МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА НАВЧАЛЬНОЇ ВЕБПЛАТФОРМИ ДЛЯ ІНТЕРПРЕТАТОРА CLOJURE»**

Виконав: студент 2 курсу, групи 6.1213-пі-с3
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія (зі скороченим терміном навчання)
(назва освітньої програми)
К.М. Жилка
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, доцент, к.ф.-м.н. Горбенко В.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Бюро С.Ю.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія (зі скороченим терміном навчання)

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Жилці Кирилу Миколайовичу
(прізвище, ім’я та по батькові)

1. Тема роботи Розробка навчальної вебплатформи для інтерпретатора Clojure

керівник роботи Горбенко Віталій Іванович, к.ф.-м.н., доцент
(прізвище, ім’я та по батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Технічне завдання.
2. Проєктування системи.
3. Програмна реалізація вебсторінки.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 26.12.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	08.01.2025	
2.	Збір вихідних даних.	31.01.2025	
3.	Обробка методичних та теоретичних	03.03.2025	
	джерел.		
4.	Розробка першого та другого розділу.	14.04.2025	
5.	Розробка третього розділу.	16.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	20.06.2025	

Студент _____ К.М. Жилка
(підпис) (ініціали та прізвище)

Керівник роботи _____ В.І. Горбенко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка навчальної вебплатформи для інтерпретатора Clojure»: 48 с., 36 рис., 21 джерело.

БІБЛІОТЕКИ, ВЕБПЛАТФОРМА, ІНТЕРПРЕТАТОР, НАВЧАННЯ, ПРОГРАМУВАННЯ, CLOJURE, CLOJURESCRIPT.

Об’єкт дослідження – процес інтерактивної розробки та виконання програмного коду мовою Clojure у вебсередовищі на основі технологій ClojureScript, CodeMirror та SCI (Small Clojure Interpreter).

Мета роботи: розробити інтерактивний вебредактор для мови програмування Clojure з підтримкою підсвічування синтаксису, відстеження помилок та інтегрованим механізмом виконання коду на клієнтській стороні.

Методи дослідження: у роботі застосовано аналіз та узагальнення наявних інструментів інтерпретації коду, метод системного підходу до проєктування програмних компонентів, а також метод моделювання взаємодії між користувачем та середовищем розробки з використанням бібліотек Reagent, CodeMirror 6 та SCI.

У роботі показано процес проєктування та створення вебсторінки інтерпретатора Clojure, застосуванні мови програмування Clojure, ClojureScript та допоміжні бібліотеки. Реалізовано вебсторінку із загальною інформацією про платформу та можливістю перейти до редактора, у якому можна написати та виконати код і отримати результати його виконання.

SUMMARY

Bachelor's qualifying paper "Development of a Learning Web Platform for the Clojure Interpreter": 48 page, 36 figures, 21 references.

CLOJURE, CLOJURESCRIPT, INTERPRETER, LEARNING, LIBRARIES, PROGRAMMING, WEB PLATFORM.

The object of the study is the process of interactive development and execution of Clojure code in a web environment based on ClojureScript, CodeMirror, and SCI (Small Clojure Interpreter) technologies.

The aim of the study is to develop an interactive web-based editor for the Clojure programming language with support for syntax highlighting, error tracking, and an integrated client-side code execution mechanism.

The methods of research are the analysis and generalization of existing code interpretation tools, the system approach to designing software components, and the modeling of user-environment interaction using the Reagent, CodeMirror 6, and SCI libraries.

The work demonstrates the process of designing and creating a web page for a Clojure interpreter, using the programming languages Clojure and ClojureScript, along with supporting libraries. A web page has been implemented that provides general information about the platform and allows users to navigate to an editor where they can write and execute code and see the results of its execution.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 7
1 Технічне завдання 9
1.1 Огляд предметної області функціонального програмування 9
1.2 Огляд предметної області функціонального програмування 10
1.3 Аналіз технологій розробки вебплатформ 12
1.4 Основні функціональні вимоги до інтерпретатора 13
1.5 Основні нефункціональні вимоги до вебдодатка 14
2 Проєктування системи 17
2.1 Огляд вибору технологій та інструментарію 17
2.1.1 Основна мова програмування 17
2.1.2 Ключові бібліотеки для розробки системи 18
2.1.3 Допоміжні інструменти при розробці та стилізація 20
2.2 Структура клієнтської частини та взаємодія з користувачем 21
3 Програмна реалізація вебсторінки 25
3.1 Підготовка середовища розробки 25
3.2 Створення основи вебсайту 27
3.3 Розробка допоміжних компонентів вебсайту 33
3.4 Створення основних сторінок вебсайту 36
3.5 Розробка редактору коду 39
3.6 Створення функціоналу виконання коду 42
Висновки 46
Перелік посилань 47

ВСТУП

У сучасному світі інформаційних технологій розробка вебінструментів для програмістів є важливим напрямом. Зростання інтересу до функціональних мов програмування, зокрема Clojure, зумовлює необхідність створення доступних засобів для їхнього вивчення та практичного застосування. Розробка вебінтерпретатора Clojure надасть можливість інтерактивно виконувати код, що сприятиме кращому розумінню мови.

Мета дослідження полягає у розробці інтерактивного вебредактора для мови програмування Clojure з підтримкою підсвічування синтаксису, відстеження помилок та інтегрованим механізмом виконання коду на клієнтській стороні.

Для досягнення мети поставлено наступні завдання:

– проаналізувати принципи побудови інтерпретаторів мов програмування;
– дослідити засоби, що дозволять реалізувати виконання Clojure коду в інтерпретаторі;
– розробити архітектуру вебплатформи інтерпретатора Clojure;
– спроєктувати вебінтерфейс для введення та виконання коду Clojure;
– виконати тестування працездатності розробленого інтерпретатора.

Об’єктом дослідження є процес інтерактивної розробки та виконання програмного коду мовою Clojure у вебсередовищі на основі технологій ClojureScript, CodeMirror та SCI (Small Clojure Interpreter).

Предметом дослідження є вебплатформа, що реалізує функціональність інтерпретатора мови Clojure. Об’єктом є процес інтерпретації Clojure коду, а предметом – вебдодаток, який забезпечує цю інтерпретацію.

Методи дослідження включають аналіз наявних інструментів інтерпретації коду, системний підхід до проєктування компонентів та моделювання взаємодії між користувачем та середовищем розробки.

Для реалізації проєкту використано мову Clojure та ClojureScript, які дозволяють якісно реалізувати клієнтську частину. Розроблений вебінтерпретатор надає користувачам можливість вводити та виконувати код Clojure без необхідності встановлення додаткового програмного забезпечення. Очікується, що розроблений інтерпретатор стане корисним інструментом для вивчення та експериментів з мовою Clojure.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та переліку посилань. У першому розділі розглядаються теоретичні аспекти побудови інтерпретаторів та особливості мови Clojure, у другому – проєктування архітектури вебінтерпретатора, у третьому – опис процесу реалізації та результати тестування.

1 ТЕХНІЧНЕ ЗАВДАННЯ

1.1 Огляд предметної області функціонального програмування

Вебінтерпретатори являють собою важливий клас інтерактивних вебдодатків, що надають користувачам можливість безпосередньо виконувати код різних мов програмування у веббраузері. Їхня зручність миттєвого доступу без необхідності встановлення локального середовища розробки робить їх особливо цінними в освітньому процесі, дозволяючи початківцям експериментувати з синтаксисом та основними концепціями програмування. Крім того, для досвідчених розробників вебінтерпретатори слугують швидким інструментом для тестування окремих фрагментів коду, обміну ідеями та демонстрації можливостей мови програмування. Враховуючи постійний розвиток вебтехнологій та попит на інтерактивні інструменти, що зростає, розробка якісних вебінтерпретаторів залишається актуальним та важливим напрямком у сфері інформаційних технологій [7].

Галузь вебінтерпретаторів є досить широкою та включає різноманітні підходи до реалізації виконання коду у вебсередовищі [8]. Ці підходи варіюються залежно від багатьох факторів, включаючи складність та особливості цільової мови програмування, вимоги до продуктивності та безпеки, а також цільову аудиторію користувачів. Існують вебінтерпретатори, що виконують код на серверній стороні, надсилаючи його на обробку та повертаючи результат у браузер, а також ті, що частково або повністю обробляють код на клієнтській стороні за допомогою технологій, таких як JavaScript або WebAssembly. Кожен з цих підходів має свої переваги та недоліки, які слід враховувати при виборі архітектури власного вебінтерпретатора.

Важливими аспектами розробки вебінтерпретаторів є забезпечення належного рівня безпеки при виконанні довільного користувацького коду,

особливо в публічно доступних інструментах. Необхідно ретельно продумувати механізми ізоляції виконання коду та обмеження доступу до системних ресурсів сервера або браузера. Оптимізація продуктивності є ключовим фактором для забезпечення комфортного досвіду користувача, особливо при виконанні ресурсомістких обчислень. Вибір ефективних алгоритмів інтерпретації або компіляції «на льоту» (Just-In-Time compilation), а також оптимізація передачі даних між клієнтом та сервером, є важливими завданнями. Крім того, розробка інтуїтивно зрозумілого та зручного користувацького інтерфейсу для введення коду з можливістю підсвічування синтаксису, відображення результатів виконання у зрозумілому форматі та надання інформативних повідомлень про помилки є важливим елементом успішного вебінтерпретатора.

1.2 Огляд предметної області функціонального програмування

Існує значна кількість вебплатформ, що надають користувачам можливість виконувати код різних мов програмування безпосередньо у веббраузері [9, 10, 12]. Ці платформи відрізняються за своєю функціональністю, архітектурою та технологіями, що лежать в їх основі.

Серед вебплатформ для виконання коду різних мов програмування можна виділити такі популярні приклади: OnlineGDB [9], JSFiddle та CodePen [10, 12], Google Colaboratory [13], Jdoodle [14].

Вебплатформа OnlineGDB є онлайн-компілятором та налагоджувачем для багатьох мов програмування, включаючи C, C++, Python, Java, C#, PHP та інші. OnlineGDB надає користувачам інтегроване середовище розробки прямо у браузері, що включає редактор коду з підсвічуванням синтаксису, консоль для введення та виведення, а також вбудований налагоджувач для полегшення пошуку помилок у коді. Платформа також підтримує збереження коду, обмін ним та можливість спільної роботи в реальному часі.

Платформи JSFiddle та CodePen орієнтовано переважно на вебтехнології (HTML, CSS, JavaScript). Вони надають зручний інтерфейс для швидкого створення та тестування невеликих фрагментів коду, а також для демонстрації вебефектів та прототипування.

Google Colaboratory – це безплатне хмарне середовище Jupyter Notebook, яке дозволяє писати та виконувати код Python, використовуючи ресурси Google Cloud. Colaboratory особливо популярний у сфері машинного навчання та аналізу даних. Він забезпечує інтеграцію з Google Drive, спільну роботу та потужні обчислювальні ресурси.

Платформа JDoodle підтримує виконання коду багатьма мовами програмування, включаючи Java, C++, Python, C# та інші. JDoodle надає простий інтерфейс для введення та виконання коду, а також API для інтеграції можливостей виконання коду в інші додатки.

Прикладами вебплатформ, що дозволяють виконувати Clojure код, є: 4Clojure [11], Try Clojure [15], MyCompiler.io [16].

Навчальний вебсайт 4Clojure пропонує серію задач з програмування на Clojure. Він має вбудований редактор коду та інтерпретатор, що дозволяє користувачам вводити свої розв’язки та миттєво їх виконувати, отримуючи зворотний зв’язок щодо правильності. Основний акцент зроблено на навчанні через розв’язання задач.

Офіційний вебсайт Try Clojure надає інтерактивну REPL (Read-Eval-Print Loop) для Clojure у браузері. Користувачі можуть вводити вирази Clojure та одразу бачити результати їх виконання. Цей інструмент є чудовим для ознайомлення з базовим синтаксисом та можливостями мови програмування.

MyCompiler.io – це онлайн-компілятор, який підтримує множину мов програмування, серед яких є Clojure. Він надає редактор коду, консоль для виводу та можливість налаштування параметрів виконання. Для Clojure, ймовірно, використовується серверний процес, який отримує код, виконує його за допомогою JVM та повертає результат у браузер.

1.3 Аналіз технологій розробки вебплатформ

Розробка вебплатформ для виконання коду передбачає використання різноманітних технологій як на клієнтській, так і на серверній стороні, а також інструментів для забезпечення безпеки та зручності користувача.

Платформа OnlineGDB підтримує виконання коду багатьма мовами програмування, включаючи C, C++, Python, Java та інші. Для серверної частини використовується, ймовірно, Node.js з фреймворком Express, для забезпечення швидкої обробки запитів та ефективного виконання коду різними мовами. Також може використовуватись PHP для внутрішніх сторінок. Для ізоляції середовищ виконання, ймовірно, використовується Docker, що забезпечує безпеку та відтворюваність. Для клієнтської частини використовується JavaScript, часто з бібліотеками, такими як jQuery та фреймворками, як Bootstrap для створення інтерактивного інтерфейсу та обробки подій.

До переваг можна віднести високу продуктивність та масштабованість серверної частини, універсальність JavaScript на сервері та клієнті, безпека та ізоляція виконання коду, спрощення розробки інтерфейсу та підтримку у різних браузерах.

Можлива складність налаштування для підтримки багатьох мов, можливі проблеми з продуктивністю при інтенсивних обчисленнях на сервері та збільшення розміру клієнтського коду є основними недоліками.

Онлайн середовище розробки JSFiddle спеціалізується на виконанні вебтехнологій: HTML, CSS та JavaScript. Основна частина роботи відбувається на клієнтській стороні. HTML відповідає за структуру документа, CSS – за його візуальне оформлення, а JavaScript – за динамічну поведінку. JSFiddle надає зручний інтерфейс для редагування цих трьох складників та миттєвого перегляду результату у браузері. Для забезпечення деякої функціональності, наприклад, збереження фіделів, може використовуватись серверна частина, написана на будь-якій відповідній мові (наприклад, Node.js, PHP) з використанням бази даних для зберігання інформації.

Його перевагами є: швидкий відгук інтерфейсу завдяки клієнтському виконанню (JavaScript), стандартизованість та широка підтримка вебтехнологій (HTML, CSS).

До недоліків можна віднести обмеженість можливостей виконання коду без серверної підтримки, потенційні проблеми з безпекою виконання JavaScript у браузері.

Навчальна вебплатформа 4Clojure використовує ClojureScript як основну технологію для клієнтської частини. Код Clojure, який користувач вводить у браузері, компілюється в JavaScript за допомогою компілятора shadow-cljs. Для безпечного виконання цього JavaScript використовується бібліотека sci, яка забезпечує виконання підмножини ClojureScript з обмеженими можливостями.

Вебсайт побудований з використанням стандартних вебтехнологій, таких як HTML для структури та CSS для стилізації. Оскільки платформа є статичним вебсайтом, вона не має традиційної серверної частини. Для локальних завдань розробки та збірки використовується Babashka.

Переваги даної вебплатформи – це функціональний підхід до розробки інтерфейсу (ClojureScript), безпечне виконання коду в браузері (sci), ефективна компіляція ClojureScript (shadow-cljs).

Недоліками ж є менша екосистема бібліотек порівняно з JavaScript, обмеженість безпечною підмножиною ClojureScript.

1.4 Основні функціональні вимоги до інтерпретатора

Аналіз аналогічних інструментів дозволяє сформувати уявлення про те, яким має бути функціонал вебінтерпретатора Clojure. Його основна мета – забезпечити користувачеві можливість вводити, запускати та миттєво бачити результати виконання Clojure-коду у веббраузері в інтерактивному режимі.

Першочергово інтерфейс має містити багаторядкову текстову область, яка дозволяє комфортно вводити код. Важливо, щоб ця область була

інтуїтивно зрозумілою, зручною для редагування та адаптованою до потреб як новачків, так і досвідчених користувачів. Після введення коду користувач повинен мати можливість ініціювати його виконання – наприклад, за допомогою кнопки з підписом “Run” або аналогічного елементу.

Результати виконання мають з’являтися у спеціально відведеній області інтерфейсу. Важливо, щоб вивід був зрозуміло структурований і відображав як значення, так і можливі повідомлення про помилки. Слід передбачити коректне форматування для різних типів даних Clojure: чисел, рядків, списків, векторів, мап тощо.

На початковому етапі розробки достатньо підтримки базового синтаксису мови. До цього належать: визначення змінних через def, виклики функцій, літерали основних типів, робота з колекціями, а також використання базових операторів і спеціальних форм, зокрема +, -, *, /, if, let та інші.

Окрема увага має бути приділена обробці помилок. Якщо в коді наявна синтаксична або логічна помилка, інтерпретатор повинен виявляти її та виводити інформативне повідомлення, яке вказує на тип помилки й, за можливості, її розташування у коді.

Також важливо забезпечити зворотний зв’язок під час виконання. Це може бути реалізовано у вигляді індикатора завантаження, анімації або текстового повідомлення про те, що код виконується чи виконання завершено. Така індикація покращує взаємодію користувача з системою та дозволяє уникнути непорозумінь щодо стану процесу.

1.5 Основні нефункціональні вимоги до вебдодатка

Нефункціональні вимоги визначають якісні характеристики вебдодатка інтерпретатора Clojure, які не пов’язані безпосередньо з його функціональністю, але мають суттєве значення для забезпечення ефективної, зручної та надійної роботи системи.

Перш за все, вебдодаток повинен демонструвати належну продуктивність. Це означає, що час відгуку на дії користувача, зокрема при виконанні введеного коду, має бути прийнятним. Прості вирази на Clojure повинні оброблятись швидко – результат варто повертати протягом кількох сотень мілісекунд. При цьому важливо забезпечити ефективне використання серверних ресурсів: як процесора, так і оперативної пам’яті. Це особливо актуально в умовах одночасної роботи кількох користувачів, коли система має залишатися стабільною та швидкодійною.

У питанні безпеки критично важливо передбачити механізми, що унеможливлюють виконання потенційно шкідливого коду, введеного користувачами. Система має бути захищена на рівні виконання – код повинен запускатися ізольовано, без доступу до системних ресурсів сервера. Окрім цього, необхідно подбати про захист від впровадження стороннього скрипту (XSS), який може виникнути внаслідок відображення введеного коду або результатів його виконання. Надійна обробка та фільтрація вмісту допоможуть уникнути таких загроз.

Зручність використання відіграє не менш важливу роль. Інтерфейс вебдодатка має бути інтуїтивно зрозумілим навіть для користувачів без технічної підготовки. Текстове поле введення, кнопка запуску та область виведення результату повинні бути логічно розміщені, легко помітні та доступні. Особливої уваги заслуговує адаптивність: додаток повинен коректно відображатися на різних пристроях, включаючи настільні комп’ютери, планшети та смартфони, з урахуванням особливостей кожного типу екрана.

Також важливо забезпечити сумісність вебдодатка з основними сучасними браузерами – такими як Chrome, Firefox, Safari та Edge – на різних операційних системах. Це гарантує, що користувачі з різними технічними середовищами зможуть безперешкодно користуватися системою.

На завершення, архітектура додатка має бути достатньо гнучкою, щоб у майбутньому підтримувати зростання навантаження. Збільшення кількості одночасних користувачів не повинно призводити до погіршення

продуктивності чи стабільності. Масштабованість повинна бути закладена на ранніх етапах розробки.

Дотримання цих нефункціональних вимог є критично важливим для досягнення високої якості вебдодатка інтерпретатора Clojure, його безпечної та зручної експлуатації, а також подальшого розвитку в міру зростання потреб користувачів.

2 ПРОЄКТУВАННЯ СИСТЕМИ

У цьому розділі розглянуті технічні аспекти розробки інтерпретатора та обрані технології, інструментальні засоби й архітектурні рішення, які є основою створення програмного продукту.

Для обраних технологій наводяться обґрунтування щодо вибору конкретних підходів та бібліотек, виходячи з поставлених завдань та вимог до функціональності, безпеки та зручності використання платформи.

Окрім теоретичного огляду, розділ містить візуальні матеріали у вигляді діаграм, що ілюструють ключові аспекти спроєктованої системи та взаємодію її компонентів.

2.1 Огляд вибору технологій та інструментарію

2.1.1 Основна мова програмування

Основними технологіями для реалізації цього проєкту було обрано мову програмування Clojure [5] та її варіацію – ClojureScript [6]. Такий вибір зумовлений низкою переваг, які ці мови надають для розробки складних інтерактивних вебзастосунків, особливо тих, що пов’язані з обробкою та виконанням коду.

Clojure, як сучасний динамічний Lisp, пропонує філософію та інструментарій, що є особливо цінними для розробки навчальної платформи, орієнтованої на саму Clojure.

Першою перевагою такого вибору є ідіоматична відповідність, оскільки розробка платформи для вивчення Clojure на самій Clojure (та її діалекті ClojureScript) дозволяє глибше зрозуміти та передати філософію мови, її сильні сторони та підходи до розв’язування задач. Це створює гармонійне середовище як для розробників платформи, так і для її майбутніх користувачів.

Наступною перевагою є потужність Lisp для обробки коду. Її використання гарантує, що програма може розглядати та обробляти фрагменти коду (як власного, так і введеного користувачем) так само як вона обробляє звичайні дані, наприклад, списки чисел чи рядків. Такий підхід «код як дані» значно спрощує завдання аналізу, модифікації та виконання коду, що є ключовим для функціоналу інтерпретатору.

Незмінні структури даних та акцент на чистих функціях спрощують управління станом інтерактивних елементів, таких як редактор коду та область виведення результатів. Це веде до більш передбачуваної поведінки та легшого налагодження.

ClojureScript, що використовується для розробки клієнтської частини застосунку, допомагає поєднати всі переваги мови Clojure та перенести їх у скомпільований JavaScript код безпосередньо у браузер користувача. В контексті інтерактивної платформи для виконання Clojure коду це також допомагає використовувати єдину мовну систему для логіки, пов’язаною з Clojure.

Основними перевагами використання цієї мови програмування є водночас використання функціональних принципів LISP синтаксису та відмінну інтеграцію з наявними JavaScript бібліотеками та інструментами, що дозволяє використовувати широкий набір готових рішень, зокрема для створення UI.

2.1.2 Ключові бібліотеки для розробки системи

Для реалізації вебінтерпретатора Clojure використовується набір перевірених інструментів та бібліотек, які забезпечують сучасний, гнучкий і масштабований підхід до розробки. Вибір кожного компонента був продиктований специфікою проєкту, мовою реалізації (Clojure/ClojureScript), а також вимогами до інтерфейсу, продуктивності й інтерактивності.

Розробка інтерфейсу користувача базується на Reagent [18] – легковажній обгортці над React, орієнтованій на екосистему ClojureScript. Reagent дозволяє створювати функціональні компоненти з чистими функціями рендерингу, де стан компонента управляється через атоми – реактивні структури, які автоматично оновлюють інтерфейс при зміні значення. Це значно спрощує керування станом та інтеграцію з іншими частинами програми. Завдяки функціональному стилю програмування, притаманному ClojureScript, Reagent є природним вибором для побудови динамічного та легко масштабованого UI.

Для компіляції та зв’язку між вихідним ClojureScript-кодом і браузером використовується Shadow CLJS [17]. Ця система збірки забезпечує глибоку інтеграцію з NPM, дозволяючи безперешкодно використовувати JavaScript-бібліотеки в ClojureScript-проєкті. Особливо корисними є можливості гарячого перезавантаження (hot reload), що значно пришвидшує цикл розробки, та автоматичне підключення залежностей. Shadow CLJS також має зручну CLI-конфігурацію та підтримує окремі режими роботи з кодом для розробки та релізу.

Для маршрутизації у вебдодатку використовується бібліотека Bidi [19], яка реалізує підхід до маршрутизації як до структури даних. Це дозволяє зручно описувати маршрути у вигляді вкладених векторів і асоціативних мап. Bidi підтримує двосторонню маршрутизацію – як перетворення URL у відповідну функцію, так і побудову URL на основі логіки додатка, що особливо важливо для SPA (single-page application). Такий підхід покращує контроль над логікою навігації та полегшує інтеграцію з Reagent-компонентами.

Ключовим елементом є інтерактивний редактор коду, реалізований на основі CodeMirror 6 [2, 4]. Цей редактор має модульну архітектуру та гнучку систему розширень. Для повноцінної підтримки Clojure інтегруються @nextjournal/lezer-clojure [3] – граматика для парсингу, та nextjournal.clojure-mode – режим з підсвіткою синтаксису та базовою інтеграцією з eval-функціоналом. Разом вони дозволяють реалізувати контекстну підсвітку,

поведінку клавіш (наприклад, Tab/Shift-Tab) та виділення виразів на основі структури коду. CodeMirror 6 також підтримує масштабовану інтеграцію з бібліотекою SCI [20], що відкриває можливості для подальшого розвитку редактора – від виконання коду до аналізу його структури.

Кожна з цих бібліотек виконує чітко визначену роль у загальній архітектурі системи, а їх комбінація забезпечує високий рівень інтерактивності, надійності та продуктивності вебдодатка.

2.1.3 Допоміжні інструменти при розробці та стилізація

Створення сучасного вебдодатка вимагає уваги не лише до логіки та архітектури, а й до зовнішнього вигляду та зручності підтримки коду. У розробці інтерпретатора Clojure було зроблено вибір на користь простоти, прозорості та можливості поступового вдосконалення, що відображається як у підходах до стилізації, так і в супутніх інструментах розробки.

Стилізація інтерфейсу реалізована з використанням чистого CSS, що дозволяє зберегти повний контроль над виглядом компонентів без залучення великих CSS-фреймворків. Основна увага приділяється використанню CSS-змінних (custom properties), які забезпечують гнучке управління кольоровими схемами, відступами, типографікою та іншими параметрами дизайну. Такий підхід полегшує тематизацію інтерфейсу, дозволяє швидко адаптувати стиль до нових вимог, а також сприяє легкій підтримці та повторному використанню стилів у різних частинах додатка.

Для роботи з JavaScript API, зокрема при інтеграції з CodeMirror 6, активно використовується бібліотека applied-science/js-interop [21]. Вона забезпечує зручні макроси для взаємодії з JavaScript-об’єктами без втрати ідіоматичного стилю ClojureScript. Завдяки цьому можлива інтеграція з низькорівневими API редактора, такими як керування курсором, події клавіш, вставка тексту або читання AST-структур, що необхідно для повноцінної реалізації функціоналу редактора коду.

Супутні інструменти розробки відіграють важливу роль у підтримці якості та зручності розробки. Контроль версій здійснюється за допомогою Git та GitHub, що дозволяє ефективно керувати історією змін, працювати з гілками та надійно зберігати написаний код. Як основне середовище для написання та тестування коду обрано Visual Studio Code – легкий, продуктивний редактор з потужною підтримкою ClojureScript через відповідні розширення (наприклад, Calva). VS Code забезпечує швидкий доступ до REPL, перегляд документації, автодоповнення та інтеграцію з Git, що істотно підвищує ефективність роботи над проєктом.

Усі ці рішення спрямовані на збереження балансу між простотою реалізації, розширюваністю системи та комфортом для розробника.

2.2 Структура клієнтської частини та взаємодія з користувачем

Побудова клієнтської частини вебінтерпретатора реалізована на основі компонентного підходу, що надається бібліотекою Reagent, яка є обгорткою над React для ClojureScript. Такий підхід забезпечує модульність, повторне використання коду та зручну організацію логіки інтерфейсу.

Інтерфейс користувача складається з низки ключових елементів, серед яких основну роль відіграють редактор коду, кнопка запуску виконання коду, а також область виводу результатів. Усі ці елементи представлені у вигляді окремих реактивних компонентів, які разом формують ієрархічну структуру застосунку. Вищий рівень структури відповідає за компонування інтерфейсу та управління потоками даних, тоді як нижчі компоненти реалізують конкретну функціональність, наприклад, обробку подій клавіатури чи відображення результатів виконання коду.

Користувацька взаємодія організована через систему подій, що обробляються на різних рівнях компонентів. Події, як-от натискання клавіш чи натискання кнопки, викликають зміни у стані застосунку (state atom), що

призводить до оновлення відображення інтерфейсу. Таким чином, забезпечується безперервний цикл: введення, обробка, вивід.

З метою кращого розуміння архітектури клієнтської частини було побудовано діаграму компонентів, яка демонструє взаємозв’язки між основними складниками застосунку – від головного модуля core.cljs до допоміжних низькорівневих компонентів (див. рис. 2.1).

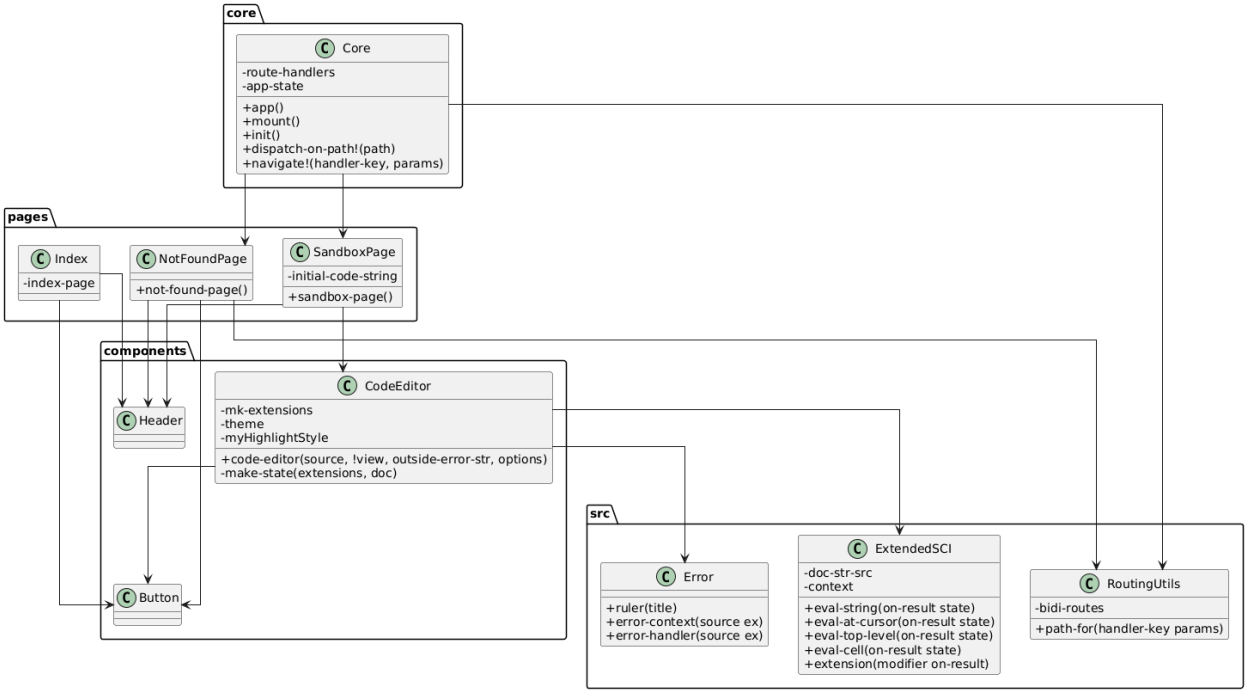


Рисунок 2.1 – Діаграма компонентів

Ця діаграма ілюструє напрямки обміну даними, ієрархію вбудованих компонентів та логіку взаємодії між частинами інтерфейсу, що забезпечують цілісну роботу інтерпретатора на стороні клієнта.

Наступним чином для кращого розуміння взаємодії користувача із клієнтською частиною вебінтерпретатора побудовано діаграму варіантів використання (use case diagram). Ця діаграма відображає основні сценарії взаємодії користувача з інтерфейсом системи, включаючи введення коду, ініціювання його виконання, отримання результату або повідомлення про помилки. Також передбачено використання гарячих клавіш як альтернативного способу запуску виконання. Діаграма дозволяє візуалізувати

ключові функціональні можливості клієнтської частини інтерпретатора та показує, як користувач ініціює відповідні процеси в межах системи (див. рис. 2.2).



Рисунок 2.2 – Діаграма використання

Діаграма послідовностей демонструє ключову логіку роботи компонента code-editor у клієнтській частині вебінтерпретатора (див. рис. 2.3).

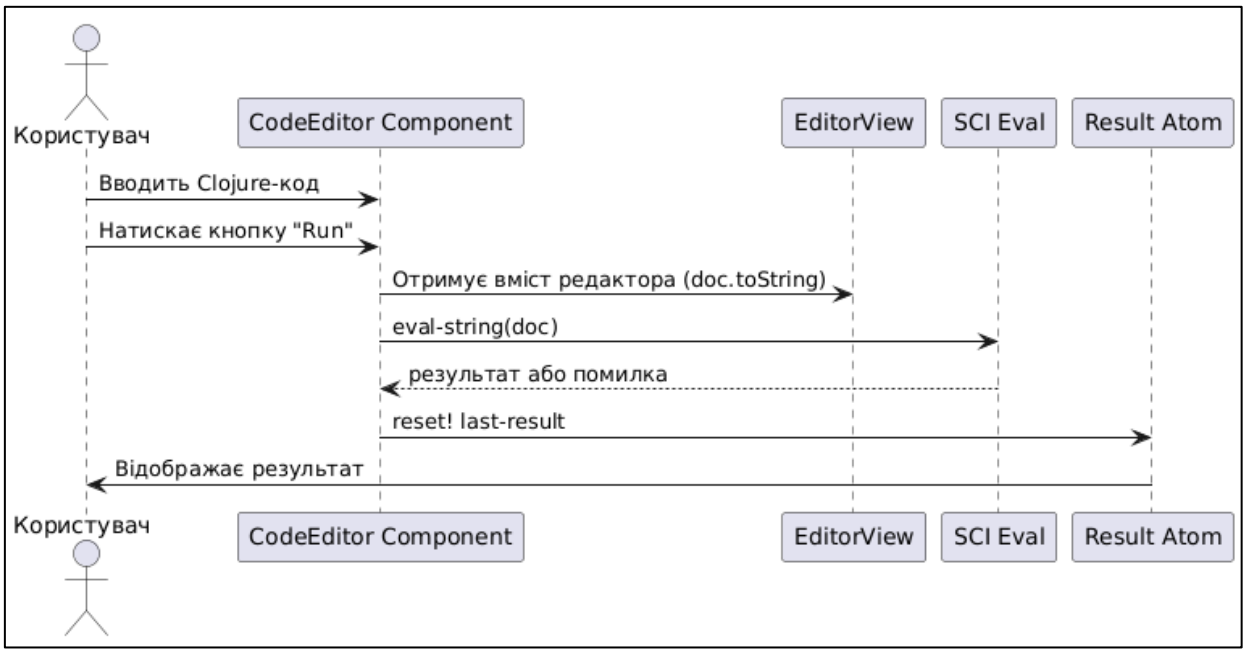


Рисунок 2.3 – Діаграма послідовностей

Процес взаємодії починається з того, що користувач вводить Clojure-код безпосередньо в інтерактивний редактор.

Після завершення введення користувач ініціює виконання коду шляхом натискання кнопки “Run”. У відповідь на цю дію компонент code-editor отримує поточний вміст редактора, викликаючи метод toString на об’єкті EditorView, щоб отримати текстову версію введеного коду.

Цей код передається до вбудованого інтерпретатора SCI, який виконує обчислення. У випадку успішного виконання повертається результат, а в разі виникнення помилки – повідомлення з описом проблеми. Незалежно від результату, отримане значення зберігається в атомі last-result, що відстежує останній вивід.

Після оновлення значення атома інтерфейс автоматично оновлюється, відображаючи результат виконання в окремому полі. Таким чином, забезпечується повний цикл реактивної взаємодії: від введення коду до отримання зворотного зв’язку без перезавантаження сторінки або втрати контексту.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБСТОРІНКИ

3.1 Підготовка середовища розробки

Для початку розробки програми необхідно було визначитись з усіма бібліотеками, що будуть необхідні для розробки подальшого застосунку, та додати їх до проєкту. У контексті даної системи це відбувалось двома способами: додавання необхідної бібліотеки до файлу deps.edn або з використанням команди npm install.

Моя конфігурація у файлі deps.edn зображена на рисунку 3.1.

```
{:paths ["src" "target" "resources"]
 :deps {org.clojure/clojure {:mvn/version "1.12.0"}
        thheller/shadow-cljs {:mvn/version "3.0.5"}
        reagent/reagent {:mvn/version "1.3.0"}
        org.clojure/core.async {:mvn/version "1.8.741"}
        bidi/bidi {:mvn/version "2.1.6"}
        applied-science/js-interop {:mvn/version "0.4.2"}
        org.babashka/sci {:mvn/version "0.9.45"}
        nextjournal/clojure-mode {:git/url "https://github.com/nextjournal/clojure-mode"
                                  :git/sha "37c16b21c5ff6b280492f1447d0d2982c9a06a1a"}}

 :aliases {:shadow-build {:main-opts ["-m" "shadow.cljs.devtools.cli" "release" "app"]}
           :shadow-watch {:main-opts ["-m" "shadow.cljs.devtools.cli" "watch" "app"]}}}
```

Рисунок 3.1 – Файл deps.edn

У даному файлі першим можна побачити визначення шляхів, за якими програма повинна буде шукати основні файли з кодом, стилями тощо. Наступним йде визначення використовуваних бібліотек та їх версії. Основними можна визначити бібліотеки clojure та shadow-cljs, оскільки перша дає можливість загалом використовувати мову Clojure в той час, коли друга

дає можливість переводити цю мову у зрозумілий формат для браузера, тобто JavaScript.

Бібліотека shadow-cljs відповідає за компіляцію програми та надає доступ до інструментів для її легкої розробки та тестування, такі як Hot Reload та вбудований REPL. Для зручності використання цього інструменту були додані aliases. Вони спрощували виконання команд завдяки заведенню більшої команди у невеликий псевдонім. Таким чином для того, щоб запустити компіляцію проєкту та отримувати результат наживо під час розробки, замість команди clj -m shadow.cljs.devtools.cli watch app було достатньо виконати clj -M:shadow-watch. Також, щоб ця команда працювала, було створено файл shadow-cljs.edn та у ньому задано параметри для збирання (див. рис. 3.2).

```
{:source-paths ["src"]

 :builds {:app {:target :browser
                :output-dir "public/js"
                :asset-path "/js"
                :modules {:main {:entries [core]}}
                :devtools {:http-root "public"
                           :http-port 8280}}}}
```

Рисунок 3.2 – Файл shadow-cljs.edn

У цьому файлі вказано, що для тегу app, який вказується у попередній команді, всі скомпільовані файли мають потрапити до теки public/js, після чого сторінка браузера має брати інформацію зі згенерованих JavaScript файлів. Далі у розділі modules було вказано вхідний файл для компіляції, яким в програмі є core.cljs, і ще нижче – за яким портом має відкриватись сторінка.

Наступним чином, для завантаження бібліотек за допомогою засобу npm, тобто менеджера JavaScript пакетів, була виконана команда npm init, після чого до згенерованого файлу package.json додано вміст, зображений на рисунку 3.3.

```
{
  "dependencies": {
    "@codemirror/autocomplete": "6.18.6",
    "@codemirror/commands": "6.8.1",
    "@codemirror/lang-markdown": "6.3.2",
    "@codemirror/language": "6.9.0",
    ....
    "process": "^0.11.10",
    "react": "^19.1.0",
    "react-dom": "^19.1.0",
    "w3c-keyname": "2.2.4"
  }
}
```

Рисунок 3.3 – Файл package.json

Частина бібліотек була підтягнута при встановленні бібліотеки nextjournal/clojure-mode з файлу deps.edn, а деякі з них були підібрані вручну та обрані необхідні версії. Після того, як це все було обрано, була виконана команда npm install, і середовище з усіма необхідними бібліотеками для створення вебінтерпретатора було готове для подальшої розробки.

3.2 Створення основи вебсайту

Для закладання основи вебсайту, на який далі можна буде накладати додатковий функціонал, потрібно 2 основні речі: сторінка index.html та файл з розширенням .cljs для того, щоб він керував доступом до всіх інших файлів у програмі.

Вміст створеного файлу index.html можна побачити на рисунку 3.4. Даний файл служить основною сторінкою для прийняття даних, згенерованих за допомогою ClojureScript та застосування стилів, вказаних у посиланні на stylesheet. Зміст на сайті буде передаватись за допомогою бібліотеки React та його альтернативи у мові Clojure – бібліотека Reagent.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Lecturian</title>
  <link rel="stylesheet" href="/css/index.css">
</head>
<body>
  <div id="root"></div>
  <script src="/js/main.js" type="text/javascript"></script>
  <script>core.init();</script>
</body>
</html>
```

Рисунок 3.4 – Файл index.html

Наступним йде заповнення файлу core.cljs, і починається це з визначення простору імен та підключення необхідних залежностей (див. рис. 3.5).

```
(ns core
  (:require ["react-dom/client" :as rdom-client]
            [reagent.core :as r]
            [bidi.bidi :as bidi]
            [clojure.string :as str]
            [routing-utils :as rout]
            [pages.index :as index-page]
            [pages.not-found :as not-found-page]
            [pages.sandbox-page :as sandbox-page]
            ))
```

Рисунок 3.5 – Визначення залежностей

Першим чином ns core оголошує назву простору, за яким з інших місць у програмі можна звертатись до коду всередині цього файлу.

Підключені залежності для core.cljs наступні:

– Reagent (reagent.core) – обгортка над React для ClojureScript, що дозволяє створювати компоненти у декларативному стилі;
– ReactDOM (react-dom/client) – API React для рендерингу компонентів у DOM;
– Bidi (bidi.bidi) – бібліотека маршрутизації, що дозволяє відповідати URL на логічні імена сторінок;
– routing-utils – власний модуль, у якому зберігаються правила маршрутизації та допоміжні функції;
– pages.* – окремі файли для кожної сторінки, наприклад pages.index, pages.sandbox-page, які містять відповідні компоненти.

Після визначення необхідних залежностей, були вказані всі наявні шляхи, що можуть бути відкриті за посиланням (див. рис. 3.6).

```
(def route-handlers
  {:home {:view #'index-page/index-page}
   :sandbox {:view #'sandbox-page/sandbox-page}
   :not-found {:view #'not-found-page/not-found-page}})
```

Рисунок 3.6 – Визначення імен маршрутів

У даній частині коду сторінкам назначено коротші ключі, щоб мати змогу зручно до них посилатись. Конструкція #' дозволяє передати функцію без її виклику, що зручно для динамічного рендерингу.

Далі, як були визначені маршрути, було створено змінну глобального стану застосунку (див. рис. 3.7).

```
(defonce app-state (r/atom {:current-view #'index-page/index-page
                            :route-params {}
                            :attempted-path "/"}))
```

Рисунок 3.7 – Змінна глобального стану

Для її створення був використаний атом – реактивна структура даних від Reagent. Його особливість є у тому, що у випадку зміни даних всередині цієї змінної, Reagent автоматично визначить які компоненти UI від цього залежать та оновить їх. У цьому випадку визначено, що за шляхом «/» буде відкриватись сторінка index-page, не передаючи до неї жодних параметрів.

Наступним кроком після реалізації можливості змінювати та зберігати глобальний стан було створення функції, що буде обробляти подію зміни шляху у посиланні. Основна частина даної функції зображена на рисунку 3.8.

```
(let [handler-key (:handler matched-route)
        route-params (:route-params matched-route)
        handler-info (get route-handlers handler-key)]
    (if handler-info
      (swap! app-state assoc
          :current-view (:view handler-info)
          :route-params route-params)
      (let [not-found-handler (get route-handlers :not-found)]
        (swap! app-state assoc
            :current-view (:view not-found-handler)
            :route-params {}))))
```

Рисунок 3.8 – Код для обробки зміни шляху посилання

Дана функція приймає шлях, що намагаються відкрити, та робить перевірку: якщо шлях було знайдено серед route-handlers – то оновлюється змінна app-state, передаючи туди об’єкт необхідної сторінки. У випадку, коли шлях не знаходиться, передається об’єкт сторінки not-found-page.

Реалізувавши можливість обробляти подію переходу за різними шляхами у посиланні, була зроблена функція рендерингу сторінки на основі app-state (див. рис. 3.9). Ця функція відповідає за те, щоб відобразити необхідну сторінку на основі введеного шляху, і відображає або отриману сторінку, або повідомлення щодо помилки під час отримання компонента.

```
(defn app []
  (let [state @app-state
        current-view-fn (:current-view state)
        route-params (:route-params state)
        attempted-path (:attempted-path state)
        props (if (= current-view-fn #'not-found-page/not-found-page)
                {:attempted-path attempted-path}
                route-params)]
    (if current-view-fn
      [current-view-fn props]
      [:div "Loading page or component not found..."])))
```

Рисунок 3.9 – Функція рендерингу

Після цього була створена функція global-link-click-handler, завдяки якій застосунок правильно обробляв переходи по внутрішніх посиланнях у рамках SPA (Single Page Application). Основна частина даного коду зображена на рисунку 3.10.

```
(when (and is-internal (not (or is-mailto is-tel is-download is-target-blank has-modifier-key is-middle-mouse)))
        (.preventDefault event)
        (let [path-to-navigate (.-pathname anchor)]
          (.pushState js/history nil "" path-to-navigate)
          (dispatch-on-path! path-to-navigate)))
```

Рисунок 3.10 – Обробка натискання на посилання

Перед тим, як дійти до цього блоку, функція робить 2 речі: знаходить на посилання у HTML коді сторінки, що було натиснуто, та виконує перевірку того, чи це посилання веде на внутрішню сторінку та чи не має воно додаткових атрибутів, з якими посилання має іншу поведінку, наприклад, відкриття посилання у новій вкладці. Якщо всі ці перевірки проходять успішно, то відбувається заборона звичайної дії, щоб сторінка не оновилась повністю при натисканні на посилання, та обробка шляху, що було вказано у

натиснутому посиланні. Тоді програма використовує розроблену раніше функцію dispatch-on-path! для знаходження потрібної сторінки.

Після реалізації всього цього функціоналу залишається дати можливість вебсторінці використовувати всі вищезазначені функції. Для цього першим чином була додана функція mount (див. рис. 3.11).

```
(defn mount []
  (let [container (.getElementById js/document "root")
        root (.createRoot rdom-client container)]
    (.render root (r/as-element [app]))
    (.addEventListener js/window "popstate"
                       (fn [_event]
                         (let [current-path (.-pathname js/window.location)]
                           (dispatch-on-path! current-path))))
    (.addEventListener js/document "click" global-link-click-handler)
    (let [initial-path (.-pathname js/window.location)]
      (dispatch-on-path! initial-path))))
```

Рисунок 3.11 – Функція mount

У даній частині зазначено, що функція має знайти елемент на сторінці index.html з id root, після чого створити React контейнер з даного елементу та згенерувати у ньому поточну відкриту сторінку, що вказується завдяки створеній раніше функції app. Також були додані слухачі подій, а саме: подія popstate, що відповідає за навігацію між сторінками кнопками браузера «назад» / «вперед» та викликає dispatch-on-path! з поточним посиланням, та подія click, для якої додається global-link-click-handler. Наприкінці програма зчитує відкрите посилання та визначає, яка саме сторінка відкрита.

Далі для того, щоб попередню функцію можна було виконати у браузері, була створена експортована функція ініціалізації init (див. рис. 3.12). Завдяки тому, що ця функція є експортованою, до файлу index.html була можливість додати рядок <script>core.init();</script> для її виклику. У ній відбувається підвантаження всієї сторінки як тільки завершується повне завантаження DOM-структури сторінки.

```
(defn ^:export init []
  (if (or (= "complete" (.-readyState js/document))
          (= "interactive" (.-readyState js/document)))
    (mount)
    (js/document.addEventListener "DOMContentLoaded" mount)))
```

Рисунок 3.12 – Експортована функція init

Таким чином було створено достатній фундамент для розробки наповнення інших сторінок та стилів.

3.3 Розробка допоміжних компонентів вебсайту

На наступному етапі розробки було постановлено зробити наповнення для наступних файлів: routing_utils.cljs, header.cljs, button.cljs, index.css. Кожен з цих файлів відіграватиме не головну, але важливу роль для роботи всього вебсайту.

Створено routing_utils.cljs було з основною метою ізоляції логіки маршрутизації у спеціальний модуль, який не залежить від основного застосунку, чим допомагає уникати замкнутого кола залежностей. Початковий код даного модулю зображений на рисунку 3.13.

```
(ns routing-utils
  (:require [bidi.bidi :as bidi]))
(def bidi-routes
  ["/"
   {"" :home
    "sandbox" :sandbox
    true :not-found
    }])
```

Рисунок 3.13 – Оголошення простору імен та змінної

Починається цей файл так само, як і core.cljs – з визначення простору імен та імпортування необхідних бібліотек, з яких для цього блоку була

потрібна лише бібліотека bidi для маршрутизації. Далі йде створення змінної, що описує які шляхи пов’язані із якими ключами. Це дозволяє діставати шляхи у двосторонньому форматі: як за "/sandbox" можна дістати ключ :sandbox, так і за ключем :sandbox можна дістати шлях "/sandbox".

Після цього було додано функцію path-for, що приймає ключ та, опціонально, параметри, і повертає правильний шлях для ключа (див. рис. 3.14).

```
(defn path-for [handler-key & [params]]
  (try
    (let [bidi-params (if (and (seq params) (map? (first params)))
                        (first params)
                        (apply hash-map params))]
      (bidi/path-for bidi-routes handler-key bidi-params))
    (catch js/Error _e
      (str "/error-generating-path-for-" (name handler-key)))))
```

Рисунок 3.14 – Функція path-for

Завдяки незалежності даного файлу від будь-яких інших просторів імен, його використання у файлах, де необхідно визначати текст у посиланні, не викликає циклічної залежності.

Наступним чином було створено header.cljs. Цей файл відповідає за верхню панель сайту з основною навігацією між сторінками. До нього було імпортовано файл routing_utils.cljs, та надано скорочений псевдонім за допомогою коду (:require [routing-utils :as rout]) для спрощеного звернення до функції з цього місця.

Після імпортування даного простору, було створено функцію, що повертає код для верхньої панелі. Блокова структура, що використовується у коді для будування елементів, потім перетворюється на HTML код у браузері завдяки тому, що Reagent використовує у своїй основі бібліотеку hiccup, яка і додає підтримку подібного синтаксису (див. рис. 3.15).

```
(defn header []
  [:header {:class "page-header"}
   [:div {:class "page-header-container container"}
    [:a {:href (rout/path-for :home)
         :class "brand-link"}
     "Lecturian"]
    [:nav {:class "main-nav"}
     [:ul {:class "nav-list"}
      [:li {:class "nav-item"}
       [:a {:href (rout/path-for :sandbox)
            :class "nav-link"}
        "Sandbox"]]
      ]]]])
```

Рисунок 3.15 – Функція header

У вказаній частині коду відображені не всі елементи списку, які містить в собі верхня панель, але зображений загальний вид побудови HTML коду за допомогою бібліотеки hiccup та застосовані класи стилів і посилання до інших сторінок.

Після цього був створений файл button.cljs для зручного та швидкого створення компонента button та накладання на нього відповідних стилів. На початку файлу, після оголошення назви простору імен та імпортування необхідних бібліотек, було додано дві змінні, що містили в собі класи, пов’язані із ключами для доступу до них (див. рис. 3.16).

```
(def button-variant-classes
  {:default "button-default"
   :destructive "button-destructive"
   :outline "button-outline"
   :secondary "button-secondary"
   :ghost "button-ghost"
   :link "button-link"})
(def button-size-classes
  {:default "button-size-default"
   :sm "button-size-sm"
   :lg "button-size-lg"
   :icon "button-size-icon"})
```

Рисунок 3.16 – Змінні зі стилями

Коли можливі стилі були визначені, була створена функція, що приймала властивості кнопки та інші вкладені елементи (див. рис. 3.17).

```
(defn button [props & children-args]
  (let [{:keys [variant size class]
         :or   {variant :default size :default class ""}} props
        html-props (dissoc props :variant :size :class)
        base-class "button"
        variant-class (get button-variant-classes variant (get button-variant-classes :default))
        size-class (get button-size-classes size (get button-size-classes :default))
        all-generated-classes (str/join " " (remove empty? [base-class variant-class size-class]))
        final-classes (str/join " " (remove empty? [all-generated-classes class]))]
    (into [:button (merge {:className final-classes
                           :type (or (:type html-props) "button")}
                          html-props)]
          children-args)))
```

Рисунок 3.17 – Функція button

За допомогою цієї функції в інших частинах коду є можливість викликати її, передати необхідні параметри стилю, розміру та додаткових класів, і отримати готовий HTML компонент кнопки із застосованими стилями.

Останнім для частини допоміжних компонентів був доданий файл index.css. У ньому вказувались всі кольори, що використовувались для стилізації сайту, і в процесі розробки додавались необхідні класи для покращення зовнішнього вигляду як окремих компонентів, так і сторінок в цілому.

3.4 Створення основних сторінок вебсайту

На розробленому вебсайті на цей момент присутні три сторінки: головна сторінка (вона ж index), сторінка пісочниці для редагування та виконання коду та сторінка, що відображається при невірному посиланні, тобто not found page.

Головна сторінка була написана в першу чергу за допомогою синтаксису hiccup та з використанням створених раніше допоміжних компонентів: header та button. Поєднання цих компонентів із розробленими класами стилів під кожен використовуваний елемент дозволили мені отримати сучасний, в міру мінімалістичний стиль, використовуючи за основу темні відтінки для комфортного користування сайтом (див. рис. 3.18).

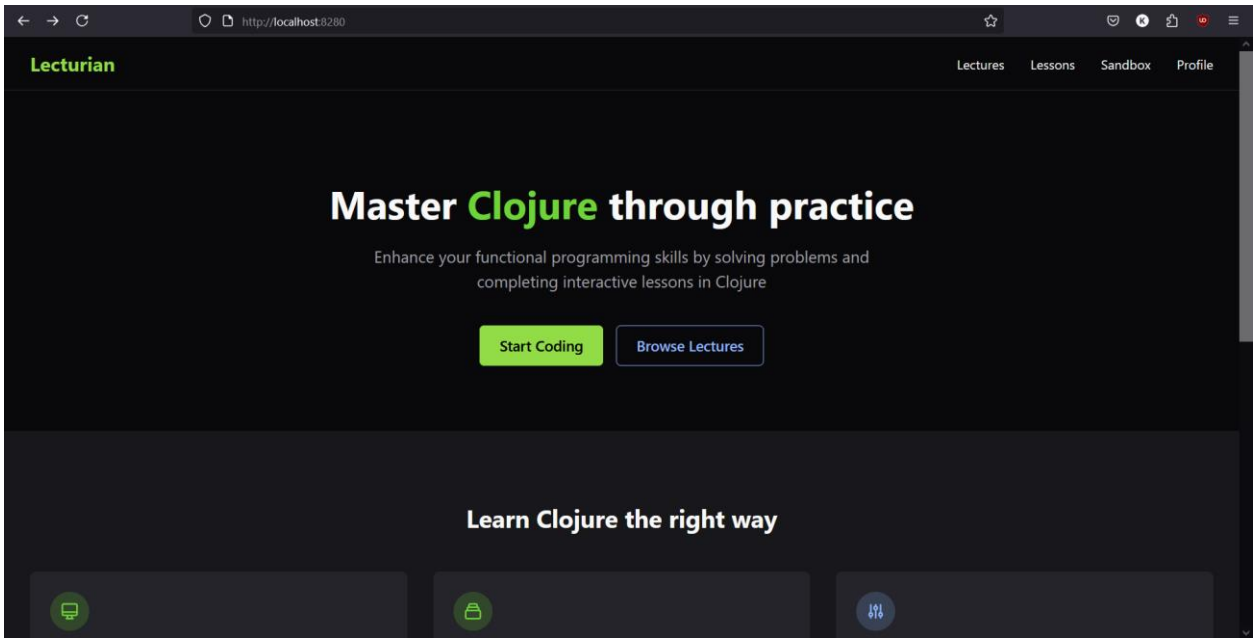


Рисунок 3.18 – Головна сторінка вебсайту

На цьому зображені видно і використання верхньої панелі з назвою сайту, і створення кнопок, що у коді мають використання, зображене на рисунку 3.19.

```
[:a {:href "/lessons"}
    [button/button {:class "button-index-large button-index-primary-hero"}
     "Start Coding"]]
```

Рисунок 3.19 – Створення кнопки з компонентом button

Після реалізації головної сторінки була зроблена сторінка not found для її відображення при спробі користувача перейти за маршрутом, що не існує. Файл, що генерує цю сторінку, містить в собі функцію, яка приймає шлях,

який користувач пробував відкрити, та виводить у консоль повідомлення, що була спроба відкрити шлях, що не існує. Окрім цього, повертається невеликий hiccup код для відображення помилки користувачеві (див. рис. 3.20).

```
[:div {:class "page-wrapper"}
      [header/header]
      [:div {:class "not-found-page-content-area"}
       [:h1 {:class "not-found-title"} "404"]
       [:p {:class "not-found-message"} "Oops! Page not found"]
       [:a {:href (rout/path-for :home)}
        [button/button {:class "button-index-large button-index-cta"}
         "Return to Home"]]]]
```

Рисунок 3.20 – Код для сторінки 404

В результаті виконання цього коду виходить сторінка, що зображена на рисунку 3.21.

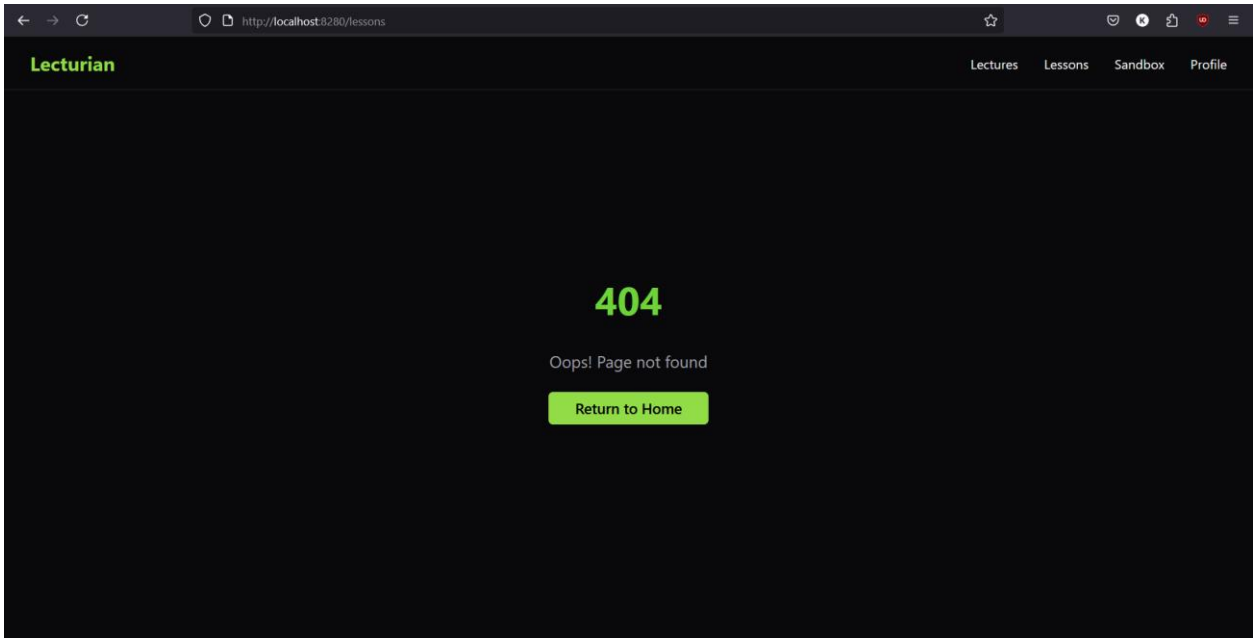


Рисунок 3.21 – Сторінка 404

Останньою з цього списку є сторінка пісочниці, в якій знаходяться редактор коду та є можливість виконання цього коду і відображення результатів. У файлі sandbox_page.cljs використовується для відображення

усього вищеперерахованого функціоналу компонент code-editor, детальна розробка якого йде далі. Також дана сторінка служить збереженням станів написаного коду та компонентів, що відображаються (див. рис. 3.22).

```
(defn sandbox-page []
  (let [code (r/atom initial-code-string)
        view (r/atom nil)]
    (fn []
      [:div {:class "page-wrapper"}
       [header/header]
       [:main
        [:div {:class "editor-wrapper"}
         [code-editor/code-editor @code view]]]])))
```

Рисунок 3.22 – Функція sandbox-page

У цій частині для відображення поля редагування коду використовуються лише додаткові класи та змінна initial-code-string, що містить в собі код, який буде відображений на початку при завантаженні сторінки.

3.5 Розробка редактору коду

Після розробки всіх сторінок вебсайту та налаштування маршрутизації був етап створення функціоналу самого редактора коду. Почалась розробка редактору як і попередні файли з розширенням .cljs – визначенням простору імен та імпортуванням всіх необхідних бібліотек для його роботи (див. рис. 3.23).

Нові бібліотеки, що не повторювались у попередніх файлах, наступні:

– extended-sci містить в собі логіку для виконання коду, що буде описана далі;
– @codemirror/* та nextjournal.cloojure-mode використовуються для побудови редактора, його стилізації та обробки подій;

– @lezer/highlight підключено для реалізації підсвітки коду;
– applied-science.js-intertop потрібен для легшої роботи з певними JavaScript модулями.

```
(ns components.code-editor
  (:require
   [extended-sci :as sci]
   [reagent.core :as r]
   [components.ui.button :as button]
   ["@codemirror/language" :as language :refer [syntaxHighlighting foldGutter
HighlightStyle]]
   ....
   ["@lezer/highlight" :refer [tags]]
   [applied-science.js-interop :as j]
   [nextjournal.clojure-mode :as cm-clj]
   ...))
```

Рисунок 3.23 – Імпортування бібліотек для редактора

Далі йшло створення змінної myHighlightStyle, в якій було створено об’єкт, що містив в собі налаштування щодо того, які стилі повинні мати різні частини коду (див. рис. 3.24).

```
(def myHighlightStyle
  (.define HighlightStyle
          (j/lit [
             {:tag tags.keyword, :color "hsl(var(--secondary))"}
             {:tag tags.comment, :color "hsl(var(--muted-foreground))", :fontStyle "italic"}
             {:tag tags.string, :color "hsl(var(--primary))"}
           ...]))))
```

Рисунок 3.24 – Визначення стилів користувача

Тут можна побачити загальну структуру відношення стилів до коментарів, ключових слів та інших сутностей.

Після вказання стилів для підсвітки була зроблена змінна theme, що містить в собі об’єкт із вказанням загальних стилів редактора, такі як: стилі заднього фону, поля вибору, цифр рядків, курсору і т.д. (див. рис. 3.25).

```
(def theme
  (.theme
   EditorView
   (j/lit {"&" {:background-color "hsl(var(--background))"
                :max-height "68vh"}
           "& ::selection" {:color "inherit !important"}
           ".cm-content" {:white-space "pre-wrap"
                          :padding "10px 0"
                          :background-color "var(--code-background-default)"
                          :color "hsl(var(--foreground))"}
            ...})))
```

Рисунок 3.25 – Налаштування стилів редактора

Наступним кроком було складання усіх створених мною змінних стилів та вбудованого функціоналу бібліотек в одну змінну з усіма розширеннями: mk-extensions. Туди були додані усі розширення щодо стилів, функціоналу клавіш, підсвітки коду та роботи цифр рядків.

Перед тим, як писати код для компонентів у самому редакторі, була створена функція make-state, що приймала список розширень та введений код і створювала об’єкт редактора (див. рис. 3.26).

```
(defn- make-state [extensions doc]
  (.create EditorState
           #js{:doc doc
               :extensions extensions}))
```

Рисунок 3.26 – Функція make-state

Наступним у цьому файлі йде функція code-editor, що складається з двох частин: створення локальних змінних, що зберігатимуть стан редактору,

результат останнього виконання коду для відображення у полі результатів та визначення компонентів, які буде мати даний редактор, що містить в собі кнопку для виконання коду, поле для його написання та поле для відображення результатів.

Після вказання всіх необхідних внутрішніх змінних йде визначення всіх компонентів, що повинен мати редактор (див. рис. 3.27).

```
[button/button
    {:variant :secondary
     :class "editor-run-button"
     :on-click
     (fn []
       (when-let [view @!view]
         (let [doc (.. view -state -doc toString)
               result (to-readable-output (sci/eval-string doc))]
           (reset! last-result result))))}
    "Run"]
   [:div {:ref mount!}]
   [:div {:class "editor-result-field"}
    (or @last-result "Result will appear here...")]]
```

Рисунок 3.27 – HTML компоненти редактора

3.6 Створення функціоналу виконання коду

З готовою областю для написання коду залишалось лише додати функціонал його виконання та правильного відображення результатів. Розробка цього функціоналу була розбита на дві частини: створення файлу для повернення детального контексту помилки коду та файлу, що дозволятиме використовувати клавіші для виконання коду. За основу для даного функціоналу було взято частину редактору з сайту 4clojure 4ever [1].

Файл error.cljs містить в собі обробку повідомлення помилки та генерацію розширеного повідомлення, що містить в собі: назву помилки, її тип, місце у коді, фазу виконання коду, під час якої вона виникла, та контекст,

де детально вказано місце, де виникла помилка. Основна частина цього коду, що відображає синтаксис створення повідомлення, зображена на рисунку 3.28.

```
(when-let [name (.-name ex)]
    (when-not (= "Error" name)
      (println "Type:    " name)))
  (when-let [m (.-message ex)]
    (println (str "Message:  " m)))
  (let [{:keys [:file :line :column]} d]
    (when line
      (println (str "Location: "
                    (when file (str file ":"))
                    line ":" column ""))))
```

Рисунок 3.28 – Створення повідомлення помилки

Окрім цього у файлі існують допоміжні функції, такі як ruler, що створює лінію для кращого візуального розділення між розділами, та error-context, що генерує повідомлення для відображення конкретного місця у коді, де була помилка.

Після цього був створений файл із назвою extended-sci.cljs, оскільки у ньому описувався додатковий функціонал для виконання коду за допомогою бібліотеки sci. У ньому є дві основні функції, що відповідають за виконання коду, перша з яких – eval-string (див. рис. 3.29).

```
(defn eval-string [source]
  (try {::result (sci/eval-string* context source)}
       (catch :default e
         {::error-str (with-out-str (error-handler source e))})))
```

Рисунок 3.29 – Функція eval-string

Дана функція використовується для того, щоб виконати код і повернути його результати при успішному виконанні, у випадку помилки – викликати функцію error-handler з попередньо створеного файлу та повернути детальний текст помилки.

Другою функцією є extension, яка приймає назву додаткової клавіші, що потрібно натиснути для виконання коду, та об’єкт, у який буде записаний результат (див. рис. 3.30).

```
(defn extension [{:keys [modifier
                         on-result]}]
  (.of view/keymap
       (j/lit
        [{:key "Mod-Enter"
          :run (partial eval-cell on-result)}
         {:key (str modifier "-Enter")
          :shift (partial eval-top-level on-result)
          :run (partial eval-at-cursor on-result)}])))
```

Рисунок 3.30 – Функція extension

Після прийняття необхідних параметрів повертається додаткове розширення, в якому міститься обробник події для натискання відповідних клавіш.

Наступним кроком залишалось під’єднати дану функцію до основного коду редактору. На рисунку 3.27 було зображено, як це реалізовано для кнопки Run та, окрім цього, збільшений функціонал розширень був доданий для code-editor (див. рис. 3.31).

```
(make-state
                                (clj->js
                                 (concat
                                  mk-extensions
                                  #js [(sci/extension
                                     {:modifier "Alt",
                                      :on-result
                                      (fn [result]
                                        (reset! last-result
                                                (to-readable-output result)))})])) source)
```

Рисунок 3.31 – Створення об’єкта редактора

У даній частині коду видно, що до визначених заздалегідь розширень додається функціонал виконання коду з допоміжною клавішею “Alt”. Можна

також побачити, що в місцях з виконанням коду використовується функція to-readable-output, що використовується для того, щоб правильно обробляти повернення результату чи помилки від блоку extended-sci.cljs (див. рис. 3.32).

```
(defn to-readable-output [result]
  (or (::sci/error-str result)
      (::sci/result result)))
```

Рисунок 3.32 – Функція to-readable-output

На цьому була завершена розробка редактору коду та сторінки Sandbox в цілому, і результатом був редактор коду, що підтримує різні функції, як: декілька курсорів, підсвітка синтаксису, звертання блоків коду, підсвітка відповідних та невідповідних дужок, автоматична табуляція та використання різних комбінацій клавіш для зручнішого виконання коду. Окрім цього, готові оброблювачі подій, що виконують код з усім базовим синтаксисом Clojure та відображають результат у відведеному для цього полі виводу (див. рис. 3.33).

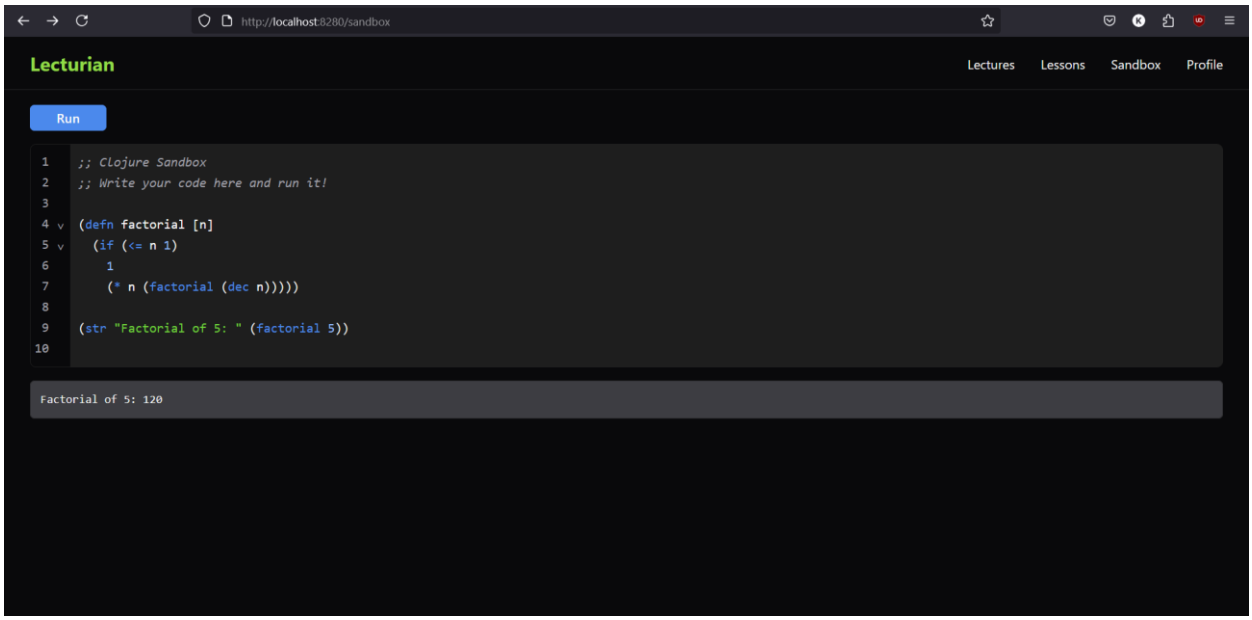


Рисунок 3.33 – Сторінка Sandbox

Таким чином було описано повний процес розробки вебсторінки для інтерпретатора Clojure з використанням самої мови програмування Clojure.

ВИСНОВКИ

У ході виконання роботи було реалізовано повноцінну вебплатформу, яка дозволяє користувачам взаємодіяти з інтерпретатором мови програмування Clojure в інтерактивному режимі без необхідності встановлення додаткового програмного забезпечення. Робота охопила всі ключові етапи розробки: від формування технічного завдання до побудови архітектури та безпосередньої реалізації функціонального вебінтерфейсу.

У процесі проєктування було визначено основні вимоги до функціональності платформи: підтримка підсвітки синтаксису, можливість виконання коду, а також зручність використання для початківців. Було обрано сучасні інструменти й бібліотеки: CodeMirror 6 для реалізації редактора коду, SCI (Small Clojure Interpreter) для інтерпретації виразів, а також ClojureScript і Reagent для побудови реактивного інтерфейсу.

Результатом роботи стала функціональна вебсторінка, яка дозволяє користувачам писати, запускати та аналізувати Clojure код у реальному часі. Платформа може слугувати навчальним інструментом для самостійного вивчення мови, проведення інтерактивних занять або демонстрації концепцій функціонального програмування.

Розроблений прототип відкриває можливості для подальшого розвитку, зокрема – логування користувачів, додавання можливості створювати задачі, інтеграції з наявними базами завдань та збереження стану виконання коду.

ПЕРЕЛІК ПОСИЛАНЬ

1. Oxalorg 4ever-clojure – сторінка на GitHub. URL: https://github.com/oxalorg/4ever-clojure (дата звернення: 20.04.2025).

2. Nextjournal Clojure-mode – сторінка на GitHub. URL: https://github.com/nextjournal/clojure-mode (дата звернення: 21.04.2025).

3. Nextjournal Lezer-clojure – сторінка на GitHub. URL: https://github.com/nextjournal/lezer-clojure (дата звернення: 25.04.2025).

4. Codemirror. URL: https://codemirror.net/ (дата звернення: 23.04.2025).

5. Hickey R. The Clojure Programming Language. *Clojure*. URL: https://clojure.org/ (дата звернення: 17.04.2025).

6. ClojureScript. URL: https://clojurescript.org/ (дата звернення: 19.04.2025).

7. Mika A. Interactive Website Essentials: Features, Examples and Tips. URL: https://www.ramotion.com/blog/interactive-website/ (дата звернення: 19.03.2025).

8. Understanding Interpreter Architecture: A Comprehensive Guide. *Diverse Daily*. URL: https://diversedaily.com/understanding-interpreter-architecture-a-comprehensive-guide/ (дата звернення: 20.03.2025).

9. OnlineGDB. URL: https://www.onlinegdb.com/ (дата звернення: 06.04.2025).

10. JSFiddle. URL: https://jsfiddle.net/ (дата звернення: 06.04.2025).

11. 4ever-clojure. URL: https://4clojure.oxal.org/ (дата звернення: 07.04.2025).

12. CodePen. URL: https://codepen.io/ (дата звернення: 06.04.2025).

13. Google Collaboratory. URL: https://colab.research.google.com/ (дата звернення: 07.04.2025).

14. JDoodle. URL: https://www.jdoodle.com/ (дата звернення: 08.04.2025).

15. Scotto E. TryClojure. URL: https://tryclojure.org/ (дата звернення: 08.04.2025).

16. MyCompiler. URL: https://www.mycompiler.io/ (дата звернення: 09.04.2025).

17. Heller T., Kay T. Shadow CLJS User’s Guide. URL: https://shadow-cljs.github.io/docs/UsersGuide.html (дата звернення: 28.04.2025).

18. Reagent. URL: https://reagent-project.github.io/ (дата звернення: 29.04.2025).

19. Juxt Bidi – сторінка на GitHub. URL: https://github.com/juxt/bidi (дата звернення: 29.04.2025).

20. Babashka SCI – сторінка на GitHub. URL: https://github.com/babashka/sci (дата звернення: 30.04.2025).

21. Applied Science Studio JS-intertop – сторінка на GitHub. URL: https://github.com/applied-science/js-interop (дата звернення: 01.05.2025).