесту.

3.3 Написання сервісу для веб-скрапінгу

У рамках роботи було створено сервісний об'єкт під назвою BrowserlessScraper, що використовує browserless.io - сервіс, що надає API для безголових браузерів. Це дозволяє виконувати скрапінг з використанням JavaScript коду безпосередньо на стороні сервера, що значно підвищує швидкість та ефективність цього процесу.

Клас BrowserlessScraper ініціалізується з джерелом, яке буде використано для скрапінгу. Для визначення URL-адреси сервера browserless, використовується змінна середовища BROWSERLESS_URL. У випадку, коли змінна не встановлена, застосовується значення за замовчуванням "http://localhost:5001".

Код сервісу BrowserlessScraper:

```
require "net/http"

class BrowserlessScraper
  attr_reader :browserless_url, :config, :source

  def initialize(source)
    @browserless_url = ENV.fetch("BROWSERLESS_URL", "http://localhost:5001")
    @source = source
    @config = ScrapperConfig.new(source)
  end

  def scrape_website
    url = URI(browserless_url + "/function?stealth=true&headless=true")
    http = Net::HTTP.new(url.host, url.port)
    http.read_timeout = 180

    request = Net::HTTP::Post.new(url)
    request["Content-Type"] = "application/json"

    payload = {
      "code" => config.js_code,
      "context" => {
        "url" => config.base_url
```

```
    }
   }

   request.body = payload.to_json

   response = http.request(request)

   JSON.parse(response.read_body).map(&:with_indifferent_access)
  end
end
```

Конструктор цього сервісу приймає параметр source – строка використовуючи яку в подальшому читаються налаштування для певного сайту.

Метод scrape_website створює POST-запит до сервера browserless для виклику функції "/function", яка передає JavaScript код, який здійснює витягування даних з веб-сайту, та контекст, що містить URL-адресу веб-сайту, який підлягає скрапінгу. Запит має таймаут на читання 180 секунд, що надає достатній час для завантаження сторінки та виконання коду JavaScript.

Отриманий відповіддю на запит з сервера browserless JSON обробляється та повертається у вигляді масиву геш-таблиць Ruby.

Було створено клас ScrapperConfig призначений для управління конфігурацією скрапера. Він включає методи для завантаження та аналізу YAML-файлу конфігурації та відповідного JavaScript-коду. Крім того, цей клас дозволяє динамічно отримувати значення налаштувань, що дозволяє гнучко працювати з різними джерелами даних без зміни коду класу.

Спеціально перевизначені методи method_missing та respond_to_missing? дозволяють викликати методи з іменами налаштувань безпосередньо на об'єкті ScrapperConfig. Якщо такий метод не визначений в класі, він звертається до значень налаштувань, що забезпечує гнучкий і інтуїтивно зрозумілий інтерфейс для роботи з налаштуваннями скрапера.

Код класу ScrapperConfig:

```
class ScrapperConfig
  attr_reader :source

  def initialize(source)
    @source = source
  end

  def config
    @config ||=
YAML.load_file(Rails.root.join("scraper_config/#{source}/config.yml")).with_indifferent_a
ccess
  end

  def js_code
    @js_code ||=
File.read(Rails.root.join("scraper_config/#{source}/#{config["script_file"]}"))
  end

  def method_missing(method_name, *args, &block)
    config[method_name.to_s]
  end

  def respond_to_missing?(method_name, include_private = false)
    config.key?(method_name.to_s) || super
  end
end
```

На прикладі на RubyFlow можна розглянути як виглядатимуть налаштування та скрипт для скрапінгу. RubyFlow є веб-сайтом, де розробники можуть поділитися посиланнями на корисні ресурси, статті, новини та проекти, пов'язані з мовою програмування Ruby. Він включає різноманітні категорії, включаючи нові релізи бібліотек, важливі оновлення, навчальні матеріали, а також дискусії та запитання від спільноти.

Для скрапінгу RubyFlow було прописано такі параметри як URL-адресу сайту, ім'я файлу з JavaScript кодом для скрапінгу, а також стилі для відображення даних:

```
name: RubyFlow
base_url: https://rubyflow.com
script_file: index.js
styles:
  text_color: "#fff"
  background_color: "#c00"
```

Для роботи Browserless в контексті хедлес браузера, в JavaScript коді який йому необхідно експортувати функцію, вона може використовувати всі API браузера для навігації по веб-сайту, витягування даних та інших дій, які зазвичай виконуються на стороні клієнта.

JavaScript код який виконується для скрапінгу новин з RubyFlow:

```
const scrapePostContent = async (browser, url) => {
  const newPage = await browser.newPage();
  await newPage.goto(url);
  const content = await newPage.$eval('.inner.post .content', content =>
content.innerHTML.trim());
  await newPage.close();
  return content;
}

module.exports = async ({ page, context }) => {
  const { url } = context;
  const browser = page.browser();
  const results = [];

  await page.goto(url);
  const posts = await page.$$('.inner.posts article.post');

  for (const post of posts) {
    const postUrl = url + await post.$eval('h1 a', a => a.getAttribute('href'));
    const hasMoreText = await post.evaluate(post => post.querySelector('.more') !==
null);
    const content = hasMoreText
```

```
      ? await scrapePostContent(browser, postUrl)
      : await post.$eval('.body p', p => p.innerHTML.trim());

    results.push({
      title: await post.$eval('h1', h1 => h1.textContent.trim()),
      uid: await post.evaluate(post => post.getAttribute('data-uid')),
      id: parseInt(await post.evaluate(post => post.getAttribute('data-id')), 10),
      url: postUrl,
      timestamp: new Date(parseInt(await post.evaluate(post =>
post.getAttribute('data-timestamp')), 10) * 1000),
      content: content,
      user: {
        name: await post.$eval('.metadata cite', cite => cite.textContent.trim()),
        url: await post.$eval('.metadata cite a', a => a.getAttribute('href'))
      }
    });
  }

  return {
    data: results,
    type: 'application/json',
  };
};
```

В даному випадку, функція переходить на URL, що визначено в контексті (в даному випадку це головна сторінка сайту RubyFlow), використовуючи об'єкт page браузера. Потім вона знаходить всі пости на сторінці, використовуючи CSS селектори для визначення місця посту в структурі HTML. Далі виконується ітерація по кожному посту для витягування даних. В результаті повертається об'єкт з масивом результатів і типом даних 'application/json'.

В циклі проводиться аналіз окремого посту, та витягується потрібна інформація яка потім зберігається в масив з результатами.

Для кожного поста витягується URL, перевіряється наявність додаткового тексту за посиланням "Read more", а також витягується основний вміст поста. Потім інформація про кожний пост, включаючи

заголовок, унікальний ідентифікатор (UID), ідентифікатор поста, URL, часову мітку, вміст та інформацію про користувача, зберігається у масиві results. Таким чином, масив results в кінці містить детальну інформацію про всі пости на поточній сторінці.

Так як контент деяких постів є надто великим і інформація може обрізатися на головній сторінці, для таких постів викликається функція scrapePostContent. Ця функція створює нову сторінку в браузері, переходить на заданий URL і виконує витягування повного вмісту поста. Після цього вона закриває сторінку і повертає отриманий вміст. Цей підхід гарантує, що ми збираємо максимально можливу кількість інформації для кожного поста, незалежно від того, як він відображається на головній сторінці сайту.

Додатково до сервісу BrowserlessScraper, було створено ще один сервіс під назвою MetadataScraper. Цей сервіс розроблено для витягування метаданих з веб-сторінок, до яких ведуть посилання зібрані з допомогою BrowserlessScraper.

Метадані – це набір даних, які описують та дають інформацію про інші дані. У контексті веб-сторінок, метадані можуть включати такі елементи, як заголовок сторінки, опис, автор, дата публікації, зображення та інші. Метадані важливі для пошукових систем, таких як Google, оскільки вони використовують цю інформацію для індексації та ранжування веб-сторінок. Крім того, метадані використовуються соціальними мережами для відображення змісту, коли посилання діляться в цих мережах.

Для витягування метаданих використовується бібліотека metascraper. Ця бібліотека дозволяє витягувати різні типи метаданих з HTML-вмісту сторінки [21]. У цьому випадку використовуються наступні модулі metascraper: author, date, description, image, logo, publisher, title, url.

Клас MetadataScraper ініціалізується з адресою сервера browserless, яку визначає змінна середовища BROWSERLESS_URL. Якщо змінна не встановлена, за замовчуванням використовується "http://localhost:5001".

Метод scrape_website класу MetadataScraper створює POST-запит до сервера browserless, що викликає функцію JavaScript, що експортується через module.exports. Ця функція використовує браузерну сторінку для переходу на URL, визначений у контексті, та отримує HTML-вміст сторінки. Потім вона використовує metascraper для витягування метаданих з цього HTML-вмісту. Результати, разом з типом даних 'application/json', повертаються як відповідь функції.

У самому JavaScript коді використовуються різні модулі metascraper, включаючи 'metascraper-author', 'metascraper-date', 'metascraper-description', 'metascraper-image', 'metascraper-logo', 'metascraper-publisher', 'metascraper-title', 'metascraper-url'. Кожен з цих модулів відповідає за витягування відповідного типу метаданих.

Після того, як код JavaScript було виконано, сервіс MetadataScraper отримує відповідь від сервера browserless і обробляє цю відповідь, перетворюючи її з JSON формату в Ruby Hash за допомогою методу JSON.parse(response.read_body).with_indifferent_access. Результатом є хеш, який містить метадані веб-сторінки, включаючи таку інформацію, як автор, дата, опис, зображення, логотип, видавець, заголовок і URL.

Код сервісу MetadataScraper:

```
require "net/http"

class MetadataScraper
  attr_reader :browserless_url

  def initialize
    @browserless_url = ENV.fetch("BROWSERLESS_URL", "http://localhost:5001")
  end

  def scrape_website(website_url)
    url = URI(browserless_url + "/function?stealth=true&headless=true")
    http = Net::HTTP.new(url.host, url.port)
    http.read_timeout = 180

    request = Net::HTTP::Post.new(url)
```

```
  request["Content-Type"] = "application/json"
  js_code = <<~JS
    module.exports = async ({ page, context }) => {
      const { url } = context;
      const browser = page.browser();

      await page.goto(url);

      const metascraper = require('metascraper')([
        require('metascraper-author')(),
        require('metascraper-date')(),
        require('metascraper-description')(),
        require('metascraper-image')(),
        require('metascraper-logo')(),
        require('metascraper-publisher')(),
        require('metascraper-title')(),
        require('metascraper-url')()
      ])

      results = await metascraper({url, html: await page.content()});

      return {
        data: results,
        type: 'application/json',
      };
    };
  JS
  payload = {
    "code" => js_code,
    "context" => {
      "url" => website_url
    }
  }

  request.body = payload.to_json

  response = http.request(request)

  JSON.parse(response.read_body).with_indifferent_access
end
```

```
end
```

Для того, щоб використовувати модулі metascraper в контексті "headless" браузера, було створено власний Docker образ на основі browserless/chrome.

Вміст Dockerfile сервісу Browserless:

```
FROM browserless/chrome:1.58-chrome-stable

RUN npm install metascraper metascraper-audio metascraper-author metascraper-date
metascraper-description metascraper-feed metascraper-image metascraper-iframe
metascraper-lang metascraper-logo metascraper-publisher metascraper-readability
metascraper-title metascraper-url metascraper-video metascraper-logo-favicon
metascraper-media-provider
```

У Dockerfile було вказано, що базовим образом буде browserless/chrome:1.58-chrome-stable. Потім за допомогою команди RUN npm install було встановлено необхідні модулі metascraper.

Таким чином, було створено новий Docker образ, який містить все необхідне для роботи сервісу MetadataScraper.

Після створення Docker образу було необхідно загрузити його на Docker Hub. Docker Hub – це сервіс, що надає репозиторії для зберігання Docker образів. Завдяки Docker Hub, образ можна легко використовувати на будь-якому сервері, що підтримує Docker, без необхідності передавати великі файли образу між серверами. Також Docker Hub автоматично відслідковує зміни в образах і може автоматично оновлювати розгорнуті контейнери.

Завдяки використанню Docker, Docker Hub та MRSK, процес розгортання і обновлення сервісу MetadataScraper спрощується і автоматизується, що дозволяє зосередитися на розробці функціоналу, замість адміністрування серверів.

3.4 Створення моделі для збереження зібраних даних

Моделі в Ruby on Rails використовуються для взаємодії з базою даних і представляють собою абстрактне представлення даних в аплікації. Вони забезпечують засоби для зчитування, запису, пошуку та зміни даних. Крім того, моделі відповідають за валідацію даних перед їх збереженням, а також можуть містити бізнес-логіку, пов'язану з цими даними. Використання моделей в Ruby on Rails забезпечує багато переваг, зокрема зручність використання, ефективність, безпеку та масштабованість.

Створити модель та міграції можна за допомогою генератора Ruby on Rails такою командою:

```
rails generate model ScrapedPost.
```

Код міграції для створення таблиці scraped_posts:

```
class CreateScrapedPosts < ActiveRecord::Migration[7.1]
  def change
    enable_extension "pgcrypto" unless extension_enabled?("pgcrypto")

    create_table :scraped_posts do |t|
      t.uuid :uuid, null: false, default: -> { "gen_random_uuid()" }
      t.string :title, null: false
      t.string :source, null: false
      t.string :source_uid
      t.string :original_url, null: false
      t.datetime :original_timestamp
      t.string :original_content
      t.jsonb :user_info, null: false, default: {}
      t.jsonb :raw_data, null: false, default: {}

      t.timestamps
    end
  end
end
```

Код моделі ScrapedPost:

```
class ScrapedPost < ApplicationRecord
  enum status: {
    pending: "pending",
    published: "published",
    rejected: "rejected"
  }

  validates :source, :title, :original_url, presence: true
  validates :user_info, :raw_data, presence: true
  validates :source_uid, presence: true

  before_validation :set_uuid, on: :create

  private

  def set_uuid
    self.uuid ||= SecureRandom.uuid
  end
end
```

Клас ScrapedPost успадковує ApplicationRecord. Кожне поле в базі даних представлене як атрибут цього класу.

Використовуючи метод validates, ми можемо задати вимоги до значень цих атрибутів. Наприклад, вищенаведений код вимагає, щоб original_url, raw_data, source, source_uid, status, title, user_info та uuid були присутніми (presence: true).

Також було задано обмеження на status з допомогою enum, так що він може приймати лише значення 'pending', 'processed' або 'error'.

Метод before_validation визначає колбек, який виконується перед кожною валідацією. У цьому випадку, генерується uuid для нових записів за допомогою SecureRandom.uuid.

3.5 Написання сервісу фонової обробки з Sidekiq

Для розробки сервісу фонової обробки з Sidekiq, насамперед, було додано гем 'sidekiq' в Gemfile. Після цього, виконана команда bundle install для його встановлення.

Sidekiq – це потужний фреймворк для обробки фонових задач в Ruby. Він використовує потоки для обробки декількох задач паралельно, що робить його ефективним рішенням для виконання великої кількості задач.

Sidekiq використовує Redis як брокер повідомлень для зберігання інформації про фонові задачі. Тому, для його коректної роботи, було необхідно встановити Redis на сервері та налаштувати з'єднання з ним в конфігурації Sidekiq.

Для реалізації фонового виконання задач було використано інтерфейс ActiveJob, який також підтримується Sidekiq. ActiveJob – це фреймворк для опису фонових задач, інтерфейс якого підтримується переважною більшість бібліотек для виконання задач в фоновому режимі.

Було створено клас ScrapeNewsJob, який успадковує ApplicationJob і використовується для скрапінгу новин. Він виконується в черзі за замовчуванням.

В методі perform цього класу використовується клас BrowserlessScraper для скрапінгу веб-сайту. Після отримання даних, вони проходять через цикл, де для кожного посту створюється новий запис в базі даних за допомогою методу find_or_create_by!. Це гарантує, що буде створено новий запис, тільки якщо не існує запису з таким самим source та source_uid.

ActiveJob дозволяє створювати адаптивні фонові задачі, які можуть використовувати різні бекенди. Це робить код більш гнучким і масштабованим, оскільки дозволяє легко змінювати біблотеку яка буде виконувати ці задачі, якщо це потрібно.

Код класу ScrapeNewsJob:

```
class ScrapeNewsJob < ApplicationJob
  queue_as :default

  def perform(source)
    posts = BrowserlessScraper.new(source).scrape_website
    posts.each do |post|
      ScrapedPost.find_or_create_by!(source: source, source_uid: post[:uid]) do
|scraped_post|
        scraped_post.assign_attributes(
          title: post[:title],
          original_url: post[:url],
          user_info: post[:user],
          original_timestamp: post[:timestamp],
          original_content: post[:content],
          raw_data: post
        )
      end
    end
  end
end
```

Для автоматизації процесу скрапінгу новин та запуску роботи ScrapeNewsJob з певною періодичністю, було вирішено використовувати sidekiq-scheduler. sidekiq-scheduler – це гем який є додатком до Sidekiq, він дозволяє запускати виконання задач згідно з вказаним графіком [22].

Спочатку, було додано гем 'sidekiq-scheduler' в Gemfile та виконано команду bundle install для його встановлення. Після цього, було створено файл sidekiq.yml у каталозі config, де вказано графік виконання робіт.

Вміст файлу sidekiq.yml:

```
scheduler:
  schedule:
    scrape_rubyflow:
      cron: '0 7,19 * * *'
      class: 'ScrapeNewsJob'
      args:
        - 'rubyflow'
```

У цьому файлі вказано, що робота ScrapeNewsJob повинна виконуватися двічі на день - о 7-й та о 19-й годині за кожен день. Це виконується за допомогою cron-виразу '0 7,19 * * *', який вказує, що задача повинна виконуватися на 0-й хвилині 7-й та 19-й години кожного дня. Додатково, передається аргумент 'rubyflow', що вказує на джерело новин для скрапінгу.

3.6 Створення моделей WeeklyDigest та Post

Для відображення даних на сайті буде використано нові моделі WeeklyDigest та Post. Ці моделі створені за допомогою генератора Ruby on Rails, який допомагає створити нові моделі, а також відповідні міграції для бази даних.

Для початку необхідно було створити модель WeeklyDigest, для чого було виконано наступну команду:

```
rails generate model WeeklyDigest start_date:date end_date:date uuid:uuid
```

Ця команда створює нову модель WeeklyDigest, а також відповідну міграцію для таблиці в базі даних. Міграція включає поля start_date, end_date та uuid.

Код міграції для створення таблиці weekly_digests:

```
class CreateWeeklyDigests < ActiveRecord::Migration[7.1]
  def change
    create_table :weekly_digests do |t|
      t.uuid :uuid, default: "gen_random_uuid()", null: false, index: {unique: true}
      t.date :start_date
      t.date :end_date

      t.timestamps
    end

    add_index :weekly_digests, %i[start_date end_date], unique: true
  end
```

```
end
```

Модель WeeklyDigest визначає відносини з моделлю Post за допомогою has_many :posts. Також вона має прикріплене зображення poster.

Код моделі WeeklyDigest:

```
class WeeklyDigest < ApplicationRecord
  validates :start_date, presence: true
  validates :end_date, presence: true

  has_many :posts, dependent: :restrict_with_error
  has_one_attached :poster
end
```

В Ruby on Rails, has_one_attached є частиною функціоналу Active Storage. Active Storage - це бібліотека, що дозволяє додавати завантаження файлів до моделей додатків Rails.

Active Storage підтримує зберігання файлів на різних сервісах зберігання, включаючи локальні диски та хмарні сервіси, такі як Amazon S3, Google Cloud Storage та Microsoft Azure Storage. Це дозволяє, легко змінити місце зберігання файлів без зміни коду аплікації.

Додатково, Active Storage надає засоби для прямого завантаження файлів з браузера на сервіс зберігання, бібліотеки для трансформації завантажених зображень (наприклад, для створення мініатюр) та вбудовану підтримку для завантаження файлів, які пов'язані з Active Record об'єктами.

Подібно до WeeklyDigest, модель Post створюється за допомогою генератора Ruby on Rails. Було використано такий ж генератор що і для WeeklyDigest з іншими параметрами:

```
rails generate model Post external_url:string published_at:datetime title:string uuid:uuid scraped_post_id:bigint weekly_digest_id:bigint
```

Після цього міграцію для створення таблиці posts:

```
class CreatePosts < ActiveRecord::Migration[7.1]
  def change
```

```
    create_table :posts do |t|
      t.uuid :uuid, null: false, default: -> { "gen_random_uuid()" }, index: {unique: true}
      t.string :external_url
      t.string :title
      t.timestamp :published_at
      t.references :scraped_post, null: true, foreign_key: true
      t.references :weekly_digest, null: false, foreign_key: true

      t.timestamps
    end
  end
end
```

Важливими атрибутами моделі Post є external_url, published_at, title та uuid. external_url – це URL-адреса оригінального джерела новини, published_at – час публікації новини, title - заголовок новини, а uuid – унікальний ідентифікатор для кожного поста. scraped_post_id та weekly_digest_id – це ідентифікатори, що встановлюють зв'язок з відповідними моделями ScrapedPost та WeeklyDigest.

Код моделі Post:

```
class Post < ApplicationRecord
  has_rich_text :content

  belongs_to :scraped_post, optional: true
  belongs_to :weekly_digest, optional: true

  validates :title, presence: true
  validates :external_url, format: {with: URI::DEFAULT_PARSER.make_regexp},
allow_blank: true

  before_validation -> { self.user ||= Current.user }, on: :create
  after_initialize :set_published_at
  before_validation :set_weekly_digest, on: :create

  private

  def set_published_at
```

```
    self.published_at ||= if scraped_post.present?
      scraped_post.original_timestamp
    else
      Time.current
    end
  end

  def set_weekly_digest
    self.weekly_digest ||= WeeklyDigest.for_date(published_at.presence || Time.current)
  end
end
```

Модель Post в Ruby on Rails включає в себе зв'язки, валідації та колбеки. Вона має зв'язок belongs_to з моделями ScrapedPost та WeeklyDigest, що дозволяє пов'язати пост з відповідними моделями. Зауважимо, що ці зв'язки встановлюються як необов'язкові за допомогою параметру optional: true. Це означає, що пост може існувати без асоціації з моделями ScrapedPost або WeeklyDigest.

Валідації перевіряють, що title присутній перед збереженням поста, а також, що external_url відповідає формату URL. Якщо external_url не задано, воно може бути пустим (allow_blank: true).

Колбек before_validation встановлює користувача (self.user) для поста як Current.user перед його валідацією при створенні.

Колбек after_initialize :set_published_at встановлює значення published_at після ініціалізації об'єкта Post, використовуючи часову мітку оригінального поста, якщо він присутній, або поточний час.

Колбек before_validation :set_weekly_digest, on: :create встановлює weekly_digest для поста перед його валідацією при створенні, використовуючи метод WeeklyDigest.for_date з датою публікації поста або поточним часом.

has_rich_text це метод, що надається Rails за допомогою Action Text. Action Text - це фреймворк, що входить до складу Rails, який дозволяє

вбудовувати та обробляти багатоформатний текст (rich text) або текст з розміткою (markup) в моделях Rails.

Коли використовується has_rich_text :content в моделі (в даному випадку, моделі Post), це створює зв'язок між моделлю Post та новим екземпляром ActionText::RichText з ім'ям content. Це означає, що можна додати багатоформатний текст до поста, використовуючи post.content.

Цей багатоформатний текст може включати HTML-розмітку, вставки зображень, відео та інших медіафайлів, а також інші формати тексту (наприклад, жирний, курсив, підкреслення тощо).

Використання has_rich_text замість звичайного поля тексту дає більше гнучкості при роботі з текстом, особливо коли потрібно включити розширений форматування або медіа-вміст. Також, Action Text забезпечує автоматичне екранування (escaping) небезпечного HTML, що робить його безпечним для використання.

3.7 Створення функціоналу автентифікації

Автентифікація є однією з ключових функцій будь-якого веб-застосунку, який передбачає роботу з даними та модерацію контенту. Для реалізації автентифікації в проекті було вирішено використовувати гем Devise через його гнучкість та легкість в налаштуванні.

Для встановлення Devise, спочатку необхідно було додати гем в Gemfile [23].

Після цього було виконано команду bundle install, щоб встановити гем. Наступним кроком було виконання команди rails generate devise:install, яка створює необхідні конфігураційні файли та ініціалізаційні скрипти.

За допомогою команди rails generate devise User було створено модель User, яка містить необхідні поля для автентифікації, такі як email і password.

Код моделі User:

```
class User < ApplicationRecord
```

```
  devise :database_authenticatable, :lockable, :trackable,
    :rememberable, :validatable

  enum role: {simple: "simple", admin: "admin"}
end
```

Для підтримки ролей користувачів, до моделі User було додано поле role з типом string. Для цього було згенеровано міграцію:

```
rails generate migration add_role_to_users role:string
```

На даному етапі розробки системи аутентифікації було вирішено обмежитися можливістю входу в систему. Функціонал реєстрації нових користувачів на даний момент відсутній, це пов'язано з необхідністю контролю над тим, хто має доступ до системи. Адміністратори системи будуть додаватися вручну через консоль, що забезпечує більшу безпеку та контроль над процесом.

Враховуючи можливість використовувати функціонал devise для того щоб закривати доступ до певних сторінок для не адмінів, було вирішено додати Sidekiq Web, він дозволяє переглянути статистику роботи Sidekiq у реальному часі, але оскільки цей інструмент повинен бути захищений, в маршрутизації проекту за допомогою блоку authenticate, доступ до Sidekiq Web обмежено тільки користувачами з роллю 'admin'. Це реалізовано за допомогою лямбда-функції, яка перевіряє, чи є користувач адміністратором:

Вміст файлу routes.rb:

```
require "sidekiq/web"
require "sidekiq-scheduler/web"

Rails.application.routes.draw do
  devise_for :users, only: :sessions, controllers: {sessions: "users/sessions"}

  authenticate :user, ->(u) { u.admin? } do
    mount Sidekiq::Web => "/sidekiq"
  end
end
```

3.8 Створення адмін панелі

Для організації адміністративних функцій було створено адміністративний простір імен "Account". Всі контроллери адміністративної частини успадковуються від базового контроллера Account::BaseController, що забезпечує спільні методи та фільтри для них.

Account::BaseController перевіряє, чи є користувач автентифікованим, та чи має він роль адміністратора. Це реалізовано за допомогою зворотніх викликів (before_action), які виконуються перед кожною дією контроллера. Якщо користувач не є адміністратором, він буде перенаправлений на головну сторінку з повідомленням про неавторизований доступ.

Код контроллера Account::BaseController:

```
class Account::BaseController < ApplicationController
  before_action :authenticate_user!
  before_action :check_admin

  layout "account"

  private

  def check_admin
    redirect_to root_path, alert: "You are not authorized to access this page" unless
current_user.admin?
  end
end
```

Account::PostsController відповідає за створення нових постів у системі. Нові пости можуть бути створені на основі вже зіскрейплених даних або вручну.

Код контроллера Account::PostsController:

```
class Account::PostsController < Account::BaseController
  def new
    scraped_post = ScrapedPost.find_by(id: params[:scraped_post_id])
    @post = scraped_post.present? ? Post.from_scraped_post(scraped_post) : Post.new
```

```
  end

  def create
    @post = Post.new(post_params)
    if @post.save
      redirect_to account_root_path, notice: "Post was successfully created."
    else
      render :new, status: :unprocessable_entity
    end
  end

  def preview_link
    @metadata =
MetadataScraper.new.scrape_website(Base64.urlsafe_decode64(params[:url]))
  end

  private

  def post_params
    params.require(:post).permit(:title, :content, :external_url, :scraped_post_id,
:published_at)
  end
end
```

Екшн new визначає, як створити новий екземпляр Post. Якщо передається параметр :scraped_post_id, пост створюється на основі відповідного ScrapedPost. Інакше створюється порожній пост.

Екшн create обробляє форму створення нового посту. Якщо збереження нового посту проходить успішно, користувач буде перенаправлений на головну сторінку адмін-панелі з повідомленням про успішне створення посту. Якщо виникли помилки, буде показана форма створення посту з повідомленнями про помилки.

Всі параметри, необхідні для створення посту, визначаються в приватному методі post_params. Цей метод використовує механізм strong parameters, який допомагає захистити приложення від небезпечних вводу

даних, забезпечуючи тільки допустимі параметри для масового призначення.

Екшн preview_link витягує метадані веб-сторінки за її URL. Це використовується для відображення більшої кількості даних в формі поста коли він створюється основі ScrapedPost, оскільки посилання згадані в оригіналі будуть відображатися з більш детальною інформацією, таким чином можна буде краще зрозуміти зміст новини чи поста непокидаючи сторінки з формою

Слід зазначити, що більшість екшинів цього контроллера мають відповідне представлення (view), що забезпечує користувацький інтерфейс для взаємодії з даним контроллером. Наприклад, представлення для дії new містить форму для створення нового посту, а представлення для дії preview_link відображає метадані з відповідного посилання в формі.

Account::ScrapedPostsController відповідає за список зібраних постів, які відображаються в адміністративному інтерфейсі. Метод index відповідає за відображення цього списку.

Код контроллера Account::ScrapedPostsController:

```
class Account::ScrapedPostsController < Account::BaseController
  def index
    scraped_posts = ScrapedPost.ordered.includes(:post)
    scraped_posts = scraped_posts.by_status(params[:status]) unless params[:status]
== "all"

    @pagy, @scraped_posts = pagy(scraped_posts)
    @facets = ScrapedPost.group(:status).count.tap do |facets|
      facets["all"] = facets.values.sum
    end.sort_by { |k, _| k }.to_h
  end
end
```

У цьому контроллері використовується модель ScrapedPost для отримання списку зібраних постів і фільтрування їх за статусом, якщо

такий параметр вказано. Це дозволяє відфільтровувати пости за статусом, створюючи більш гнучкий і зручний інтерфейс для користувача.

Потім, з допомогою гему pagy, відбувається пагінація результатів, щоб вони відображались на сторінках. Це дозволяє керувати великою кількістю зіскрейплених постів, розподіляючи їх між сторінками для зручності перегляду.

Використовуючи ScrapedPost.group(:status).count, збирається статистика по статусу постів. Це створює об'єкт, який відображає кількість постів для кожного статусу. Результат сортується по ключах для зручності відображення.

Кожен зібраний пост має посилання на форму для публікації. Це дозволяє адміністраторам легко перейти до публікації конкретного посту прямо зі сторінки перегляду зібраних постів (рис. 3.1).

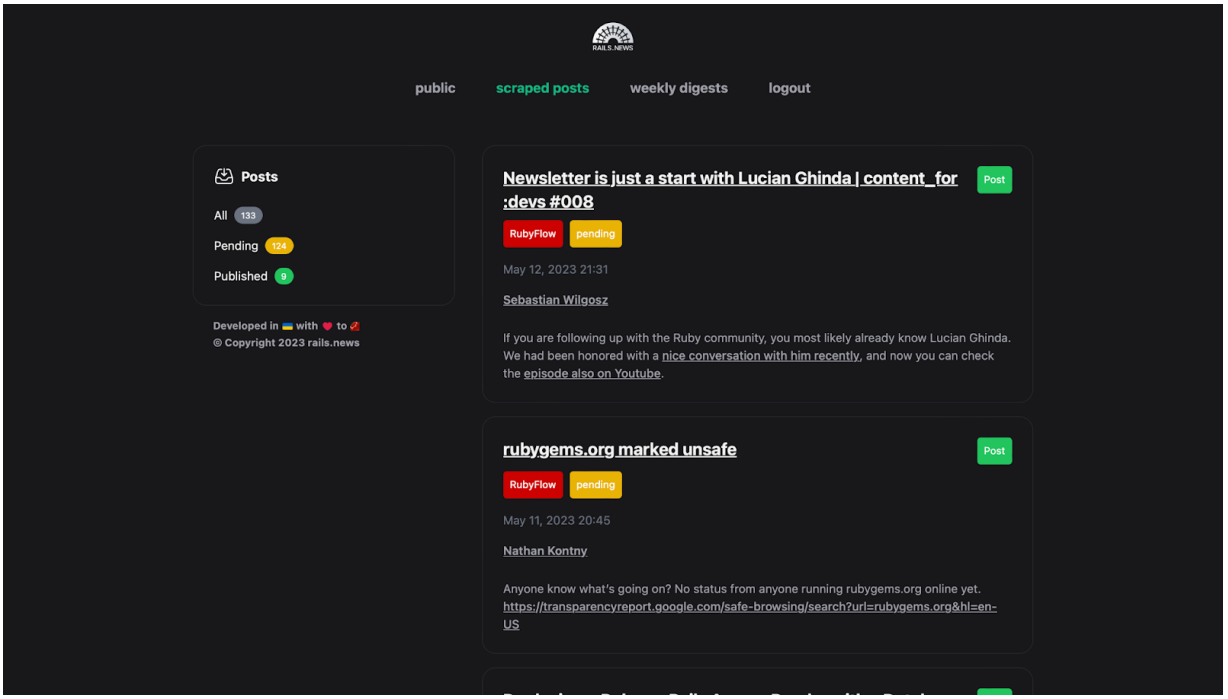

Рисунок 3.1 – Сторінка зібраних постів в адмін панелі

Account::WeeklyDigestsController дозволяє адміністраторам переглядати та редагувати щотижневі дайджести. Варто зауважити, що в даному контроллері відсутній функціонал створення щотижневих

дайджестів, оскільки вони будуть генеруватися автоматично при створенні нових постів.

Код Account::WeeklyDigestsController:

```
class Account::WeeklyDigestsController < Account::BaseController
  def index
    @pagy, @weekly_digests = pagy(WeeklyDigest.order(start_date: :desc))
  end

  def edit
    @weekly_digest = WeeklyDigest.find_by!(uuid: params[:id])
  end

  def update
    @weekly_digest = WeeklyDigest.find_by!(uuid: params[:id])

    if @weekly_digest.update(weekly_digest_params)
        redirect_to account_weekly_digests_path, notice: "Weekly digest was successfully
updated."
    else
      render :edit, status: :unprocessable_entity
    end
  end

  private

  def weekly_digest_params
    params.require(:weekly_digest).permit(:poster)
  end
end
```

Основні методи цього контроллера включають:

- index: відображає список всіх щотижневих дайджестів, відсортованих за датою початку (start_date) у зворотньому порядку. Ми використовуємо гем pagy для пагінації результатів;
- edit: знаходить щотижневий дайджест за його uuid та передає його у форму редагування;

- update: спробує оновити щотижневий дайджест з даними, отриманими від форми редагування. Якщо оновлення успішне, адміністратора перенаправляють на сторінку зі списком щотижневих дайджестів та отримують повідомлення про успішне оновлення. В іншому випадку, адміністратор повертається до форми редагування з відповідним статусом.

У приватному методі weekly_digest_params визначено дозволені параметри для щотижневих дайджестів, які можуть бути оновлені через цей контроллер. У даному випадку, адміністратори можуть редагувати лише параметр poster.

Було використано shared_examples, що дозволяють описувати спільну поведінку для різних контекстів в тестах Rspec. Це корисно, коли є кілька тестів, які виконують подібні дії, в даному випадку, перевірку на те що тільки автентифікований адмін має доступ до адмін панелі.

Код shared_example “protected path”:

```
RSpec.shared_examples "a protected path" do |method, path, params = {}|
  context "when logged in as a non-admin" do
    let(:user) { create(:user) }

    before do
      sign_in user
    end

    it "redirects to the login page" do
      send(method, path.is_a?(Symbol) ? send(path, **params) : path)

      expect(response).to be_redirect
      expect(flash[:alert]).to eq("You are not authorized to access this page")
    end
  end
end
```

Код тесту контроллера Account::ScrapedPosts:

```
RSpec.describe "Account::ScrapedPosts", type: :request do
  before { sign_in create(:user, :admin) }
```

```
  describe "GET /index" do
    it_behaves_like "a protected path", :get, :account_root_path
    let!(:scraped_post) { create(:scraped_post) }
    let!(:scraped_post2) { create(:scraped_post, :published) }
    let!(:scraped_post3) { create(:scraped_post, :rejected) }

    it "returns http success" do
      get account_root_path
      expect(response).to have_http_status(:success)
      expect(response.body).to include(scraped_post.title)
      expect(response.body).not_to include(scraped_post2.title)
      expect(response.body).not_to include(scraped_post3.title)
    end

    it "shows facets" do
      create_list(:scraped_post, 2, :published)
      create_list(:scraped_post, 1, :rejected)

      get account_root_path
      expect(response.body).to include(ScrapedPost.count.to_s)
      expect(response.body).to include(ScrapedPost.pending.count.to_s)
      expect(response.body).to include(ScrapedPost.published.count.to_s)
      expect(response.body).to include(ScrapedPost.rejected.count.to_s)
    end
  end
end
```

Тести для Account::ScrapedPostsController включають:

- перевірку, чи повертає запит GET /index успішний HTTP статус, та чи включає відповідь тіло поста, що очікується. Також переконуємося, що відповідь не включає тіла постів з статусом "опубліковано" або "відхилено";
- перевірку на наявність фасет (категорій) в відповіді. Ми створюємо кілька постів з різними статусами і переконуємося, що кількість постів у кожній категорії відповідає очікуваному значенню.

Ці тести допомагають перевірити, що адмін панель працює належним чином і відповідає нашим вимогам до функціоналу. Вони є важливою частиною процесу розробки, оскільки дозволяють виявити та виправити проблеми на ранніх стадіях.

3.9 Налаштування ViewComponent та Lookbook

ViewComponent – це фреймворк для Ruby on Rails, який дозволяє розробникам створювати повторно використовувані, самодостатні компоненти вигляду [24]. Ці компоненти можуть бути тестовані незалежно один від одного, що значно спрощує процес розробки і покращує якість кінцевого продукту.

Для ініціалізації ViewComponent потрібно додати гем 'view_component' до Gemfile та виконати bundle install. Після цього потрібно виконати команду rails generate view_component ExampleComponent, яка створить новий компонент з іменем ExampleComponent.

У порівнянні зі засобами які передбачені в Ruby on Rails (паршиали), ViewComponent має ряд переваг:

- тестування: Компоненти ViewComponent легше тестувати, оскільки вони є самодостатніми і можуть бути тестованими незалежно один від одного. Це відрізняється від паршиалів, які зазвичай тестуються в контексті вигляду, в якому вони використовуються;
- перевикористання: Компоненти ViewComponent легше перевикористати, оскільки вони мають визначений інтерфейс і можуть бути використані в різних виглядах без змін;
- ефективність: ViewComponent ефективніше використовує ресурси, та є більш оптимізованим в порівнянні з паршиалами.

Lookbook – це інструмент для документації і демонстрації компонентів ViewComponent [25]. Він дозволяє розробникам легко

створювати і переглядати приклади компонентів в дії. Для налаштування Lookbook необхідно зробити наступні кроки:

- додати гем 'lookbook' до Gemfile;
- виконати команду bundle install для встановлення Lookbook;
- монтувати роутер Lookbook на обраному шляху у файлі config/routes.rb:

```
Rails.application.routes.draw do
  if Rails.env.development?
    mount Lookbook::Engine, at: "/lookbook"
  end
end
```

Після встановлення, Lookbook буде доступним на шляху /lookbook (рис. 3.2).

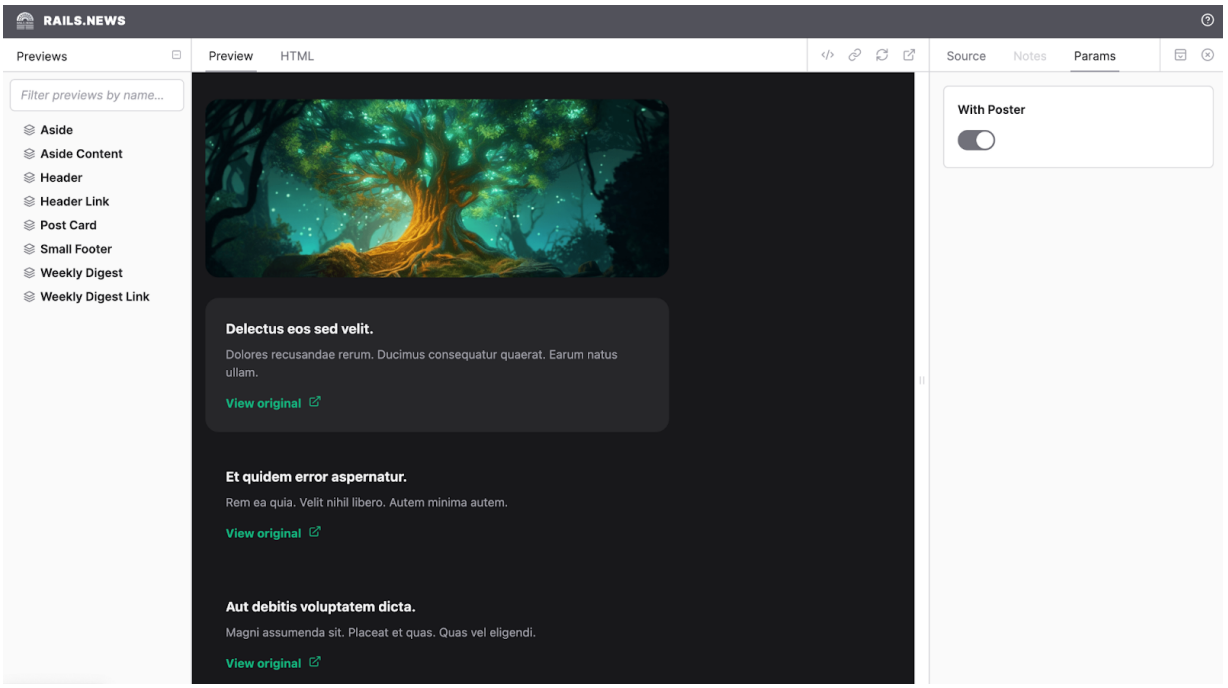

Рисунок 3.2 – Інтерфейс Lookbook

3.10 Створення публічних сторінок

Після створення всіх необхідних моделей, налаштування адміністративної панелі та розробки компонентів відповідно до макету в Figma, було розпочато роботу над головними сторінками сайту.

Спочатку було створено контроллер HomeController, який відповідає за відображення головної сторінки сайту. Екшн index цього контроллера отримує останній випуск щотижневого дайджесту, що отримується за допомогою методу "latest!" моделі WeeklyDigest (рис. 3.3).

Код контроллера HomeController:

```
class HomeController < ApplicationController
  def index
    @weekly_digest = WeeklyDigest.latest!
  end
end
```

Наступним етапом було створення контроллера "WeeklyDigestsController", який відповідає за відображення сторінок з архівом та окремих випусків щотижневого дайджесту.

Код контроллера WeeklyDigestsController:

```
class WeeklyDigestsController < ApplicationController
  def index
    @weekly_digest = WeeklyDigest.latest!

    redirect_to @weekly_digest
  end

  def show
    @weekly_digests = collection
    @weekly_digest = @weekly_digests.find_by!(uuid: params[:id])
  end

  private

  def collection
```

```
    WeeklyDigest.for_archive
  end
end
```

Екшн index цього контроллера також отримує останній випуск щотижневого дайджесту, а потім перенаправляє користувача на сторінку цього випуску.

Екшн show використовується для відображення окремого випуску дайджесту. Він отримує колекцію всіх випусків дайджесту, які призначені для архіву (за допомогою методу "for_archive"), а потім знаходить серед них випуск з потрібним UUID.

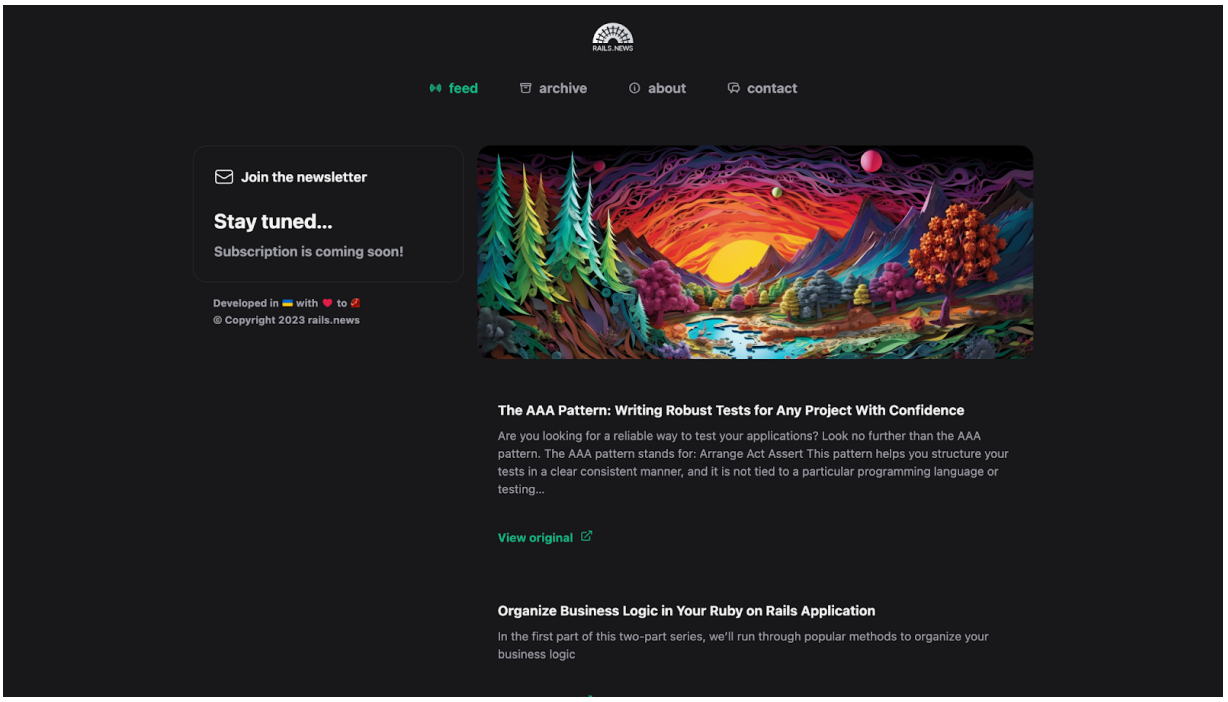

Рисунок 3.3 – Головна сторінка готового сайту

Варто зауважити, що для роботи цього контроллера в моделі WeeklyDigest було додано два нових метода: for_archive та latest!.

Метод for_archive є областю (scope), який визначає SQL-запит до бази даних. Він повертає 12 останніх випусків щотижневого дайджесту, відсортованих за датою початку в порядку спадання. За допомогою методу

joins(:posts) з БД витягуються тільки ті випуски, в яких є пости, а метод distinct гарантує, що кожен випуск включено лише один раз.

Код скоупа for_archive:

```
scope :for_archive, -> { order(start_date: :desc).joins(:posts).distinct.limit(12) }
```

Метод latest! використовує for_archive для отримання останнього випуску дайджесту. Оператор "!" в кінці назви методу означає, що в разі, якщо жоден випуск не знайдено, буде викинуто виключення ActiveRecord::RecordNotFound, що скаже Ruby on Rails про те що необхідно відобразити сторінку 404.

Код методу latest!:

```
def self.latest!
  for_archive.first!
end
```

3.11 Розгортання веб-сайту в продакшені

Для розгортання веб-сайту "rails.news" на сервері використовується інструмент MRSK. Перш ніж розгортати веб-сайт, необхідно встановити MRSK за допомогою команди gem install mrsk.

Після встановлення MRSK потрібно перейти в директорію веб-сайту і виконати команду mrsk init. Наступним кроком є налаштування файла config/deploy.yml, який має наступний вигляд для веб-сайту "rails.news":

```
service: railsnews

image: manastyretskyi/railsnews

servers:
  web:
    - 178.128.255.245
  job:
    hosts:
      - 178.128.255.245
    cmd: bundle exec sidekiq -e production
```

```
registry:
  username: manastyretskyi
  password:
    - MRSK_REGISTRY_PASSWORD

env:
  clear:
    DB_HOST: 178.128.255.245
    BROWSERLESS_URL: http://178.128.255.245:5001
  secret:
    - RAILS_MASTER_KEY

accessories:
  db:
    image: postgres:15.2-alpine
    host: 178.128.255.245
    port: 5432
    env:
      clear:
        POSTGRES_USER: rails_news
      secret:
        - POSTGRES_PASSWORD
    directories:
      - data:/var/lib/postgresql/data

  redis:
    image: redis:7.0
    host: 178.128.255.245
    port: 6379
    directories:
      - data:/data

  browserless:
    image: manastyretskyi/browserless-chrome:1.58-chrome-stable
    host: 178.128.255.245
    port: 5001
    env:
      clear:
        PORT: 5001
```

```
    CONNECTION_TIMEOUT: 180000
                    FUNCTION_EXTERNALS: '["metascraper","metascraper-audio",
"metascraper-author", "metascraper-date", "metascraper-description",
"metascraper-feed", "metascraper-image", "metascraper-iframe", "metascraper-lang",
"metascraper-logo", "metascraper-publisher", "metascraper-readability",
"metascraper-title", "metascraper-url", "metascraper-video", "metascraper-logo-favicon",
"metascraper-media-provider"]'
```

Секція servers у файлі конфігурації MRSK вказує на сервери, на яких буде розгорнуто веб-сайт. Можна вказати декілька серверів, у випадку якщо необхідно мати розподілену систему для роботи з великою кількістю запитів. Підсекції, такі як web і job, вказують на різні типи серверів або ролі, які вони виконують. Зазвичай web вказує на сервери, які обробляють веб-запити, а job вказує на сервери, призначені для обробки фонових задач, таких як обробка великих даних або завдань, які потребують тривалого виконання. В даному випадку в job запускається команда sidekiq, оскільки sidekiq працює на одній і тій ж код базі що й вебсайт, ніяких додаткових налаштувань не потрібно, таким чином для того щоб розгорнути сервіс sidekiq в продакшені знадобилося тільки декілька строк коду в налаштуваннях MRSK.

Секція registry пов'язана з Docker registry, місцем, де зберігаються образи додатку. Зазвичай, це хмаровий сервіс, такий як Docker Hub, Google Container Registry, чи інший. Секретні дані для доступу до Docker registry (ім'я користувача і пароль) зберігаються у секції env.

Секція env містить конфігураційні змінні середовища, які використовуються вашим додатком. Вони розділені на дві категорії: clear і secret. Змінні clear - це змінні, які не містять конфіденційної інформації, а secret - це змінні, що містять конфіденційні дані, такі як паролі, ключі API та інше. Змінні secret зазвичай беруться з файлу .env.

Секція accessories використовується для налаштування додаткових сервісів, необхідних для роботи додатку, таких як база даних, кешування, пошукові сервіси тощо. Кожен додатковий сервіс (accessory) має власні

налаштування, включаючи образ, хост, порт, середовище і так далі. Ці сервіси запускаються і керуються разом з основним веб-додатком. В даному прикладі конфігурації визначено три допоміжні сервіси: db, redis і browserless:

- db (База даних): Цей сервіс використовує Docker образ PostgreSQL (postgres:15.2-alpine) для створення бази даних. База даних розміщується на хості 178.128.255.245 і слухає порт 5432. Крім того, вказані змінні середовища POSTGRES_USER та POSTGRES_PASSWORD, що використовуються для автентифікації. Вказано також директорію, де будуть зберігатися дані бази даних;
- redis: Цей сервіс використовує Docker образ Redis (redis:7.0). Цей сервер розміщується на хості 178.128.255.245 і слухає порт 6379. Вказано також директорію, де будуть зберігатися дані бази даних;
- browserless (Хедлес браузер): Цей сервіс використовує Docker образ browserless (manastyretskyi/browserless-chrome:1.58-chrome-stable), той образ який було створено в підрозділі 3.3, для створення хедлес браузера. Він також розміщується на хості 178.128.255.245, слухає порт 5001. Змінні середовища вказують налаштування для сервісу Browserless, а саме, вказується які зовнішні бібліотеки мають бути доступні для використання, це необхідно для коректної роботи MetadataScraper.

Після налаштування файлу deploy.yml та файлу .env з необхідними креденшиалами, можна розгорнути веб-сайт на сервері за допомогою команди mrsk deploy.

Ця команда виконує наступні дії:

- з'єднується з серверами через SSH (використовуючи root за замовчуванням, аутентифікація відбувається за допомогою ssh ключа користувача);
- встановлює Docker на сервері, якщо він ще не встановлений;
- виконує вхід в реєстр для локальної та віддаленої систем;

- створює образ Docker використовуючи Dockerfile в кореневій директорії додатку;
- завантажує образ в реєстр;
- завантажує образ з реєстру на сервер;
- запускає контейнери з новою версією додатку;
- зупиняє контейнери з попередньою версією додатку;
- видаляє не використовувані образи та зупинені контейнери;
- після виконання цих дій, веб-сайт "rails.news" доступний в продакшені на зазначеному сервері.

Висновки до розділу 3

У даному розділі детально розглянуто процес розробки веб-сайту агрегатора новин на основі Ruby on Rails: створено базову аплікацію та налаштовано тестове середовище, розроблено сервіс веб-скрапінгу для збору інформації з інтернет-ресурсів. Спеціально розроблена модель дозволяє зберігати зібрану інформацію у структурованому вигляді, спрощуючи її подальше використання. Для оптимальної обробки великої кількості запитів та їх подальшого виконання в фоновому режимі реалізовано використання Sidekiq. Розроблено моделі WeeklyDigest та Post для створення системи регулярного надсилання дайджестів новин та зберігання окремих публікацій. Функціонал автентифікації реалізовано для захисту адміністративної частини сайту від несанкціонованого доступу, було створено адмін панель для керування ресурсами сайту, налаштовано ViewComponent та Lookbook для ефективного створення та демонстрації компонентів інтерфейсу. Створено публічні сторінки сайту з допомогою цих інструментів. В результаті, було розроблено повноцінний веб-сайт з системою збору та обробки новин, системою автентифікації користувачів та адміністративним функціоналом.

ВИСНОВКИ

У рамках дослідження було детально вивчено предметну область проекту, яка охоплює аналіз новин та інформації зі світу Ruby і Ruby on Rails. Дослідження двох існуючих аналогів - RubyWeekly та LibHunt Ruby - показало недоліки у їхній системі відбору та публікації новин, обмеженість архіву та застарілий дизайн сайтів. Особистим внеском була розробка веб-сайту агрегатора новин з використанням Ruby on Rails, що включає в себе елементи веб-скрапінгу для збору новин з різних джерел.

При виборі інструментів для розробки сайту було враховано їхню ефективність, надійність та доступність, що дозволило створити стабільний та продуктивний веб-ресурс. Використання MRSK для розгортання додатку в продакшені дозволило забезпечити його гнучкість, автономність та ефективність.

Щодо адміністративного функціоналу, було створено панель адміністратора для керування ресурсами сайту та захищено адміністративну частину сайту від несанкціонованого доступу. Для ефективного створення та демонстрації компонентів інтерфейсу було налаштовано ViewComponent та Lookbook, за допомогою яких було створено публічні сторінки сайту.

В результаті, було створено повноцінний веб-сайт, що виконує всі поставлені завдання: збирає, організовує та відображає новини для спільноти Ruby і Ruby on Rails.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Агрегатори новин. Wikipedia.com. веб-сайт. URL: https://en.wikipedia.org/wiki/News_aggregator (дата звернення: 25.03.2023)

2. Веб-скрапінг. Wikipedia.com. веб-сайт. URL: https://en.wikipedia.org/wiki/Web_scraping (дата звернення: 27.03.2023)

3. Веб-сайт Nokogiri. веб-сайт. URL: https://nokogiri.org/index.html (дата звернення: 18.04.2023)

4. Документація Puppeteer. веб-сайт. URL: https://pptr.dev/ (дата звернення: 18.04.2023)

5. Документація сервісу Browserless. веб-сайт. URL: https://www.browserless.io/docs/start (дата звернення: 18.04.2023)

6. Документація Ruby on Rails. веб-сайт. URL: https://edgeguides.rubyonrails.org/ (дата звернення: 20.04.2023)

7. Документація БД PostgreSQL. веб-сайт. URL: https://www.postgresql.org/docs/ (дата звернення: 20.04.2023)

8. Документація БД Redis. веб-сайт. URL: https://redis.io/docs/ (дата звернення: 20.04.2023)

9. Хедлесс браузер. Wikipedia.com. веб-сайт. URL: https://en.wikipedia.org/wiki/Headless_browser (дата звернення: 20.04.2023)

10. Документація бібліотеки Sidekiq. веб-сайт. URL: https://github.com/sidekiq/sidekiq (дата звернення: 20.04.2023)

11. Документація сховища S3. веб-сайт. URL: https://aws.amazon.com/s3/ (дата звернення: 20.04.2023)

12. Reverse-proxy. Wikipedia.com. веб-сайт. URL: https://en.wikipedia.org/wiki/Reverse_proxy (дата звернення: 20.04.2023)

13. Веб-сайт Figma. веб-сайт. URL: https://www.figma.com/ (дата звернення: 25.04.2023)

14. Документація Midjourney. веб-сайт. URL: https://docs.midjourney.com/docs/quick-start (дата звернення: 25.04.2023)

15. Документація MRSK. веб-сайт. URL: https://github.com/mrsked/mrsk (дата звернення: 25.04.2023)

16. Документація Docker. веб-сайт. URL: https://docs.docker.com/get-started/ (дата звернення: 2.05.2023)

17. Веб-сайт RSpec. веб-сайт. URL: https://rspec.info/ (дата звернення: 2.05.2023)

18. Документація бібліотеки simplecov. веб-сайт. URL: https://github.com/simplecov-ruby/simplecov (дата звернення: 2.05.2023)

19. Документація бібліотеки factory_bot. веб-сайт. URL: https://github.com/thoughtbot/factory_bot (дата звернення: 2.05.2023)

20. Документація бібліотеки faker. веб-сайт. URL: https://github.com/faker-ruby/faker (дата звернення: 2.05.2023)

21. Документація бібліотеки Metascraper. веб-сайт. URL: https://github.com/microlinkhq/metascraper (дата звернення: 3.05.2023)

22. Документація бібліотеки sidekiq-scheduler. веб-сайт. URL: https://github.com/sidekiq-scheduler/sidekiq-scheduler (дата звернення: 3.05.2023)

23. Документація Devise. веб-сайт. URL: https://github.com/heartcombo/devise (дата звернення: 6.05.2023)

24. Документація ViewComponent. веб-сайт. URL: https://viewcomponent.org/ (дата звернення: 8.05.2023)

25. Документація Lookbook. веб-сайт. URL: https://lookbook.build/guide (дата звернення: 8.05.2023)