Полтавський університет економіки і торгівлі
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп’ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
_____Олена ОЛЬХОВСЬКА
(підпис)
«___»_____202_ р.

КВАЛІФІКАЦІЙНА РОБОТА
на тему

«РОЗРОБКА НАВЧАЛЬНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ТЕМИ «ОБРОБКА ВИНЯТКОВИХ СИТУАЦІЙ В JAVA» ДИСЦИПЛІНИ «ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ»

зі спеціальності 122 Комп’ютерні науки
освітня програма «Комп’ютерні науки»
ступеня бакалавр

Виконавець роботи Олешко Анна Русланівна
_____«____» _____202_ р.
(підпис)

Науковий керівник к.ф.-м.н. Олексійчук Ю. Ф.
_____«____»_____202_ р.
(підпис)

Рецензент

ПОЛТАВА 2026

РЕФЕРАТ

Записка: 93 с., 17 рис., 3 таблиці, 1 додаток, 18 джерел.

НАВЧАЛЬНА ПЛАТФОРМА, JAVA, ВИНЯТКИ, EXCEPTIONS, NEXT.JS, MONACO EDITOR, PISTON API, ВЕБ-ДОДАТОК, ІНТЕРАКТИВНЕ НАВЧАННЯ, BANKING SYSTEM

Об'єктом розробки є програмне забезпечення — інтерактивна веб-платформа для навчання обробки виняткових ситуацій у мові програмування Java у межах дисципліни «Об'єктно-орієнтоване програмування».

Предметом розробки є програмна реалізація клієнт-серверного веб-додатку з використанням фреймворку Next.js, бібліотеки React, мови TypeScript, редактора Monaco та сервісу віддаленого виконання коду Piston.

Метою роботи є створення зручного у використанні навчального програмного забезпечення, що поєднує теоретичний матеріал, практичні вправи з можливістю запуску Java-коду у браузері та тестування знань для опанування механізмів обробки виняткових ситуацій.

Результатом роботи стало розроблення навчальної платформи «JavaExceptions» на базі Next.js 16 та React 19. Реалізовано ключові модулі:

- модуль теорії — одинадцять послідовних навчальних модулів зі структурованим текстом, блоками коду, примітками, попередженнями та діаграмою ієрархії винятків, у тому числі модулі продакшн-практик та підсумковий проєкт спрощеної банківської системи;
- модуль практичних вправ — редактор Monaco з підтримкою Java, чотири типи вправ (доповнення коду, пошук помилки, передбачення виводу, написання з нуля) та автоматична перевірка результату;
- модуль виконання коду — серверний маршрут /api/run, що інтегрується з API Piston та запускає Java 15.0.2 з підтримкою введення зі stdin;
- модуль тестування — інтерактивні квізи з 8 питаннями множинного вибору, поясненням правильної відповіді та обчисленням відсотка успіху;
- модуль фінального екзамену — підсумкове оцінювання за всіма темами курсу зі збереженням найкращого результату;

- модуль прогресу — сторінка статистики, кільця прогресу, облік спроб і типів помилок з персистентним зберіганням у localStorage через Zustand;
- ігровий майданчик та глосарій — вільне середовище для написання Java-коду й довідник термінів з прив'язкою до модулів.

Особливості: повністю клієнт-серверна архітектура без зовнішньої бази даних, адаптивний дизайн на Tailwind CSS, підтримка темної та світлої теми, безкоштовне виконання коду через публічне API Piston, типобезпечність завдяки TypeScript, серверний рендеринг (App Router) для покращення продуктивності.

Проведено функціональне, інтеграційне та юзабіліті-тестування: перевірено коректність виконання Java-коду через Piston, валідацію результатів вправ, збереження прогресу між сесіями, навігацію між модулями та адаптивність інтерфейсу на різних розмірах екрану.

Розроблений програмний продукт може бути використаний як допоміжний інструмент під час вивчення дисципліни «Об'єктно-орієнтоване програмування» у закладах вищої освіти, для самостійної підготовки студентів та як основа для подальшого розширення на інші теми мови Java.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ, ТЕРМІНІВ

Умовні позначення, символи, скорочення, терміни	Пояснення умовних позначень, скорочень, символів
API	Інтерфейс прикладного програмування (Application Programming Interface)
CSS	Каскадні таблиці стилів (Cascading Style Sheets)
HTML	Мова гіпертекстової розмітки (HyperText Markup Language)
HTTP	Протокол передавання гіпертексту (HyperText Transfer Protocol)
IDE	Інтегроване середовище розробки (Integrated Development Environment)
JS	Мова програмування JavaScript
JSON	Текстовий формат обміну даними (JavaScript Object Notation)
JSX	Розширення синтаксису JavaScript для React (JavaScript XML)
JVM	Віртуальна машина Java (Java Virtual Machine)
ООП	Об'єктно-орієнтоване програмування
ПЗ	Програмне забезпечення
REST	Архітектурний стиль взаємодії (REpresentational State Transfer)
SPA	Односторінковий веб-додаток (Single Page Application)
SSR	Серверний рендеринг (Server-Side Rendering)
TS	Мова програмування TypeScript
UI	Інтерфейс користувача (User Interface)
URL	Уніфікований локатор ресурсу (Uniform

	Resource Locator)
UX	Взаємодія користувача з продуктом (User Experience)

ВСТУП

Сучасне програмне забезпечення характеризується високою складністю, великою кількістю зовнішніх інтеграцій та стрімким зростанням обсягу даних, які обробляються у режимі реального часу. У таких умовах надзвичайної важливості набуває здатність програми коректно та передбачувано реагувати на нештатні ситуації — некоректні вхідні дані, недоступність зовнішніх ресурсів, втрату з'єднання або перевищення допустимих обмежень. Якісна обробка виняткових ситуацій є одним з ключових критеріїв надійності, безпечності та зручності супроводу прикладного програмного забезпечення.

Мова програмування Java займає одну з провідних позицій у світовому рейтингу мов і широко застосовується для розробки корпоративних, мобільних, веб-та хмарних систем. Механізм винятків Java є невід'ємною частиною мови і базується на розгалуженій ієрархії класів Throwable, поділі винятків на перевірювані компілятором (checked) та неперевірювані (unchecked), а також на конструкціях try-catch-finally, throw, throws, multi-catch, try-with-resources та механізмі ланцюжків винятків. Грамотне використання цих конструкцій безпосередньо впливає на якість програмного коду, тому навчанню обробки виняткових ситуацій приділяється значна увага у курсі дисципліни «Об'єктно-орієнтоване програмування».

Традиційне навчання обробки винятків переважно зводиться до конспектування лекційного матеріалу та виконання лабораторних робіт у локальному середовищі розробки. Такий підхід має низку недоліків: відсутність миттєвого зворотного зв'язку, потреба у попередньому встановленні JDK та інтегрованого середовища розробки, обмеженість контрольних матеріалів та неможливість для викладача відстежувати реальний прогрес здобувачів освіти. Сучасні тенденції розвитку освітніх технологій вимагають появи інтерактивних веб-платформ, які дозволяють поєднувати теоретичний матеріал, практичні завдання з можливістю запуску коду у браузері та автоматизоване тестування знань.

Актуальність кваліфікаційної роботи зумовлена потребою у доступному, безкоштовному та зручному навчальному програмному забезпеченні з обробки виняткових ситуацій у Java, яке не вимагає від користувача встановлення жодних інструментів і дозволяє послідовно опрацьовувати теорію, відпрацьовувати практичні навички та перевіряти знання у єдиному веб-середовищі.

Метою кваліфікаційної роботи є створення інтерактивної веб-платформи для навчання обробки виняткових ситуацій у мові програмування Java, яка поєднує структурований теоретичний матеріал, виконання Java-коду у браузері та автоматизовану перевірку знань.

Для досягнення поставленої мети було визначено такі завдання:

- проаналізувати предметну область та дослідити сучасні підходи до інтерактивного навчання програмування;
- виконати огляд існуючих освітніх платформ-аналогів і обґрунтувати доцільність розробки власного програмного продукту;
- розглянути теоретичні засади механізму обробки виняткових ситуацій у Java та обрати технологічний стек для реалізації веб-додатку;
- спроєктувати архітектуру навчальної платформи та розробити блок-схему алгоритму її роботи;
- підготувати структурований навчальний матеріал, що охоплює усі основні теми обробки винятків у Java;
- реалізувати клієнтську частину веб-додатку з підтримкою редактора коду, інтерактивних вправ і тестів;
- реалізувати серверну частину з інтеграцією сервісу віддаленого виконання Java-коду;
- реалізувати систему збереження прогресу користувача та фінального екзамену;
- провести тестування розробленого програмного забезпечення та підготувати інструкцію для користувача.

Об'єктом дослідження є процес інтерактивного навчання обробки виняткових ситуацій у мові програмування Java у межах дисципліни «Об'єктно-орієнтоване програмування».

Предметом дослідження є програмні засоби та технології розробки навчальних веб-додатків з підтримкою віддаленого виконання Java-коду, автоматизованої перевірки результатів та збереження прогресу користувача.

Практичне значення одержаних результатів полягає у тому, що розроблене ПЗ може бути використане під час викладання дисципліни «Об'єктно-орієнтоване програмування» у закладах вищої освіти, для самостійної підготовки студентів спеціальностей у галузі інформаційних технологій та як основа для розширення курсу на інші теми мови Java.

Кваліфікаційна робота складається зі вступу, постановки задачі, трьох розділів, висновків, списку інформаційних джерел та одного додатку. У першому розділі виконано аналіз предметної області та порівняльний огляд аналогів. Другий розділ містить теоретичне обґрунтування технологічного стеку та архітектурних рішень. Третій розділ присвячений практичній реалізації навчальної платформи, тестуванню та інструкції для користувача.

ПОСТАНОВКА ЗАДАЧІ

Основною задачею кваліфікаційної роботи є розробка інтерактивної навчальної веб-платформи з обробки виняткових ситуацій у мові програмування Java. Програмний продукт має забезпечувати єдине середовище для опрацювання теоретичного матеріалу, виконання практичних вправ із можливістю запуску Java-коду безпосередньо у браузері, проходження тестів та фінального екзамену, а також для автоматизованого збереження прогресу користувача без використання зовнішньої бази даних.

Розроблений програмний засіб призначений для здобувачів вищої освіти комп'ютерних спеціальностей, які вивчають дисципліну «Об'єктно-орієнтоване програмування», а також може використовуватися викладачами як допоміжний інструмент під час проведення лекційних та лабораторних занять.

Для досягнення поставленої задачі необхідно виконати такі підзадачі:

1. проаналізувати сучасний стан розвитку інтерактивних освітніх платформ і визначити вимоги до функціональності навчального ПЗ;
2. провести огляд та порівняльний аналіз існуючих рішень-аналогів для виявлення їхніх переваг і недоліків;
3. обґрунтувати вибір технологічного стеку для реалізації веб-додатку, зокрема фреймворку, мови програмування, бібліотеки інтерфейсу та засобу віддаленого виконання Java-коду;
4. розглянути теоретичні засади обробки виняткових ситуацій у Java та сформувати структуру навчального курсу з одинадцяти тематичних модулів;
5. спроєктувати архітектуру програмного продукту та розробити блок-схему алгоритму його роботи;
6. реалізувати клієнтську частину веб-додатку з адаптивним інтерфейсом, підтримкою світлої та темної теми;
7. інтегрувати редактор коду Monaco з підсвічуванням синтаксису Java та сервіс віддаленого виконання Piston для запуску коду користувача;

8. реалізувати модуль практичних вправ з чотирма типами завдань та автоматичною перевіркою результатів виконання;
9. реалізувати модуль тестування з підтримкою питань множинного вибору, поясненнями відповідей та обчисленням відсотка успіху;
10. реалізувати модуль фінального екзамену для підсумкової перевірки знань;
11. реалізувати систему збереження прогресу користувача з персистентним зберіганням у локальному сховищі браузера;
12. провести функціональне, інтеграційне та юзабіліті-тестування розробленої платформи;
13. підготувати інструкцію для користувача та оформити супровідну документацію.

До розробленого програмного забезпечення висуваються такі функціональні вимоги: наявність структурованого теоретичного матеріалу з підтримкою блоків коду, приміток та діаграм; редактор коду з підсвічуванням синтаксису Java; можливість запуску коду з підтримкою введення зі стандартного потоку; автоматичне порівняння виводу програми користувача з очікуваним результатом; ведення статистики спроб та помилок; інтерактивні тести з негайним зворотним зв'язком; збереження прогресу між сесіями.

Нефункціональні вимоги охоплюють: адаптивність інтерфейсу до різних розмірів екрану від мобільних пристроїв до настільних комп'ютерів; підтримку темної та світлої теми оформлення; швидке завантаження сторінок завдяки серверному рендерингу; типобезпечність кодової бази; зручність розгортання у хмарних сервісах; можливість роботи без реєстрації та авторизації користувача.

Вхідними даними для роботи системи є: код, який користувач вводить у вбудованому редакторі; необов'язкове введення зі стандартного потоку; вибір варіанта відповіді у тестових питаннях. Результатом роботи системи є виведення результату виконання Java-коду, повідомлення про правильність відповіді у тестах, оновлення показників прогресу та підсумкових результатів модулів.

1. ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1. Аналіз предметної області навчання обробки виняткових ситуацій у Java

Обробка виняткових ситуацій (exception handling) — це механізм мови програмування, призначений для опрацювання помилок, що виникають під час виконання програми, і дозволяє відокремити логіку штатного перебігу обчислень від логіки реагування на нештатні умови. У мові Java цей механізм є фундаментальним, оскільки інтегрований у систему типів та компілятор: значна частина бібліотечних класів кидає винятки, а компілятор примусово вимагає їх обробляти або декларувати у сигнатурі методу [1]. Без розуміння принципів роботи з винятками неможливо ефективно використовувати стандартну бібліотеку Java, виконувати операції з файловою системою, мережею, базами даних або багатопотоковими обчисленнями.

Виняткова ситуація у Java є об'єктом, що наслідує клас Throwable. Дві основні гілки ієрархії — Exception (відновлювані ситуації, які може й повинна обробляти прикладна програма) та Error (критичні помилки рівня віртуальної машини, такі як OutOfMemoryError та StackOverflowError, з яких звичайний код зазвичай не відновлюється). Окрему підгілку утворює клас RuntimeException — так звані неперевірювані (unchecked) винятки, які компілятор не зобов'язує обробляти явно: до них належать NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException та інші типові помилки програміста. Усі інші нащадки Exception (поза RuntimeException) — це перевірювані (checked) винятки, обробка яких контролюється компілятором [2].

Опанування механізму обробки винятків передбачає засвоєння як теоретичного, так і практичного матеріалу. До теоретичних аспектів належать: ієрархія класів Throwable; розрізнення checked та unchecked винятків; синтаксис конструкцій try-catch-finally, throw, throws, multi-catch, try-with-resources; механізм ланцюжків винятків (exception chaining); створення власних класів винятків;

найкращі практики та антипатерни обробки помилок. Практичний аспект охоплює здатність обирати тип винятку для конкретної ситуації, формулювати інформативні повідомлення, коректно структурувати блоки обробки, уникати поглинання помилок та надмірно широких блоків catch [3].

Особливість дисципліни полягає у тому, що для повноцінного засвоєння студент повинен не лише прочитати теоретичний матеріал, а й написати та запустити велику кількість невеликих фрагментів коду, щоб на власному досвіді побачити, як саме компілятор та віртуальна машина реагують на різні конструкції. У традиційному форматі викладання це вимагає попереднього встановлення JDK та інтегрованого середовища розробки, що для початківця створює суттєвий поріг входу та відволікає від суті навчального матеріалу.

Сучасні дослідження у галузі освітніх технологій підкреслюють ефективність так званого активного навчання — навчального процесу, у якому здобувач освіти не лише сприймає інформацію, а й безпосередньо взаємодіє з навчальним середовищем, отримує миттєвий зворотний зв'язок та має можливість одразу застосувати теоретичні знання на практиці [4]. Інтерактивні веб-платформи з вбудованим виконанням коду повністю реалізують цей підхід, оскільки усувають технічний бар'єр входу та забезпечують зворотний зв'язок у режимі реального часу.

Аналіз навчальних програм закладів вищої освіти України, у тому числі ПУЕТ, показує, що тема обробки винятків викладається у курсах об'єктно-орієнтованого програмування на молодших курсах інформаційно-технологічних спеціальностей. Типовий обсяг лекційного матеріалу складає 4–6 академічних годин, до яких додаються 2–4 години лабораторних робіт. За цей час студенту необхідно засвоїти значний обсяг понять, які тісно пов'язані між собою. Цьому процесу можуть суттєво посприяти інтерактивні навчальні засоби, що дозволяють розосередити навчальну активність у часі та надати студенту можливість самостійно опрацьовувати матеріал у зручному темпі [5].

На основі проведеного аналізу можна сформулювати ключові вимоги до сучасного навчального ПЗ з теми обробки виняткових ситуацій у Java: структурованість та модульність теоретичного матеріалу; наявність вбудованого

редактора коду з підсвічуванням синтаксису; можливість запуску Java-програм без локального встановлення JDK; автоматизована перевірка результатів виконання; різноманітність типів практичних завдань для відпрацювання різних навичок; контроль засвоєння через тестові питання; ведення статистики прогресу для рефлексії з боку студента. Зазначені вимоги покладено в основу розробки навчальної платформи, яка є предметом цієї кваліфікаційної роботи.

1.2. Огляд існуючих аналогів

На сучасному ринку освітніх онлайн-сервісів представлено значну кількість платформ, що пропонують навчальні матеріали з мови програмування Java. Для проведення коректного порівняльного аналізу було відібрано п'ять найпопулярніших ресурсів, які або повністю присвячені вивченню Java, або містять самостійний модуль з обробки виняткових ситуацій. Розглянемо кожен з них.

Codecademy — одна з найвідоміших у світі платформ інтерактивного навчання програмуванню, заснована у 2011 році [6]. Платформа пропонує курс «Learn Java» з кількома уроками, присвяченими обробці винятків. Серед сильних сторін: вбудований редактор коду, інтерактивні завдання з покроковою перевіркою, гейміфікація навчального процесу. Недоліки: курс англомовний; повний доступ до завдань потребує платної підписки Codecademy Pro; редактор має обмежену функціональність порівняно з повноцінними IDE; Java-вправи переважно зосереджені на основах синтаксису, а тема винятків розглядається оглядово, без занурення у специфіку checked/unchecked, multi-catch чи try-with-resources. Загальний інтерфейс платформи Codecademy подано на рисунку 1.1 (див. рис. 1.1).

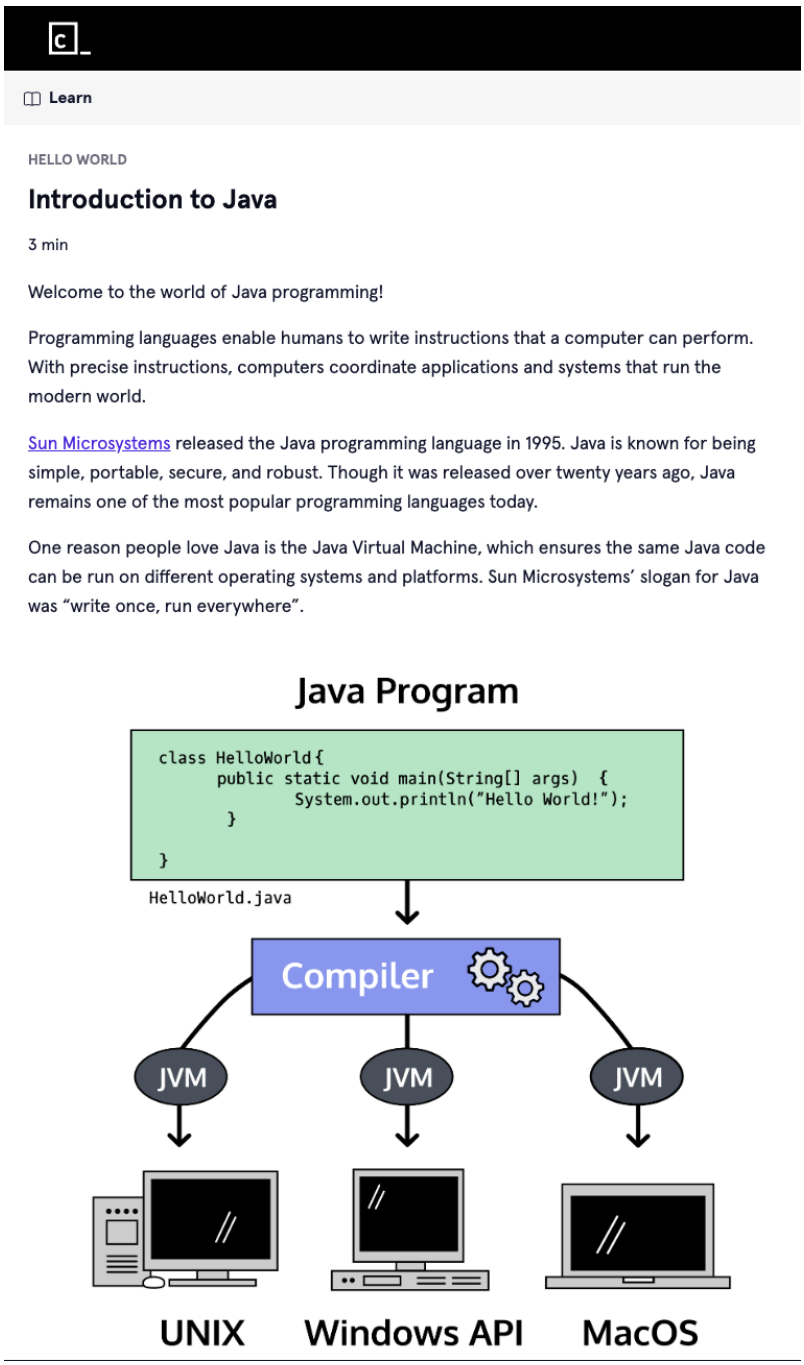

Рисунок 1.1 — Інтерфейс інтерактивного курсу Java на Codecademy

JetBrains Academy (раніше Hyperskill) — освітня платформа від компанії JetBrains, орієнтована на проєктно-орієнтоване навчання [7]. Курс Java Developer містить розгорнутий блок щодо обробки винятків з власними завданнями та тестами. Перевагою є інтеграція з IntelliJ IDEA, що дозволяє виконувати завдання у повноцінному локальному IDE. Однак це водночас і недолік для початківців: студент змушений встановити та налаштувати IntelliJ IDEA та JDK, що ускладнює

перший контакт з матеріалом. Платформа також україномовну версію наразі не пропонує, а доступ до більшої частини курсу платний.

W3Schools — популярний англомовний довідник з веб-технологій, що містить розділ Java Tutorial з підрозділом Java Exceptions [8]. Сильні сторони: безкоштовність; можливість запуску невеликих фрагментів коду через вбудований сервіс «Try it Yourself»; чіткий структурований виклад матеріалу. Недоліки: вкрай поверхневий розгляд механізму винятків (по суті, обмежується прикладом try-catch); відсутність системи прогресу; немає інтерактивних завдань з автоматичною перевіркою; немає підсумкових тестів для перевірки знань; інтерфейс не оновлювався протягом тривалого часу і виглядає застарілим.

JavaTpoint — індійський освітній портал з великою кількістю текстових матеріалів, орієнтованих на підготовку до співбесід [9]. Розділ «Java Exception Handling» є одним з найповніших серед оглянутих ресурсів і охоплює всі ключові теми, включно з кастомними винятками та ланцюжками. Однак платформа має суттєві недоліки: відсутність вбудованого редактора коду та можливості запуску прикладів у браузері (зразкові приклади подано виключно у вигляді статичного тексту); надмірна кількість рекламних блоків, що відволікають від навчального процесу; архаїчний інтерфейс без адаптивності та темної теми; відсутність системи практичних вправ з автоматичною перевіркою; англомовність контенту.

Codingame — ігрова платформа для розвитку навичок програмування через виконання творчих задач [10]. Підтримує Java серед інших мов і має зручний редактор коду на основі Monaco. Сильні сторони: розвинене ігрове середовище; реальне виконання коду на сервері; розгорнута система балів та змагальних турнірів. Недоліки: немає послідовних навчальних модулів з теорії; платформа орієнтована на досвідчених програмістів, а не на початківців; задачі не структуровані за темами на кшталт «обробка винятків»; інтерфейс англомовний.

Зведений візуальний образ розглянутих платформ дозволяє зробити висновок, що жодна з них не поєднує одночасно україномовний контент, повне покриття теми обробки винятків, безкоштовний доступ до всіх вправ, можливість запуску Java-коду у браузері без встановлення додаткового програмного забезпечення та

сучасний адаптивний інтерфейс. Загальний вигляд платформи Codingame наведено на рисунку 1.2 (див. рис. 1.2).

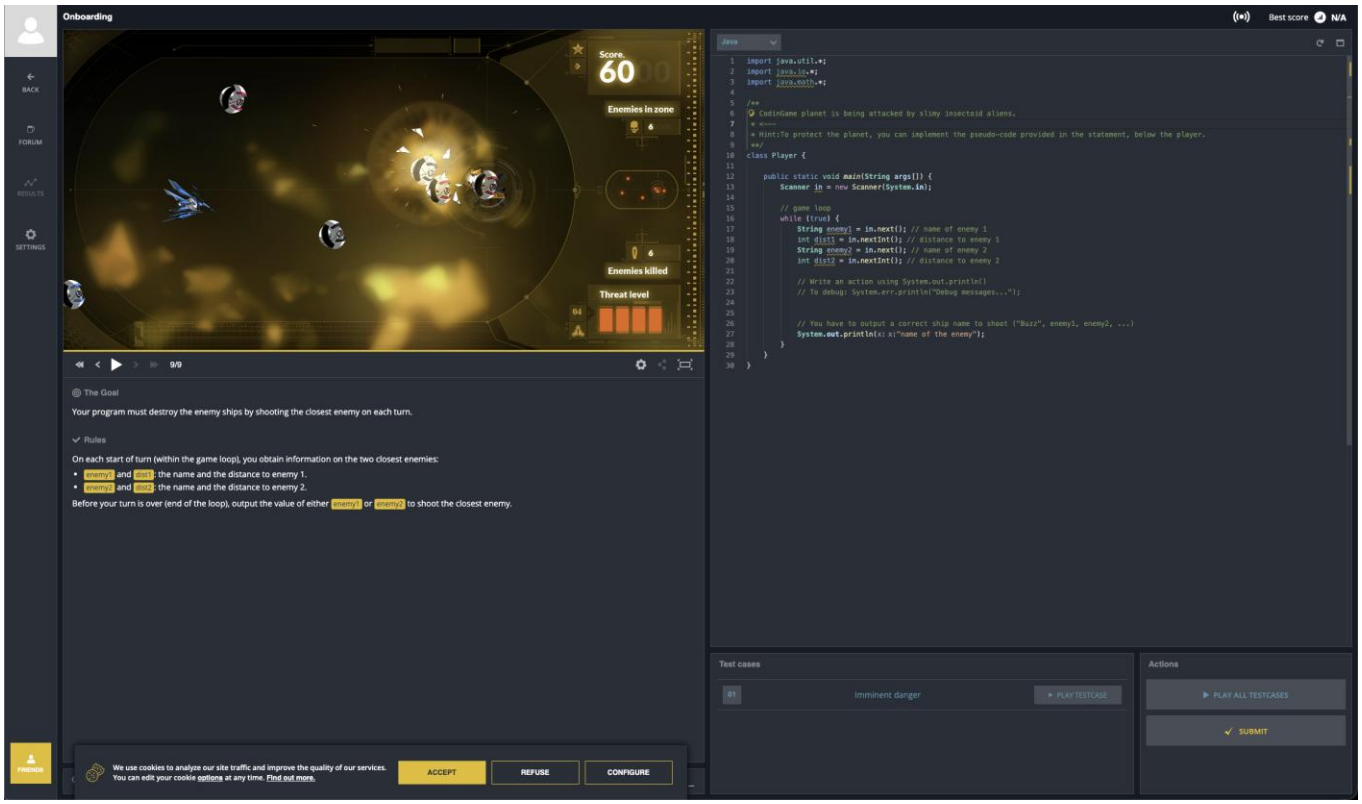

Рисунок 1.2 — Інтерфейс редактора коду на платформі Codingame

Аналіз вищезазначених аналогів виявив низку поширених недоліків: переважно англомовний контент; необхідність встановлення локальних інструментів для повноцінного виконання завдань; платний доступ до значної частини матеріалу; відсутність єдиного середовища для теорії, практики і тестування; недостатня глибина розгляду саме теми обробки винятків. Зазначені недоліки слугують додатковим обґрунтуванням актуальності розробки власного навчального ПЗ, що враховує українську освітню специфіку та вимоги дисципліни «Об'єктно-орієнтоване програмування».

1.3. Порівняльна характеристика аналогів та обґрунтування власної розробки

На основі здійсненого огляду існуючих рішень було сформульовано перелік критеріїв, за якими доцільно проводити порівняльну характеристику навчальних платформ. До таких критеріїв належать: україномовність контенту; безкоштовний доступ до повного обсягу матеріалу; наявність вбудованого редактора коду; можливість запуску Java-коду у браузері без встановлення додаткових інструментів; повне покриття теми обробки винятків; наявність інтерактивних практичних вправ з автоматичною перевіркою; наявність тестів для контролю знань; ведення статистики прогресу; адаптивність інтерфейсу та підтримка темної теми. Результати порівняння подано у таблиці 1.1.

Таблиця 1.1 — Порівняльна характеристика навчальних платформ

Критерій	**Codecademy**	**JetBrains Academy**	**W3Schools**	**JavaTpoint**	**Codingame**	**Розроблене ПЗ**
Україномовний контент	–	–	–	–	–	+
Безкоштовний доступ до всього обсягу	–	–	+	+	±	+
Вбудований редактор коду	+	±	±	–	+	+
Запуск Java-коду без встановлення JDK	+	–	+	–	+	+
Повне покриття теми обробки винятків	–	+	–	+	–	+
Інтерактивні вправи з автоперевіркою	+	+	–	–	+	+
Тести для контролю знань	+	+	–	±	–	+
Ведення статистики прогресу	+	+	–	–	+	+
Адаптивний інтерфейс і темна тема	+	+	–	–	±	+

У таблиці 1.1 використано такі позначення: «+» — критерій повністю виконується; «–» — критерій не виконується; «±» — критерій виконується частково. Як видно з таблиці, жодний з оглянутих аналогів одночасно не задовольняє усіх дев'яти сформульованих критеріїв. Найближчою за функціональністю є платформа JetBrains Academy, проте вона не пропонує україномовного контенту, не має повністю безкоштовного доступу та вимагає попереднього встановлення інтегрованого середовища розробки IntelliJ IDEA. Такі недоліки роблять її менш зручною для широкого впровадження у навчальний процес ПУЕТ.

Розроблене у межах кваліфікаційної роботи навчальне програмне забезпечення задовольняє усім сформульованим критеріям одночасно. Україномовність забезпечується підготовкою всього контенту державною мовою, що відповідає вимогам Закону України «Про забезпечення функціонування української мови як державної». Безкоштовний доступ зумовлений вибором архітектури без зовнішньої бази даних та використанням безкоштовного публічного API сервісу Piston для виконання Java-коду. Вбудований редактор реалізовано на базі Monaco Editor — того самого движка, що використовується у середовищі Visual Studio Code, що забезпечує професійний рівень підсвічування синтаксису. Повне покриття теми досягається за рахунок поділу матеріалу на вісім модулів, які охоплюють усі ключові аспекти обробки винятків у Java.

Окрім задоволення базових критеріїв, розроблене ПЗ має низку додаткових переваг: чотири різних типи практичних завдань, що дозволяють відпрацьовувати різні навички (доповнення коду, виправлення помилки, передбачення виводу, написання з нуля); деталізована система обліку спроб з розрізненням типів помилок (compile-error, runtime-error, wrong-output); глосарій термінів з прив'язкою до модулів; ігровий майданчик для вільної роботи з кодом; підтримка введення зі стандартного потоку для виконання інтерактивних програм; персистентне зберігання прогресу між сесіями браузера. Такий набір функцій робить розроблене ПЗ повноцінним самодостатнім навчальним інструментом, що може бути використаний без додаткового допоміжного програмного забезпечення.

Висновки за першим розділом: проведено аналіз предметної області навчання обробки виняткових ситуацій у Java; здійснено огляд п'яти найпопулярніших аналогів — Codecademy, JetBrains Academy, W3Schools, JavaTpoint та Codingame; на основі дев'яти сформульованих критеріїв побудовано порівняльну таблицю, яка показала, що жоден з аналогів не задовольняє усіх вимог одночасно. Сформульовано обґрунтування доцільності розробки власного навчального ПЗ та визначено його ключові переваги порівняно з існуючими платформами.

2. ТЕОРЕТИЧНА ЧАСТИНА

2.1. Концепції обробки виняткових ситуацій у мові Java

Механізм виняткових ситуацій у Java базується на трьох ключових поняттях: ієрархії класів виняткових ситуацій, синтаксичних конструкціях для їхньої обробки та правилах розповсюдження винятків стеком викликів. Розглянемо кожне з цих понять детальніше.

В основі ієрархії знаходиться клас java.lang.Throwable. Будь-який об'єкт, який може бути «кинутий» оператором throw або зловлений конструкцією catch, є нащадком цього класу. Throwable надає методи getMessage(), printStackTrace(), getCause(), getStackTrace(), які доступні усім винятковим ситуаціям. Безпосередніми нащадками Throwable є два класи — Error та Exception, що утворюють дві принципово різні гілки [1]. Графічне представлення ієрархії Throwable показано на рисунку 2.1 (див. рис. 2.1).

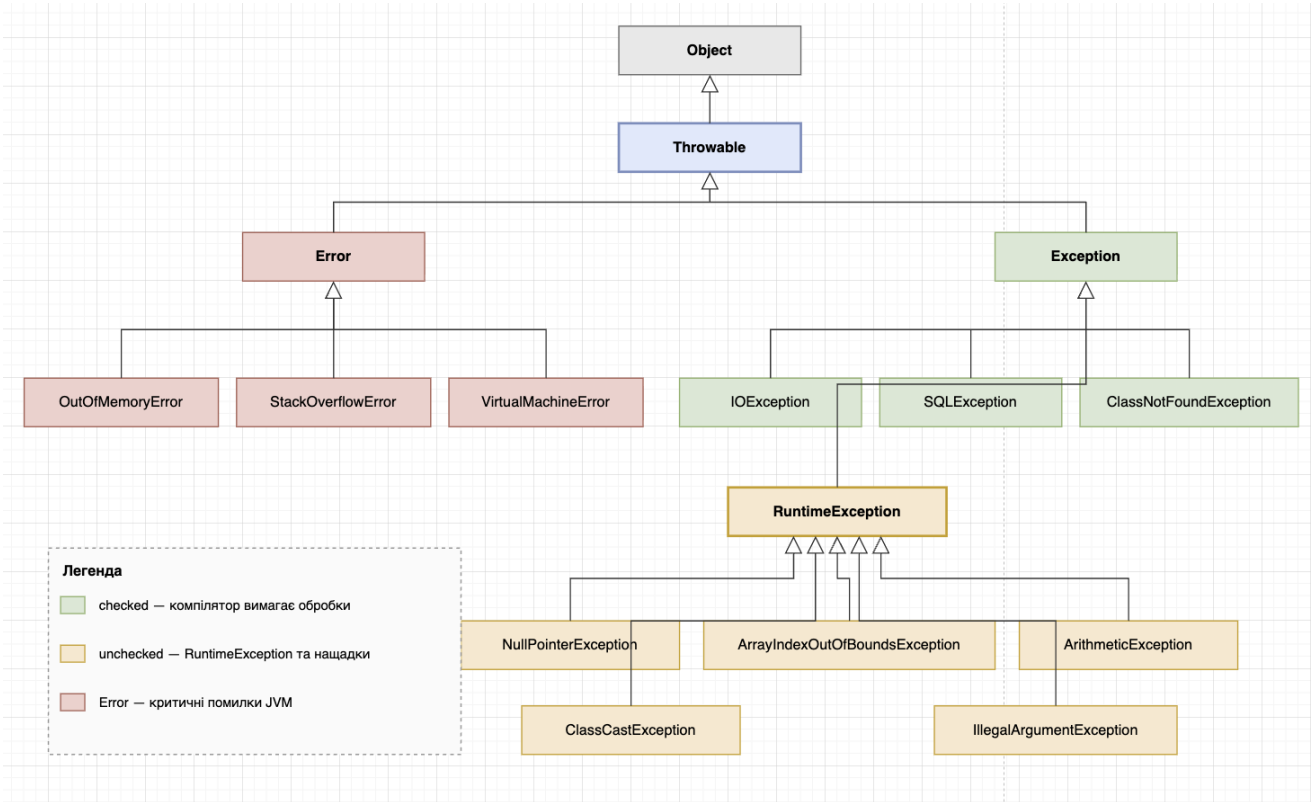

Рисунок 2.1 — Ієрархія класів виняткових ситуацій у Java

Гілка Error призначена для критичних помилок рівня віртуальної машини, з яких застосунок зазвичай не може відновитися: OutOfMemoryError виникає при вичерпанні пам'яті купи; StackOverflowError — при переповненні стеку викликів; VirtualMachineError свідчить про внутрішні проблеми JVM. За загальноприйнятою практикою, прикладний код не повинен ловити Error, оскільки після такої помилки стан програми вважається невизначеним.

Гілка Exception, навпаки, охоплює відновлювані ситуації, які прикладна програма може й повинна обробляти. Класи цієї гілки поділяються на дві категорії: перевірювані (checked) та неперевірювані (unchecked). Неперевірюваними є RuntimeException та усі його нащадки — NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, ClassCastException, IllegalArgumentException та інші. Компілятор Java не вимагає їхньої явної обробки, оскільки такі помилки переважно є наслідком програмних дефектів і повинні виправлятися на рівні логіки коду, а не маскуватися блоками catch [2]. Усі інші нащадки Exception, що не наслідують RuntimeException (IOException, SQLException, ClassNotFoundException), є перевірюваними: компілятор примусово вимагає або обгорнути виклик у try-catch, або декларувати throws у сигнатурі методу.

Базова конструкція обробки винятків — try-catch-finally. У блоці try розміщується код, що потенційно може кинути виняток; у блоках catch вказуються типи винятків, які слід зловити, та логіка реагування; блок finally виконується завжди — і за нормального виходу з try, і за вильоту винятку, і навіть якщо catch повертає значення (єдиний виняток — виклик System.exit, який припиняє роботу JVM до виконання finally) [3]. Допускається кілька catch-блоків, причому їхній порядок суттєвий: компілятор вимагає, щоб конкретніші типи передували загальнішим, інакше нижні блоки стануть недосяжними.

Версія Java 7 додала декілька важливих удосконалень. По-перше, синтаксис multi-catch дозволяє об'єднувати кілька типів винятків в одному catch-блоці через символ «|», за умови що ці типи не перебувають у відношенні наслідування. Це

зменшує дублювання коду, коли різні типи винятків обробляються однаково. По-друге, конструкція try-with-resources автоматично закриває ресурси, що реалізують інтерфейс AutoCloseable, після виходу з блоку try — навіть якщо стався виняток. Це усуває ризик витоку ресурсів через забутий виклик close() та робить код лаконічнішим за традиційну зв'язку try-finally.

Окрему важливу роль відіграє механізм ланцюжків винятків (exception chaining). У сучасних багатошарових системах низькорівневий виняток (наприклад, IOException під час читання конфігурації) часто потребує переведення у вищерівневий доменний виняток (наприклад, ApplicationStartupException). При цьому критично важливо зберегти оригінальну причину для подальшої діагностики. Конструктор Exception(String message, Throwable cause) та метод initCause() дозволяють пов'язати новий виняток з його першопричиною. Метод getCause() повертає цю причину, а виведення printStackTrace() автоматично виводить ланцюжок «Caused by:» з усіма рівнями. Це дає змогу зберегти інкапсуляцію між шарами без втрати діагностичної інформації.

Власні класи винятків доцільно створювати для домен-специфічних помилок. Конвенцією є наслідування від Exception для checked-винятків бізнес-логіки (наприклад, InsufficientFundsException у банківському застосунку) або від RuntimeException для unchecked-винятків програмних дефектів. Конструктори зазвичай повторюють підмножину з чотирьох конструкторів Exception: без аргументів, (String message), (Throwable cause), (String message, Throwable cause). Найменування повинно закінчуватися суфіксом Exception відповідно до загальноприйнятої традиції [4].

На практиці ефективність використання механізму винятків значною мірою залежить від дотримання найкращих практик і уникання типових антипатернів. До антипатернів належать: «поглинання» винятку (catch (Exception e) {}, що приховує помилку без жодних дій); надмірно широкий catch (Exception e), що ловить як очікувані, так і непередбачені винятки; повторне логування і кидання, що призводить до дублювання повідомлень у логах; використання винятків для звичайного керування потоком виконання, що погіршує продуктивність та

читабельність [5]. Натомість рекомендується ловити максимально конкретні типи винятків, логувати помилку лише один раз у відповідальному за неї шарі, активно використовувати try-with-resources для роботи з ресурсами та власні винятки для бізнес-логіки.

Окрім фундаментальних тем, повноцінне опанування механізму винятків у Java потребує ознайомлення з прикладними та продакшн-аспектами, що набувають особливого значення під час реальної інженерної практики. До таких тем належать: розширений синтаксис try-with-resources з existing variables, доступний з Java 9, який дозволяє передавати у блок try ефективно фінальні зовнішні змінні без створення локальних копій; коректна обробка InterruptedException та принципи переривання потоків у багатопотокових обчисленнях; особливості розповсюдження винятків у Stream API та лямбда-виразах, де checked-винятки потребують спеціального обгортання; вплив винятків на продуктивність застосунку та критерії, за якими доцільно уникати винятків у гарячих ділянках коду. На рівні промислової інженерії розглядаються також практики структурованого логування через бібліотеки SLF4J/Logback замість прямого використання System.err, реєстрація глобального обробника необроблених винятків через Thread.setDefaultUncaughtExceptionHandler, забезпечення коректних меж транзакцій у поєднанні з винятками, а також патерн «wrap & translate» для перетворення низькорівневих винятків у вищерівневі доменні з обов'язковим збереженням ланцюжка причин.

Узагальнюючи, концепція обробки винятків у Java є комплексною і вимагає ґрунтовного засвоєння на рівні теорії, синтаксису та практичних патернів. Саме цей перелік концепцій складає основу навчальних модулів розробленої платформи: ієрархія Throwable, базова конструкція try-catch-finally, throw/throws, multi-catch, try-with-resources, власні винятки, ланцюжки винятків, найкращі практики, прикладні аспекти у реальних задачах, продакшн-практики та підсумковий проєкт спрощеної банківської системи разом утворюють одинадцять тематичних модулів, послідовне вивчення яких забезпечує системне опанування предметної області.

2.2. Огляд та обґрунтування технологічного стеку

Вибір технологічного стеку є одним з найвідповідальніших етапів розробки програмного продукту, оскільки безпосередньо впливає на швидкість розробки, продуктивність кінцевого додатку, зручність супроводу та якість користувацького досвіду. У межах цієї роботи технологічний стек було сформовано з урахуванням специфіки задачі — створення інтерактивної освітньої веб-платформи з вбудованим виконанням Java-коду — та вимог до сучасних веб-додатків.

Як основний фреймворк було обрано Next.js версії 16 з архітектурою App Router. Next.js — це повноцінний фреймворк для побудови веб-додатків на базі бібліотеки React 19, що забезпечує серверний рендеринг, статичну генерацію сторінок, маршрутизацію на основі файлової системи, оптимізацію зображень та інших ресурсів, серверні маршрути API без потреби у окремому back-end-сервері [6]. Архітектура App Router базується на серверних компонентах React, що дозволяє виконувати значну частину логіки на сервері та передавати клієнту вже готовий HTML, мінімізуючи обсяг JavaScript, який завантажується у браузер. Серед альтернатив було розглянуто звичайний React з бібліотекою React Router, проте він вимагає окремого налаштування інструментів збірки, серверного рендерингу та маршрутизації, що збільшує час розробки. Інший популярний фреймворк — Remix — також було відкинуто на користь Next.js через ширшу екосистему документації, плагінів та готових компонентів, орієнтованих саме на освітні платформи.

Мовою програмування клієнтської та серверної частин обрано TypeScript — суворо типізовану надбудову над JavaScript, що компілюється у звичайний JavaScript. Перевагами TypeScript є: статична перевірка типів на етапі компіляції, що зменшує кількість помилок у виконуваному коді; покращена підтримка з боку IDE, включаючи автодоповнення, рефакторинг та навігацію; самодокументованість коду через явні описи типів даних; зручність роботи з великими кодовими базами завдяки механізмам інтерфейсів та узагальнень. Усі типи навчальних даних — модулів, вправ, питань тестів, прогресу користувача — описано через інтерфейси

TypeScript, що гарантує консистентність структур у клієнтській та серверній частинах.

Стилізацію інтерфейсу побудовано на базі Tailwind CSS версії 4 — utility-first CSS-фреймворку, що пропонує великий набір атомарних класів, які покривають усі типові потреби стилізації: відступи, шрифти, кольори, сітки, тіні, переходи [7]. Завдяки інтеграції з Next.js та механізму JIT-компіляції у фінальну збірку потрапляють лише ті класи, що реально використовуються, що мінімізує розмір CSS-файлу.

Для побудови складніших компонентів інтерфейсу — кнопок, карток, бейджів, прогрес-барів — використано власну бібліотеку примітивів, що поставляється у вигляді набору вихідних файлів безпосередньо у проєкті у каталозі components/ui. Кожен примітив побудовано на базі стандартних HTML-елементів та системі варіантів через бібліотеку class-variance-authority. Об'єднання класів, що походять з варіантів та користувацьких пропсів, виконується через комбінацію бібліотек clsx і tailwind-merge: clsx формує умовний рядок класів, tailwind-merge усуває конфлікти між Tailwind-класами однієї категорії (наприклад, два значення padding). Такий підхід забезпечує повний контроль розробника над стилем і поведінкою примітивів та усуває залежність від зовнішніх UI-бібліотек, що часто ламають зворотну сумісність.

Іконки в інтерфейсі подано через бібліотеку lucide-react — open-source-форк іконкового набору Feather Icons, що пропонує понад тисячу високоякісних SVG-іконок з підтримкою TypeScript та Tree Shaking [8]. Перевагою є нативна інтеграція з React, можливість задавати розмір, колір та інші атрибути через звичайні JSX-пропси.

Управління станом застосунку реалізовано на базі бібліотеки Zustand — мінімалістичного стейт-менеджера, що надає простий API на основі хуків React і не вимагає декларативних редюсерів та провайдерів, типових для Redux. Zustand повністю інтегрується з мідлваром persist, який автоматично серіалізує стан у localStorage браузера й відновлює його під час наступного відкриття додатку. Для цієї роботи такий підхід є особливо цінним, оскільки дозволяє зберігати прогрес

користувача без потреби у зовнішній базі даних або сервісі автентифікації — типовому пороговому функціоналі для невеликих освітніх проєктів. Альтернативами були Redux Toolkit (надмірно складний для поточної задачі), Recoil (експериментальний статус) та React Context з useReducer (швидко стає громіздким за зростання обсягу стану).

Для редагування коду користувачем використано Monaco Editor — той самий движок, що лежить в основі редактора Visual Studio Code [9]. Обгортка @monaco-editor/react спрощує його інтеграцію у React-компоненти. Серед переваг Monaco: професійне підсвічування синтаксису для понад 70 мов програмування, у тому числі Java; вбудована підтримка тем (зокрема vs-dark, що використано у проєкті); гнучке налаштування поведінки (мінімапа, нумерація рядків, розмір шрифту, режим тільки для читання). Альтернатива — CodeMirror 6 — теж потужний редактор, проте Monaco має ширше визнання та краще підходить для відображення великих фрагментів коду.

Підтримку темної теми реалізовано через CSS-медіазапит prefers-color-scheme та механізм класу dark Tailwind CSS, що автоматично відображає системну тему користувача без потреби у додаткових бібліотеках. Анімації переходів і мікровзаємодій реалізовано через стандартні утиліти Tailwind (transition-*, animate-*) без залучення сторонніх бібліотек анімації, що зменшує обсяг клієнтського JavaScript-бандла.

Сформований технологічний стек поєднує сучасні, добре задокументовані та активно підтримувані інструменти, що дозволило побудувати масштабовану архітектуру навчальної платформи з мінімумом зовнішніх залежностей та без потреби у власній серверній інфраструктурі.

2.3. Архітектурні підходи до побудови інтерактивних освітніх веб-додатків

Архітектура веб-додатку визначає, як саме розподілені обов'язки між клієнтською та серверною частинами, як відбувається маршрутизація, як зберігається стан застосунку та як відбувається обмін даними між компонентами системи. У сучасній веб-розробці виокремлюють три основні архітектурні моделі: класичні багатосторінкові додатки (Multi-Page Applications, MPA), односторінкові додатки (Single Page Applications, SPA) та гібридні архітектури з підтримкою серверного рендерингу [10].

Класичні MPA-додатки повертають клієнту окрему повноцінну HTML-сторінку у відповідь на кожен HTTP-запит. Перевагою такої архітектури є простота, швидкий «перший байт» та хороша індексація пошуковими системами. Недоліком — повне перезавантаження сторінки при кожному переході та обмежена інтерактивність. SPA-додатки, навпаки, завантажують з сервера один HTML-документ і весь подальший рендеринг відбувається на боці клієнта засобами JavaScript: переходи між «сторінками» виконуються інструментально через маніпуляцію History API. SPA забезпечують високу інтерактивність та плавні переходи, проте мають значний обсяг початкового JavaScript-бандла, гіршу індексацію та довший час до першого корисного контенту. Гібридні архітектури, до яких належить App Router у Next.js, поєднують переваги обох підходів: початковий рендеринг сторінки виконується на сервері, після чого клієнт отримує мінімізований JavaScript для подальшої інтерактивності [11].

Розроблена навчальна платформа використовує саме гібридну архітектуру. Архітектурну схему системи подано на рисунку 2.2 (див. рис. 2.2).

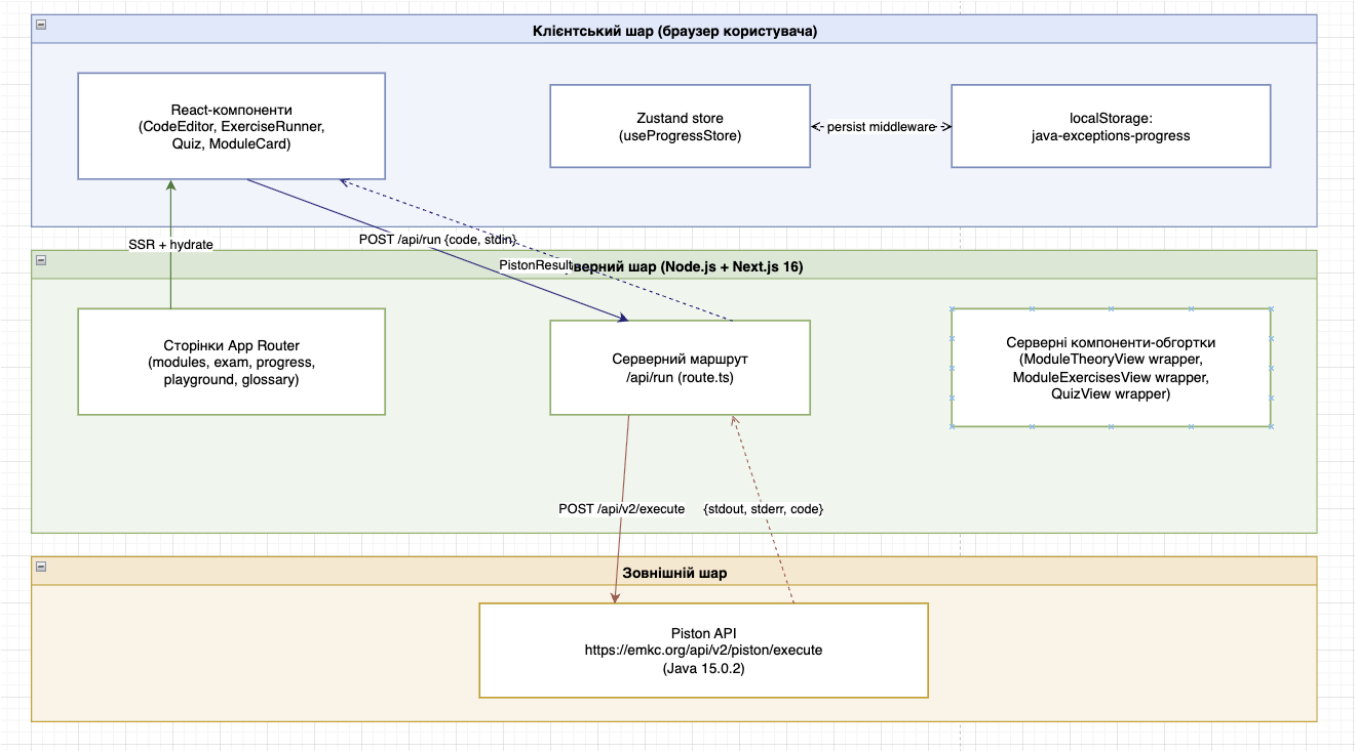

Рисунок 2.2 — Архітектурна схема навчальної платформи

На рисунку 2.2 виділено три основні логічні шари: клієнтський (браузер користувача з виконанням JavaScript та зберіганням прогресу у localStorage), серверний (Node.js-сервер Next.js з обробниками сторінок і маршрутами API) та зовнішній (публічний сервіс Piston для виконання Java-коду). Взаємодія між шарами відбувається за такими сценаріями: рендеринг навчальних сторінок здійснюється переважно на сервері з передачею клієнту готового HTML; інтерактивні компоненти (редактор коду, тести) виконуються на клієнті як React-компоненти; запуск Java-коду користувача проксується через серверний маршрут /api/run, який, своєю чергою, звертається до публічного API Piston та повертає результат клієнту; прогрес користувача зберігається у локальному сховищі браузера без участі серверу.

Маршрутизація у Next.js App Router базується на структурі файлової системи: кожен підкаталог у директорії app/ автоматично відповідає окремому маршруту, а наявність файлу page.tsx у такому каталозі вказує, що цей маршрут є кінцевою сторінкою. Динамічні сегменти позначаються квадратними дужками — наприклад, app/modules/[id]/page.tsx відповідає шляху /modules/1, /modules/2 тощо. Серверні

API-маршрути розміщуються у підкаталозі app/api/ та визначаються через експорт іменованих функцій GET, POST, PUT, DELETE з файлу route.ts. Такий підхід дозволяє чітко відокремити клієнтську та серверну логіку у єдиному репозиторії й позбутися потреби у супутньому back-end-сервері.

Розрізнення серверних і клієнтських компонентів у Next.js відбувається через директиву "use client", що розміщується першим рядком файлу. За замовчуванням компоненти вважаються серверними: вони рендеряться лише на сервері та не передають свій код у браузер. Клієнтські компоненти, навпаки, доставляються у вигляді JavaScript і підтримують весь арсенал інтерактивних можливостей React — стани (useState), ефекти (useEffect), доступ до DOM, обробники подій. Розроблена платформа використовує серверні компоненти для статичних сторінок (наприклад, лендінга, переліку модулів, теоретичної частини) та клієнтські — для інтерактивних елементів (редактора коду, тестів, форми введення стандартного потоку). Такий розподіл мінімізує обсяг JavaScript, що завантажується у браузер, та підвищує швидкість роботи додатку.

Управління станом застосунку поділяється на дві категорії: локальний стан окремого компонента (поточний код у редакторі, обраний варіант відповіді, прапорці видимості підказок) та глобальний стан, спільний для багатьох компонентів (прогрес користувача, статистика спроб, результати тестів). Локальний стан реалізовано через стандартний хук useState, глобальний — через сховище Zustand з мідлваром persist. Така архітектура дотримується принципу мінімально достатньої складності: глобальний стан виноситься поза межі компонентів лише тоді, коли цього вимагає реальна потреба обміну даними між віддаленими частинами дерева компонентів.

Зберігання даних у локальному сховищі браузера є типовим архітектурним рішенням для невеликих освітніх та інструментальних веб-додатків, у яких відсутність повноцінного облікового запису не є критичним недоліком. Перевагами цього підходу є відсутність необхідності у власній серверній БД, виключення проблем з конфіденційністю користувацьких даних та значна економія інфраструктурних ресурсів. Недоліком є обмеженість зберігання одним браузером і

одним пристроєм, а також ризик втрати даних при очищенні кешу. Для контексту дисциплінарної навчальної платформи такий компроміс є виправданим, оскільки прогрес користувача не є критично важливим артефактом і може бути відновлений повторним проходженням модулів [12].

Композиційна модель компонентів у React сприяє повторному використанню логіки та інтерфейсних елементів. У розробленій платформі виділено такі рівні композиції: примітивні компоненти інтерфейсу (Button, Card, Badge, Progress) у каталозі components/ui; складніші тематичні компоненти (CodeEditor, ExerciseRunner, Quiz, ProgressRing, ModuleCard, Navbar, TheoryContent, ModuleTheoryView, ModuleExercisesView, QuizView) у каталозі components; сторінки-маршрути у каталозі app/. Така ієрархія відповідає рекомендаціям офіційної документації React та забезпечує зрозумілу організацію кодової бази для подальшого супроводу.

Узагальнюючи, обрана архітектура базується на гібридній моделі рендерингу з чітким розподілом серверних і клієнтських компонентів, файловій маршрутизації, серверних API-маршрутах для проксіювання зовнішніх викликів та глобальному сховищі стану з персистентним збереженням у локальному сховищі браузера. Такий набір рішень дозволяє реалізувати повноцінну інтерактивну освітню платформу без власної back-end-інфраструктури, з мінімальним обсягом JavaScript у клієнтському бандлі та зручною подальшою еволюцією системи.

2.4. Засоби віддаленого виконання Java-коду: API Piston та редактор Monaco

Ключовою функціональною вимогою до розроблюваної навчальної платформи є можливість компіляції та виконання Java-коду користувача безпосередньо у браузері без встановлення локальних інструментів. Java є мовою, що компілюється у байт-код віртуальної машини JVM, тому її пряме виконання у клієнтському середовищі браузера неможливе. Хоча існують експериментальні проєкти на кшталт CheerpJ, які транслюють JVM-байт-код у WebAssembly, вони мають значний обсяг початкового завантаження (десятки мегабайтів) і досі не

забезпечують повної сумісності зі стандартною бібліотекою. Тому для виконання Java-коду використовується підхід зі зовнішнім сервером, який отримує код через мережевий запит, компілює та виконує його у ізольованому середовищі та повертає результат.

На ринку існує кілька спеціалізованих сервісів для віддаленого виконання коду. До найвідоміших належать: JDoodle Compiler API — комерційний сервіс з обмеженим безкоштовним планом та обов'язковою реєстрацією; Sphere Engine — платформа, що використовується у системах змагального програмування і потребує платної підписки; Judge0 — open-source-проєкт, який можна розгорнути самостійно, проте такий варіант вимагає підтримки власної інфраструктури; Piston — open-source-сервіс з публічним API, що не потребує реєстрації та доступний безкоштовно у межах розумного використання. Для розроблюваної платформи було обрано Piston як найбільш зручний для дипломного та освітнього використання сервіс [13].

Piston розробляється спільнотою EngineerMan і базується на ізольованих контейнерах. Він підтримує понад 60 мов програмування у різних версіях, у тому числі Java версій 15.0.2, 11.0.2, 10.0.2 та інших. Публічний інстанс сервісу доступний за адресою https://emkc.org/api/v2/piston/execute. Запит на виконання коду виконується методом HTTP POST з тілом у форматі JSON, що містить ідентифікатор мови, версію, перелік файлів з вихідним кодом та опціональне введення зі стандартного потоку. Відповідь містить структуру з полями stdout, stderr, code (код виходу процесу), а також метадані компіляції. Послідовність взаємодії компонентів системи під час виконання Java-коду користувача показано на рисунку 2.3 (див. рис. 2.3).

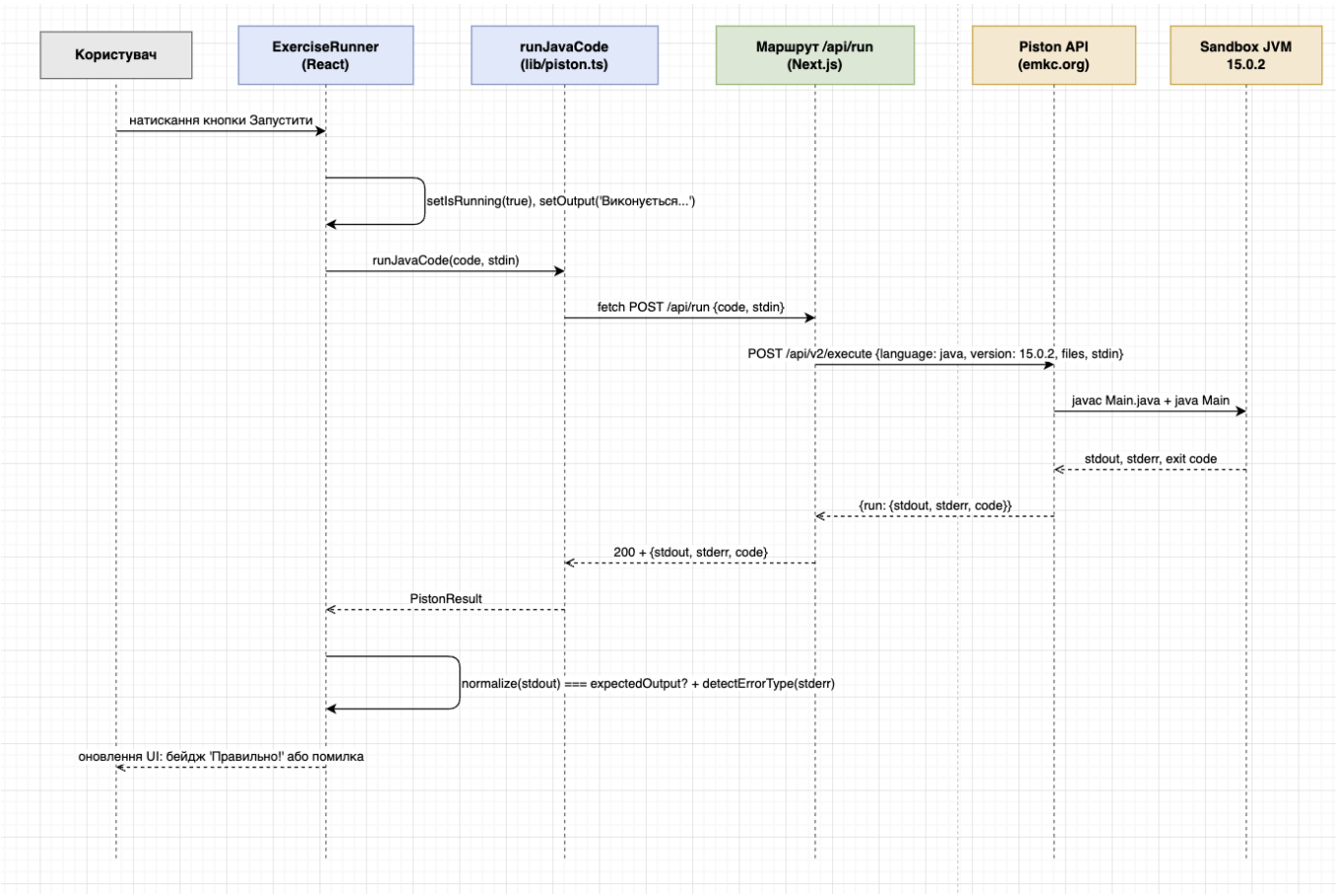

Рисунок 2.3 — Послідовність взаємодії під час виконання Java-коду

На рисунку 2.3 наведено типовий сценарій: користувач натискає кнопку «Запустити», після чого компонент ExerciseRunner надсилає POST-запит на серверний маршрут /api/run з тілом, що містить код та необов'язковий stdin; серверний маршрут проксує запит до публічного API Piston, передаючи додатково ідентифікатор мови «java» та її версію; Piston створює ізольоване середовище виконання, компілює код, запускає його, фіксує stdout, stderr та код виходу, після чого повертає результат серверному маршруту; маршрут трансформує відповідь у спрощену структуру і передає її клієнту; клієнт виводить результат у блок «Вивід програми» та порівнює stdout з очікуваним результатом вправи.

Серверне проксіювання запитів через власний маршрут /api/run має кілька переваг порівняно з прямим зверненням до Piston з браузера. По-перше, воно усуває проблеми CORS — обмежень міжсайтових запитів у браузері. По-друге, надає можливість централізовано контролювати конфігурацію (URL сервісу, версія мови) та підмінювати її через змінні середовища, наприклад, для локального тестування

проти приватного інстансу Piston. По-третє, спрощує можливе майбутнє додавання обмежень на частоту запитів, кешування, аутентифікації або моніторингу. Скріншот вкладки Network у DevTools браузера, який ілюструє цикл «запит-відповідь», подано на рисунку 2.4 (див. рис. 2.4).

Рисунок 2.4 — Запит до сервісу виконання коду в інспекторі мережевих запитів

Для написання коду на стороні клієнта використано Monaco Editor — open-source-редактор від Microsoft, що є основою інтегрованого середовища Visual Studio Code [14]. Розгорнута підтримка Java включає підсвічування ключових слів та операторів, нумерацію рядків, відступи відповідно до синтаксичних блоків, мінімапу, відображення помилок (linting у вбудованому режимі), згортання блоків коду. Monaco також забезпечує асинхронне завантаження мовних модулів, що дозволяє завантажувати лише потрібні мови та зменшує початковий розмір бандла.

Інтеграція Monaco у React-проєкт виконана через офіційний пакет @monaco-editor/react, що інкапсулює складну логіку завантаження редактора та надає простий декларативний інтерфейс на основі компонентів. Компонент CodeEditor у розробленій платформі обгортає базовий Editor і налаштовує його під потреби навчального продукту: фіксований темний фон vs-dark, шрифт 14 пунктів з нумерацією рядків, вимкнена мінімапа для збереження вертикального простору, автоматичний перерахунок розмірів при зміні контейнера, табуляція 4 пробіли, відключений режим продовження прокрутки після останнього рядка.

Введення зі стандартного потоку реалізовано через додатковий текстовий блок у компоненті ExerciseRunner. Кожен рядок цього блоку інтерпретується як окреме введення для Scanner або BufferedReader на стороні Java-програми. Така

функціональність є обов'язковою для більшості вправ із валідацією користувацьких даних — наприклад, перевірки коректності віку, парсингу числа з рядка або зчитування рядка з обробкою IOException.

Перевірка коректності виконання користувацького коду побудована на порівнянні текстових представлень фактичного stdout та очікуваного результату, нормалізованих процедурою заміни послідовностей \r\n на \n та видаленням пробілів на початку й у кінці. Така стратегія є компромісом між суворою бінарною ідентичністю та гнучкістю, необхідною для коректної обробки крос-платформних відмінностей перенесення рядка. Для розрізнення типів помилок реалізовано функцію detectErrorType, що аналізує stderr: за наявності рядка виду «error:» у поєднанні з посиланням на «.java:номер» помилка класифікується як compile-error, в усіх інших випадках — як runtime-error.

У результаті обрано та обґрунтовано два ключові інструменти для забезпечення інтерактивного виконання Java-коду у браузерному середовищі — публічне API Piston для серверного компілятора та виконавця коду, та редактор Monaco для клієнтського введення коду. Інтеграція цих інструментів через серверний маршрут API забезпечує безпечну, гнучку та легко конфігуровану архітектуру, що повністю задовольняє функціональні вимоги до розроблюваної платформи.

3. ПРАКТИЧНА ЧАСТИНА

3.1. Проєктування архітектури навчальної платформи

Розробка програмного продукту розпочинається з етапу проєктування, на якому формується загальна структура майбутньої системи, виокремлюються основні компоненти, визначаються їхні обов'язки та взаємозв'язки. Для розроблюваної навчальної платформи на етапі проєктування було ухвалено низку ключових рішень: відмова від власної бази даних та винесення стану користувача у локальне сховище браузера; використання серверних API-маршрутів виключно як легкого проксі для зовнішнього сервісу виконання коду; чіткий поділ на серверні та клієнтські React-компоненти; модульна організація навчального контенту з єдиною типізованою структурою.

Першим артефактом проєктування є діаграма використання (Use Case Diagram), яка ідентифікує всіх потенційних акторів системи та основні сценарії їхньої взаємодії з продуктом. У контексті розроблюваної платформи виділено єдиного актора — здобувача освіти, що відвідує сайт як неавторизований користувач. Зовнішнім актором у розширеному розумінні є сервіс Piston, який бере участь у сценарії виконання коду. Діаграму використання подано на рисунку 3.1 (див. рис. 3.1).

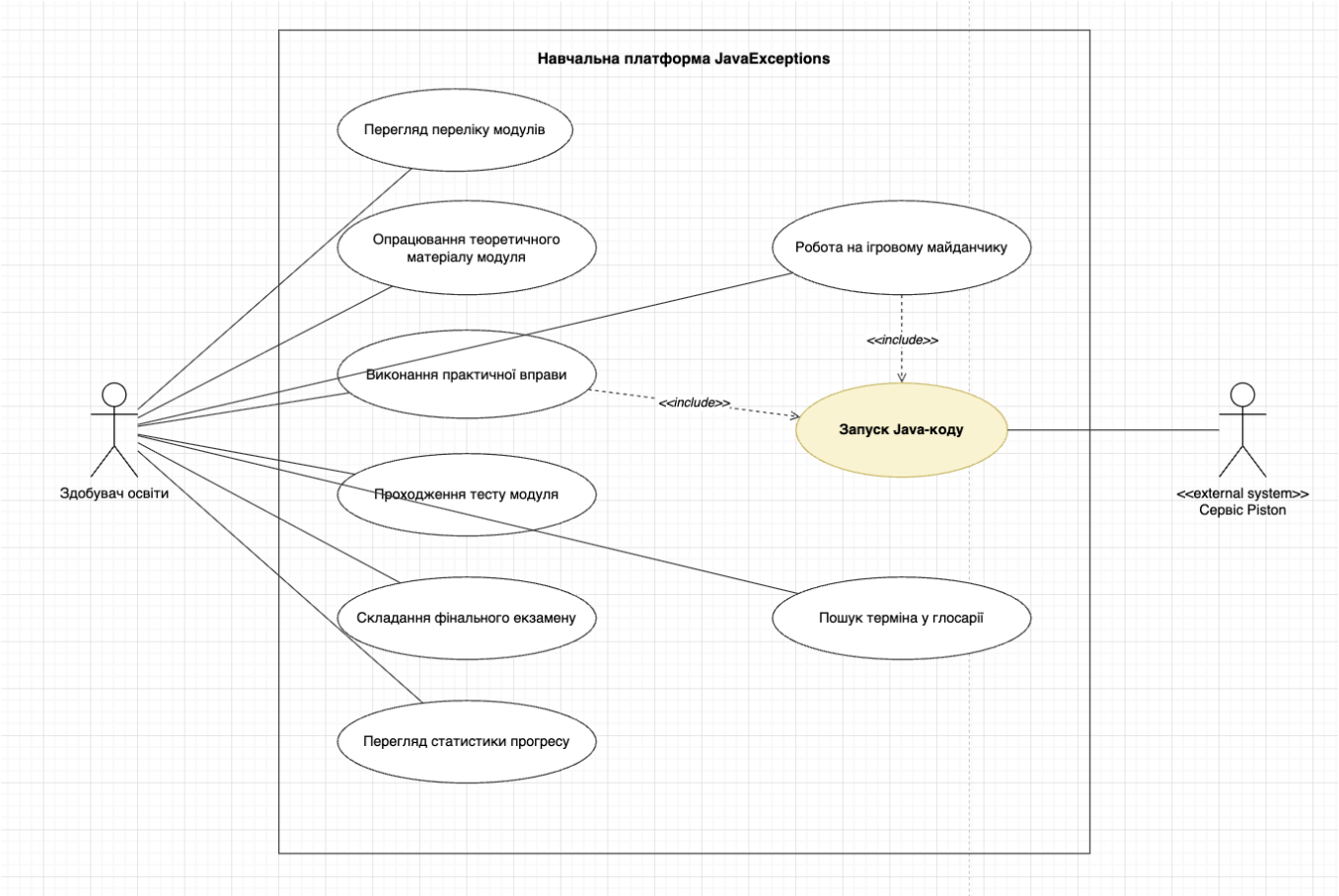

Рисунок 3.1 — Діаграма використання навчальної платформи

На рисунку 3.1 виділено такі основні сценарії використання: «Перегляд переліку модулів», «Опрацювання теоретичного матеріалу модуля», «Виконання практичної вправи», «Запуск Java-коду», «Проходження тесту модуля», «Складання фінального екзамену», «Перегляд статистики прогресу», «Робота на ігровому майданчику», «Пошук терміна у глосарії». Сценарій «Запуск Java-коду» включається у сценарії «Виконання вправи» та «Робота на ігровому майданчику» через відношення include і взаємодіє із зовнішнім актором — сервісом Piston.

Другим артефактом проєктування є структурна організація вихідного коду проєкту. Загальну файлову структуру навчальної платформи подано на рисунку 3.2 (див. рис. 3.2).

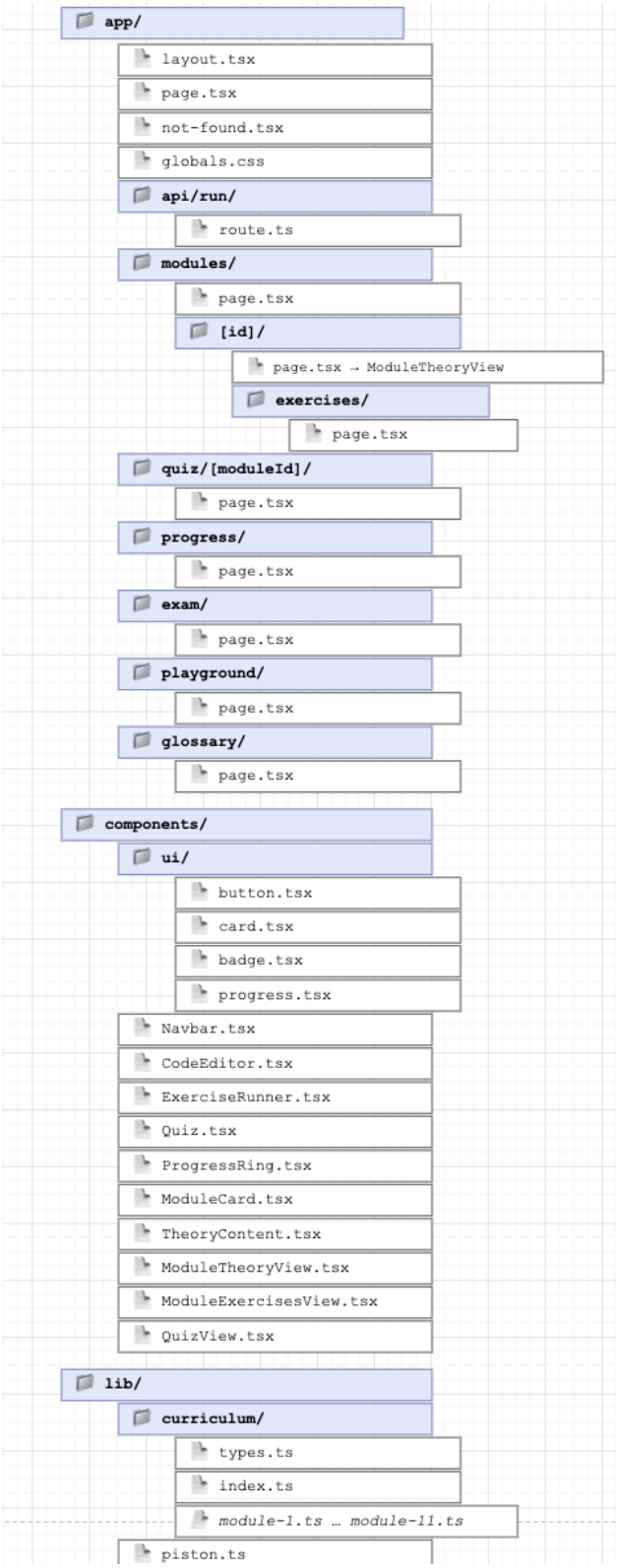

Рисунок 3.2 — Структура каталогів проєкту

Каталог app/ містить файлові маршрути Next.js: лендінгова сторінка page.tsx, перелік модулів modules/page.tsx, динамічний маршрут окремого модуля modules/[id]/page.tsx, сторінка вправ модуля modules/[id]/exercises/page.tsx, сторінка

тесту quiz/[moduleId]/page.tsx, сторінка прогресу progress/page.tsx, сторінка фінального екзамену exam/page.tsx, ігровий майданчик playground/page.tsx, глосарій glossary/page.tsx, серверний маршрут виконання коду api/run/route.ts, спільний макет layout.tsx, сторінка помилки 404 not-found.tsx та глобальні стилі globals.css. Каталог components/ містить дві групи компонентів: примітиви інтерфейсу у підкаталозі components/ui (Button, Card, Badge, Progress) та тематичні компоненти (CodeEditor, ExerciseRunner, Quiz, ProgressRing, ModuleCard, Navbar, TheoryContent, ModuleTheoryView, ModuleExercisesView, QuizView). Каталог lib/ містить чисто функціональні модулі: lib/curriculum/ — навчальний контент у вигляді типізованих структур, lib/piston.ts — клієнтський хелпер для виклику серверного маршруту, lib/progress.ts — глобальне сховище прогресу, lib/glossary.ts — словник термінів, lib/utils.ts — допоміжні функції.

Сторінкові маршрути модулів реалізовано за схемою «тонкий серверний обгортка + клієнтське представлення»: файл modules/[id]/page.tsx виконує лише декомпозицію динамічного параметра id і рендерить клієнтський компонент ModuleTheoryView. Аналогічно для маршрутів вправ та тесту створено компоненти ModuleExercisesView та QuizView. Такий поділ полегшує підтримку статичного експорту проєкту та чітко розмежовує серверну і клієнтську логіку.

Третім артефактом проєктування є концептуальна модель навчального контенту, представлена у вигляді інтерфейсів TypeScript у файлі lib/curriculum/types.ts. Модель оперує чотирма основними сутностями: Module — навчальний модуль; TheoryBlock — окремий блок теоретичного матеріалу всередині модуля (текст, код, примітка, попередження або діаграма); Exercise — практична вправа з кодом; QuizQuestion — питання тесту з варіантами відповідей. Кожна сутність має чітко визначений набір полів, що гарантує консистентність структури всіх восьми модулів та полегшує подальше розширення курсу.

Четвертим артефактом є модель прогресу користувача, описана у файлі lib/progress.ts. Глобальне сховище зберігає об'єкт типу ModuleProgress для кожного модуля, що містить прапорець ознайомлення з теорією, перелік виконаних вправ, статистику спроб з розрізненням типів помилок (компіляція, виконання,

неправильний вивід), результат тесту та прапорець завершення тесту. Окремо зберігається модель ExamProgress, що містить найкращий результат фінального екзамену, кількість спроб та позначку часу останнього проходження. Сховище інтегровано з мідлваром persist бібліотеки Zustand, що автоматично серіалізує його у localStorage під ключем «java-exceptions-progress» з контролем версії схеми.

Архітектурна декомпозиція навчальної платформи слідує принципу єдиної відповідальності: серверні API-маршрути обмежуються логікою проксіювання запитів до зовнішнього сервісу; клієнтські компоненти відповідають за рендеринг інтерфейсу та локальну логіку; функціональні модулі у lib/ — за чисту логіку без побічних ефектів. Така декомпозиція робить код передбачуваним, легко тестованим і відкритим до подальшого розвитку. Підтвердженням гнучкості архітектури є природне розширення курсу з вісьмох до одинадцяти модулів: для додавання нового модуля достатньо створити файл module-N.ts і зареєструвати його у переліку lib/curriculum/index.ts без жодних змін у компонентах інтерфейсу або логіці прогресу — глобальне сховище автоматично адаптує метрики до нової кількості модулів завдяки динамічній реалізації функції getTotalProgress.

3.2. Блок-схема алгоритму роботи системи

Для повного розуміння логіки роботи навчальної платформи необхідно сформувати алгоритм, що покроково описує дії користувача і реакцію системи у типовому сценарії проходження навчального курсу. Алгоритм оформлено у вигляді блок-схеми, яка візуалізує послідовність дій, точки прийняття рішень та цикли виконання. Структура блок-схеми відповідає вимогам стандарту ДСТУ ISO 5807:2016 «Інформаційні технології. Документація програмного забезпечення».

Загальний алгоритм роботи системи передбачає такі основні етапи: ініціалізацію сторінки лендінга та відновлення стану з локального сховища; перегляд переліку модулів і вибір цільового модуля; послідовне опрацювання теорії, виконання вправ та проходження тесту; перехід до наступного модуля; складання фінального екзамену після опрацювання всіх восьми модулів.

Узагальнену блок-схему алгоритму проходження курсу подано на рисунку 3.3 (див. рис. 3.3).

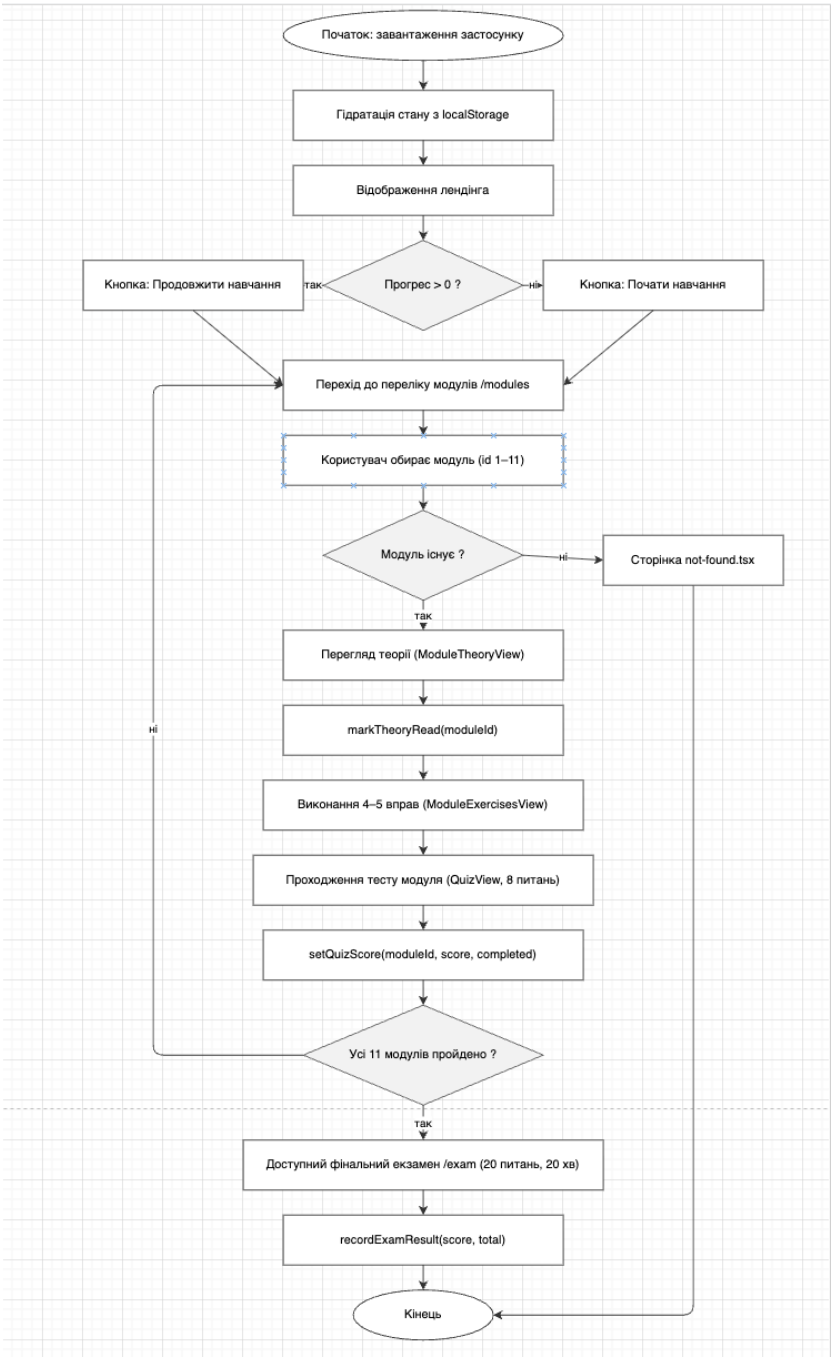

Рисунок 3.3 — Блок-схема загального алгоритму роботи навчальної платформи

Як видно з рисунка 3.3, після ініціалізації застосунку керування передається до екрана лендінга. Користувач може перейти до переліку модулів або до сторінки прогресу. Якщо обрано модуль, виконується перевірка ідентифікатора у динамічному маршруті: за коректного ідентифікатора відображається сторінка

модуля з вкладками «Теорія», «Вправи» та «Тест»; за відсутнього — рендериться спеціальна сторінка not-found.tsx. Після перегляду теорії користувач переходить до сторінки вправ, де реалізовано окремий цикл виконання чотирьох послідовних вправ. Завершення модуля передбачає проходження тесту з восьми питань, після чого користувач повертається до переліку модулів і обирає наступний. Цикл повторюється вісім разів — для всіх модулів курсу. Після завершення усіх модулів стає доступним фінальний екзамен як заключний контрольний етап.

Найбільш складною частиною загального алгоритму є цикл виконання практичної вправи, оскільки він містить взаємодію з зовнішнім сервісом, обробку трьох можливих типів помилок та оновлення глобального стану. Деталізовану блок-схему виконання вправи подано на рисунку 3.4 (див. рис. 3.4).

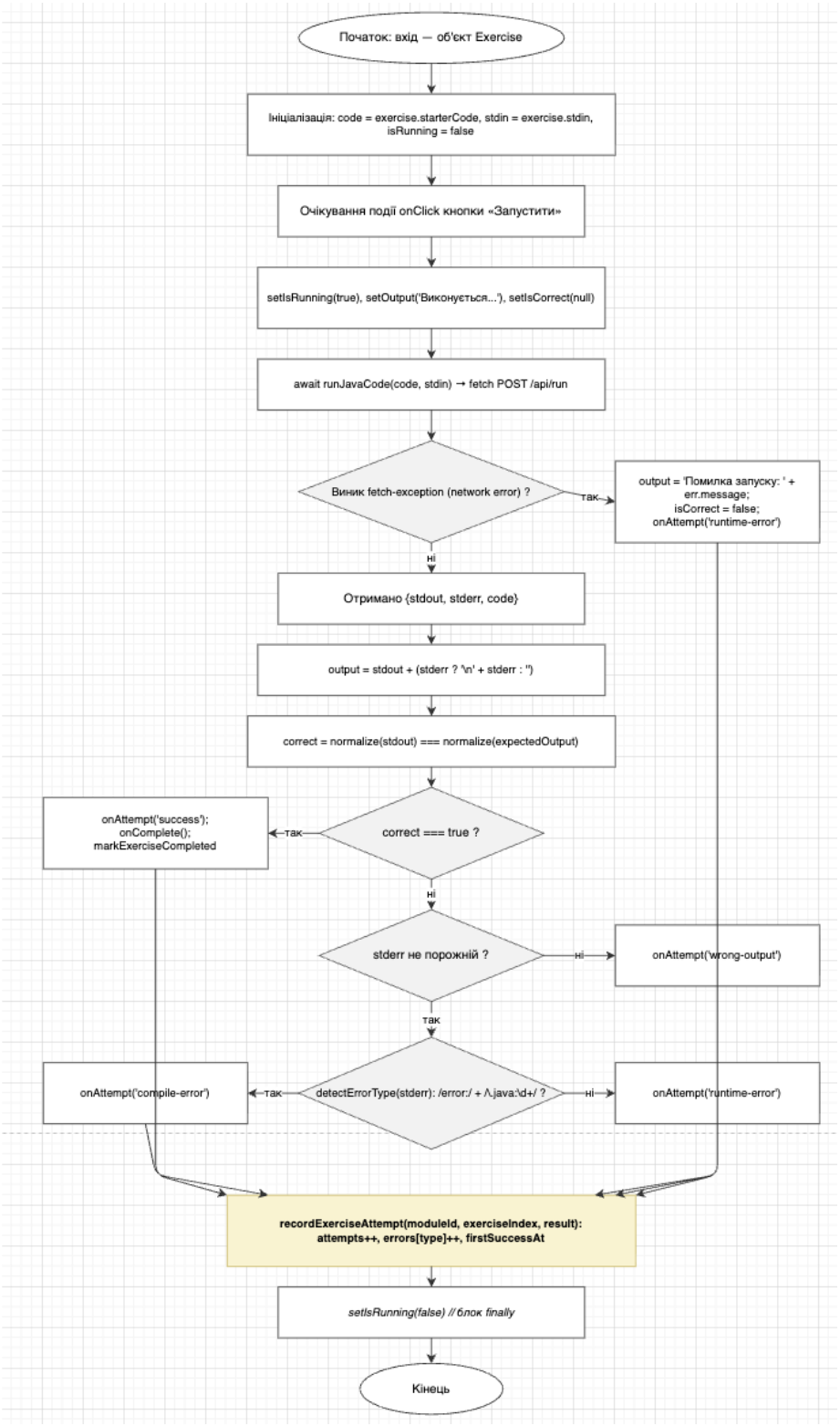

Рисунок 3.4 — Блок-схема алгоритму виконання практичної вправи

Алгоритм виконання вправи розпочинається з ініціалізації локальних станів компонента ExerciseRunner: код у редакторі встановлюється у значення starterCode з опису вправи, ввід зі stdin — у значення exercise.stdin (за наявності), вивід та індикатор правильності — у початковий стан. Далі система очікує події натискання

кнопки «Запустити». При обранні цієї дії компонент переводить себе у стан isRunning, відображає у блоці виводу повідомлення «Виконується...» та надсилає асинхронний POST-запит на серверний маршрут /api/run з тілом, що містить поточний код та stdin. Серверний маршрут проксує запит до публічного API Piston і повертає клієнту структуру з полями stdout, stderr та code.

Наступним кроком є перевірка коректності виконання. Спочатку у блоці виводу демонструється комбінований текст stdout та stderr (за наявності), щоб користувач бачив повний результат своєї спроби. Потім виконується нормалізація stdout та очікуваного результату: послідовності \r\n замінюються на \n, видаляються пробіли на початку і в кінці. Якщо нормалізовані рядки збігаються, спроба класифікується як успішна — викликається обробник onAttempt з аргументом «success», обробник onComplete (що оновлює прапорець виконаної вправи у глобальному сховищі), а у блоці виводу з'являється бейдж «Правильно!». У протилежному випадку, якщо stderr не порожній, викликається допоміжна функція detectErrorType: коли у stderr знайдено патерн виду «error: ... .java:номер», помилка класифікується як «compile-error», в інших випадках — як «runtime-error». Якщо stderr порожній, але вивід не збігається з очікуваним, спроба класифікується як «wrong-output». Усі ці типи помилок передаються у onAttempt і потрапляють у статистику модуля.

Особливим випадком є помилка мережевого запиту, що оброблюється у блоці catch навколо асинхронного fetch-виклику. У такому разі у блоці виводу з'являється повідомлення «Помилка запуску» з текстом виключення, прапорець isCorrect встановлюється у false, а у статистику записується «runtime-error». Після завершення обробки незалежно від результату прапорець isRunning повертається у false, що дозволяє користувачу запустити нову спробу.

Алгоритм оновлення статистики реалізовано у методі recordExerciseAttempt сховища Zustand. Для конкретної пари (moduleId, exerciseIndex) формується об'єкт ExerciseStats: загальна кількість спроб збільшується на одиницю; лічильники типів помилок оновлюються відповідно до результату; поле firstSuccessAt отримує номер поточної спроби лише за умови, що це перша успішна спроба і поле досі не

заповнене. Метод setQuizScore зберігає кількість правильних відповідей та прапорець завершення тесту, recordExamResult — оновлює найкращий результат фінального екзамену.

Окремим алгоритмом є обчислення загального відсотка прогресу, реалізоване у методі getTotalProgress сховища. Алгоритм перебирає всі вісім модулів і для кожного підраховує кількість завершених активностей: «теорію прочитано», «всі чотири вправи виконано», «тест пройдено». Підсумкова частка обчислюється як співвідношення завершених активностей до загальної кількості (24) і округлюється до цілого відсотка. Цей показник відображається на лендінговій сторінці та на сторінці прогресу у вигляді кільцевого індикатора (компонент ProgressRing).

Сформовані блок-схеми покривають усі ключові алгоритми навчальної платформи: загальний потік користувача, цикл виконання практичної вправи з перевіркою коду, оновлення глобальної статистики прогресу. Чіткість і однозначність наведених алгоритмів дозволяє використовувати їх як основу для подальшого програмування, документування та супроводу системи.

3.3. Розробка структури навчального контенту

Навчальний контент розроблюваної платформи побудовано за модульним принципом і охоплює основні теми обробки виняткових ситуацій у мові Java. Курс складається з одинадцяти послідовних модулів, кожен з яких має сталу внутрішню структуру: теоретична частина, набір з чотирьох-п'яти практичних вправ та тест з восьми питань множинного вибору. Така уніфікація полегшує сприйняття матеріалу здобувачами освіти та спрощує супровід контенту з боку викладача.

Перелік тем модулів сформовано на основі аналізу типового навчального плану дисципліни «Об'єктно-орієнтоване програмування» у закладах вищої освіти та доповнено блоком прикладних і продакшн-практик. Узагальнений склад курсу подано у таблиці 3.1.

Таблиця 3.1 — Структура навчальних модулів курсу

№	Тема модуля	Вправ	Питань тесту
1	Ієрархія винятків у Java	4	8
2	Конструкція try-catch-finally	4	8
3	Оператори throw та throws	5	8
4	Multi-catch (Java 7+)	4	8
5	Конструкція try-with-resources	4	8
6	Власні класи винятків	4	8
7	Ланцюжок винятків (chaining)	4	8
8	Найкращі практики та антипатерни	4	8
9	Винятки у реальних задачах	5	8
10	Production практики	4	8
11	Підсумковий проєкт — Banking System	5	8

Як видно з таблиці 3.1, загальний обсяг практичних завдань становить 47 вправ, тестових питань — 88. Послідовність модулів вибудувана за принципом наростання складності та глибини: перший модуль дає системне уявлення про ієрархію Throwable і слугує контекстною основою для подальшого матеріалу; модулі 2–4 присвячені базовому синтаксису обробки винятків; модуль 5 розкриває важливу тему автоматичного управління ресурсами; модулі 6–7 розглядають проєктні аспекти роботи з власними винятками та ланцюжками; модуль 8 узагальнює матеріал базового курсу через найкращі практики та поширені антипатерни. Модулі 9–11 виносять курс на прикладний та продакшн-рівень: модуль 9 присвячений винятковим ситуаціям у реальних задачах (try-with-resources Java 9+, InterruptedException, винятки у Stream API, performance impact); модуль 10 охоплює промислові практики обробки винятків (структуроване логування, глобальний обробник, transaction boundaries, патерн wrap & translate); модуль 11 є підсумковим інтеграційним проєктом — у ньому здобувачі освіти проєктують власну стратегію обробки винятків для невеликої банківської системи з трирівневою архітектурою «Service → Repository → Storage».

Кожен модуль описано як TypeScript-об'єкт, що відповідає інтерфейсу Module з файлу lib/curriculum/types.ts. Інтерфейс містить такі поля: id (числовий ідентифікатор модуля); slug (URL-сумісне строкове позначення для побудови

маршрутів); title (заголовок модуля); shortDescription (короткий опис, що відображається у переліку модулів); theory (масив теоретичних блоків); exercises (масив вправ); quiz (масив тестових питань). Реалізація інтерфейсу подана у Додатку А.

Теоретична частина модуля представлена масивом об'єктів типу TheoryBlock, що дозволяє чергувати у тексті п'ять типів блоків: text — основний прозовий текст із підтримкою елементарної розмітки (заголовки рівнів H2 та H3, маркіровані списки, виділення жирним у форматі **текст** та інлайн-код у форматі `текст`); code — багаторядковий блок коду з фіксованою моноширинною типографікою; note — інформативна примітка, виділена кольоровим фоном і піктограмою лампочки; warning — застереження, виділене жовтим фоном і піктограмою трикутника; diagram — псевдо-графічна діаграма у моноширинному шрифті, що використовується для побудови ієрархії класів. Розмаїття блоків дозволяє зробити теоретичний матеріал візуально насиченим і добре сприйнятним для здобувачів освіти.

Практичні вправи описано масивом об'єктів типу Exercise. Кожна вправа має тип, що належить до одного з чотирьох жанрів: «fill-in» — доповнення коду пропущеними фрагментами; «fix-bug» — пошук та виправлення помилки у заздалегідь некоректному прикладі; «predict-output» — передбачення результату виконання шляхом написання у коді відповідних println-викликів; «write-from-scratch» — написання повного розв'язку з нуля. Така диверсифікація типів забезпечує різнобічне відпрацювання навичок: «fill-in» розвиває знання синтаксису, «fix-bug» — діагностичне мислення, «predict-output» — глибоке розуміння семантики мови, «write-from-scratch» — самостійну побудову повноцінних програм. Поля starterCode та solution містять, відповідно, початковий шаблон і еталонний розв'язок; expectedOutput — очікуваний результат stdout; hint — текст підказки, що активується кнопкою «Підказка»; необов'язкове поле stdin — попередньо визначене введення зі стандартного потоку.

Тестові питання описано масивом QuizQuestion. Кожне питання має поле question з формулюванням, кортеж options з чотирьох варіантів відповіді, поле

correctIndex (значення 0–3) із номером правильного варіанта та поле explanation — текстове пояснення, що демонструється після обрання користувачем будь-якого варіанта. Пояснення є важливим педагогічним елементом: воно перетворює тестове питання з простого контролю на самостійну навчальну активність, незалежно від того, обрав користувач правильний варіант чи ні.

Окремою складовою навчального контенту є глосарій термінів, реалізований у файлі lib/glossary.ts. Глосарій містить понад тридцять записів типу GlossaryEntry, кожен з яких включає основний термін, опціональний перелік синонімів-аліасів, текстове означення та ідентифікатор пов'язаного модуля. На сторінці глосарію (app/glossary/page.tsx) реалізовано повнотекстовий пошук за основним терміном і аліасами, а також відображення прив'язки до модуля у вигляді кольорового бейджа.

Повний обсяг навчального контенту, реалізованого у проєкті, становить понад тридцять теоретичних блоків у кожному модулі (загалом близько 330 елементів), 47 практичних вправ з еталонними розв'язками та 88 тестових питань з поясненнями. Цей контент покриває всі одинадцять тем курсу — від базових концепцій ієрархії Throwable до продакшн-практик і підсумкового банківського проєкту — і забезпечує самодостатнє навчання обробки виняткових ситуацій у Java без потреби у додаткових зовнішніх матеріалах.

3.4. Реалізація компонентів інтерфейсу користувача

Інтерфейс навчальної платформи побудовано на основі компонентного підходу — фундаментальної парадигми бібліотеки React, що передбачає декомпозицію складних інтерфейсів на ієрархію невеликих, відносно незалежних компонентів. Кожен компонент інкапсулює власну візуальну презентацію та логіку, отримує дані через пропси і взаємодіє з зовнішнім середовищем через колбеки або глобальне сховище. У розробленій платформі ієрархія компонентів утворює два рівні: примітиви інтерфейсу у каталозі components/ui (Button, Card, Badge, Progress) та тематичні складніші компоненти (Navbar, ModuleCard, TheoryContent, CodeEditor, ExerciseRunner, Quiz, ProgressRing, ModuleTheoryView, ModuleExercisesView, QuizView).

Загальну структуру сторінок забезпечує спільний макет app/layout.tsx, що містить кореневий HTML-каркас, підключення глобальних стилів та компонент Navbar. Стартова сторінка app/page.tsx (див. рис. 3.5) виконує роль лендінга й містить три блоки: вітальний титульний екран з градієнтним заголовком та основними кнопками переходу; блок зведеної статистики курсу (кількість модулів, вправ та тестових питань, отримана з допоміжних функцій модуля curriculum); попередній перегляд переліку тем з короткими описами кожного модуля.

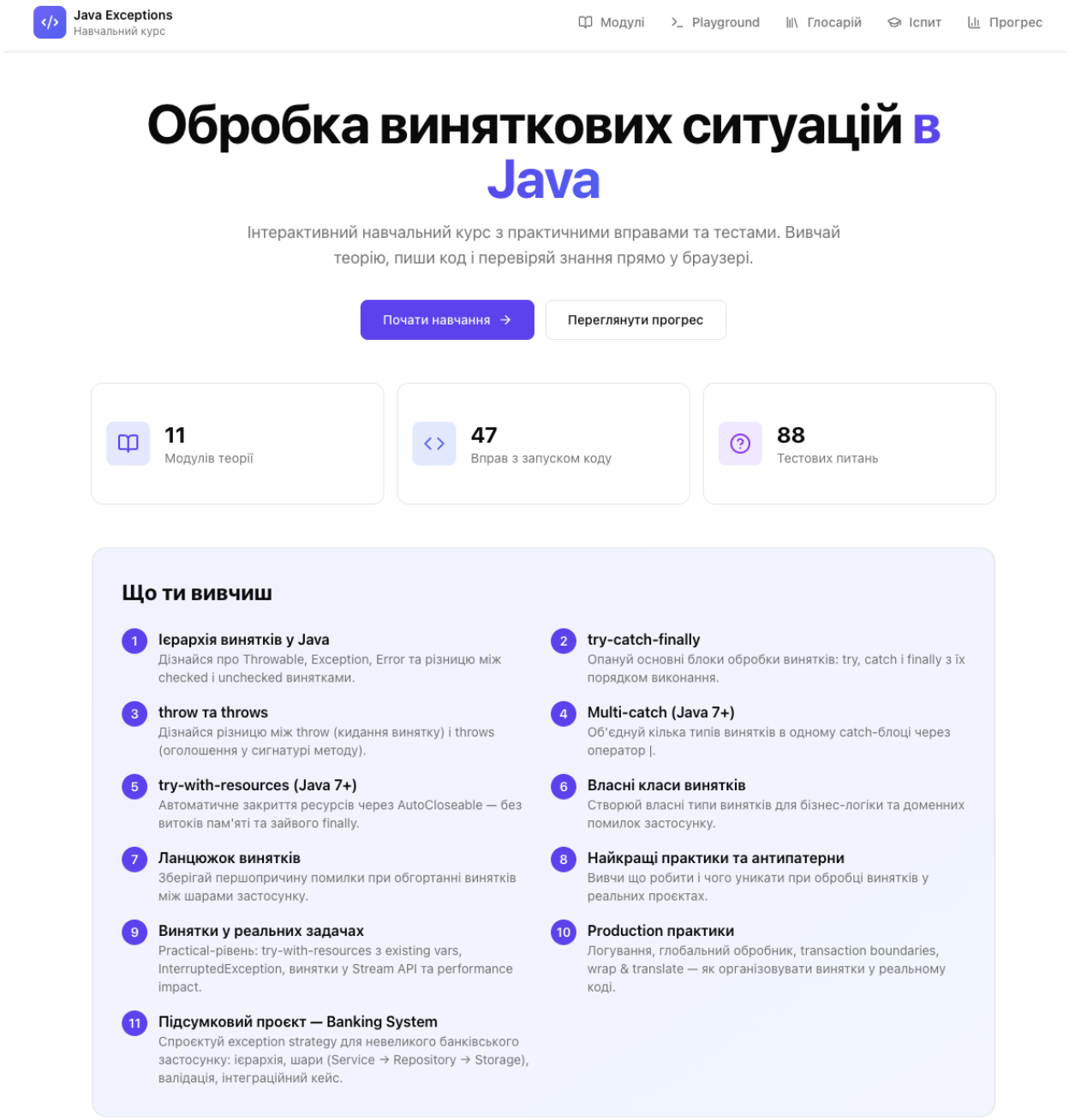

Рисунок 3.5 — Стартова сторінка навчальної платформи

Лендінгова сторінка реалізує також важливу деталь користувацького досвіду: вибір тексту головної кнопки залежить від наявності збереженого прогресу. Якщо користувач уже починав навчання, кнопка відображає напис «Продовжити навчання», у іншому випадку — «Почати навчання». Перевірка виконується через виклик useProgressStore та допоміжний хук useHydrated, що повертає прапорець завершення гідратації стану з localStorage. Гідратація необхідна для запобігання

помилкам неузгодженості серверного і клієнтського рендерингу у Next.js. Фрагмент відповідного коду подано у Додатку А.

Компонент Navbar реалізує верхню панель навігації з наступними пунктами: «Модулі», «Playground», «Глосарій», «Іспит», «Прогрес». Поточна сторінка підсвічується кольоровим виділенням, що визначається через хук usePathname Next.js. На широких екранах поряд з кожною піктограмою відображається текстова мітка, на мобільних — лише піктограми, що дозволяє зберігати читабельність на маленьких екранах без переходу на бургер-меню.

Сторінка переліку модулів (app/modules/page.tsx) реалізує сітку карток ModuleCard. Кожна картка містить порядковий номер модуля у вигляді кольорового кружка, заголовок, короткий опис, статус «Завершено / В процесі / Не розпочато» та піктограму, що відповідає поточному стану. Прогрес модуля обчислюється у реальному часі: «Завершено» виставляється, коли всі чотири вправи пройдено та тест завершено; «В процесі» — якщо є хоча б одна виконана вправа або прочитана теорія; «Не розпочато» — якщо стан модуля порожній. Загальний вигляд переліку модулів подано на рисунку 3.6 (див. рис. 3.6).

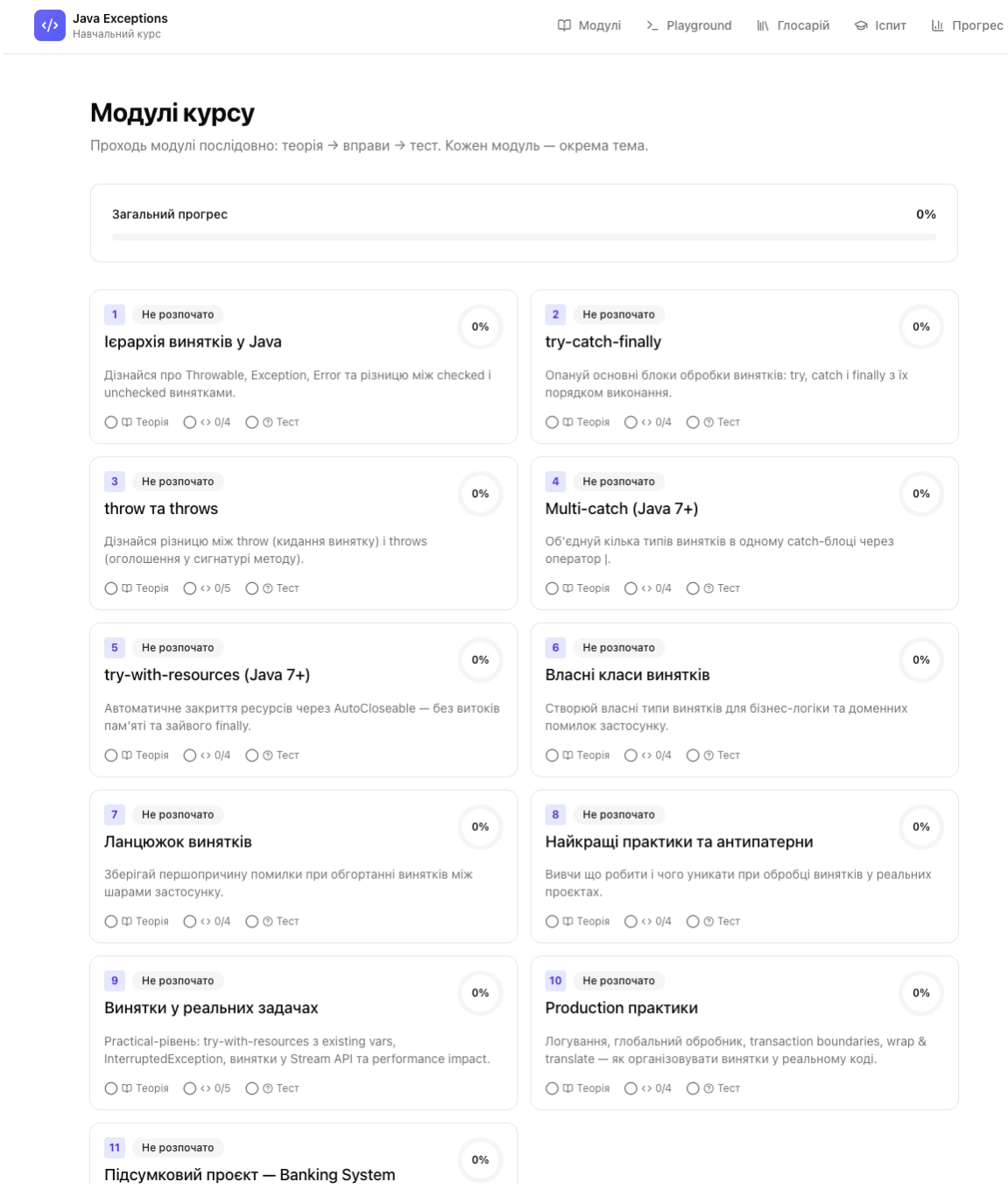

Рисунок 3.6 — Перелік навчальних модулів

Сторінка окремого модуля (app/modules/[id]/page.tsx) реалізована як тонкий серверний обгортка, що отримує ідентифікатор модуля з динамічного сегмента маршруту і передає його у клієнтський компонент ModuleTheoryView. Останній рендерить теоретичну частину модуля через підкомпонент TheoryContent, який приймає масив TheoryBlock і виводить блоки відповідно до типу: «text» — через функцію renderTextWithFormat з підтримкою заголовків H2/H3, маркіруваних списків, виділення жирним і інлайн-коду; «code» — як preformatted-блок з темним

фоном; «note» — як панель з лампочкою; «warning» — як панель з трикутником; «diagram» — як моноширинний блок з піктограмою інформації. Перехід до вправ модуля здійснюється кнопкою внизу сторінки, що веде на маршрут /modules/[id]/exercises, де відображається клієнтський компонент ModuleExercisesView. Перехід до тесту модуля — на маршрут /quiz/[moduleId], який рендерить компонент QuizView. Зразок відображення теоретичної частини подано на рисунку 3.7 (див. рис. 3.7).

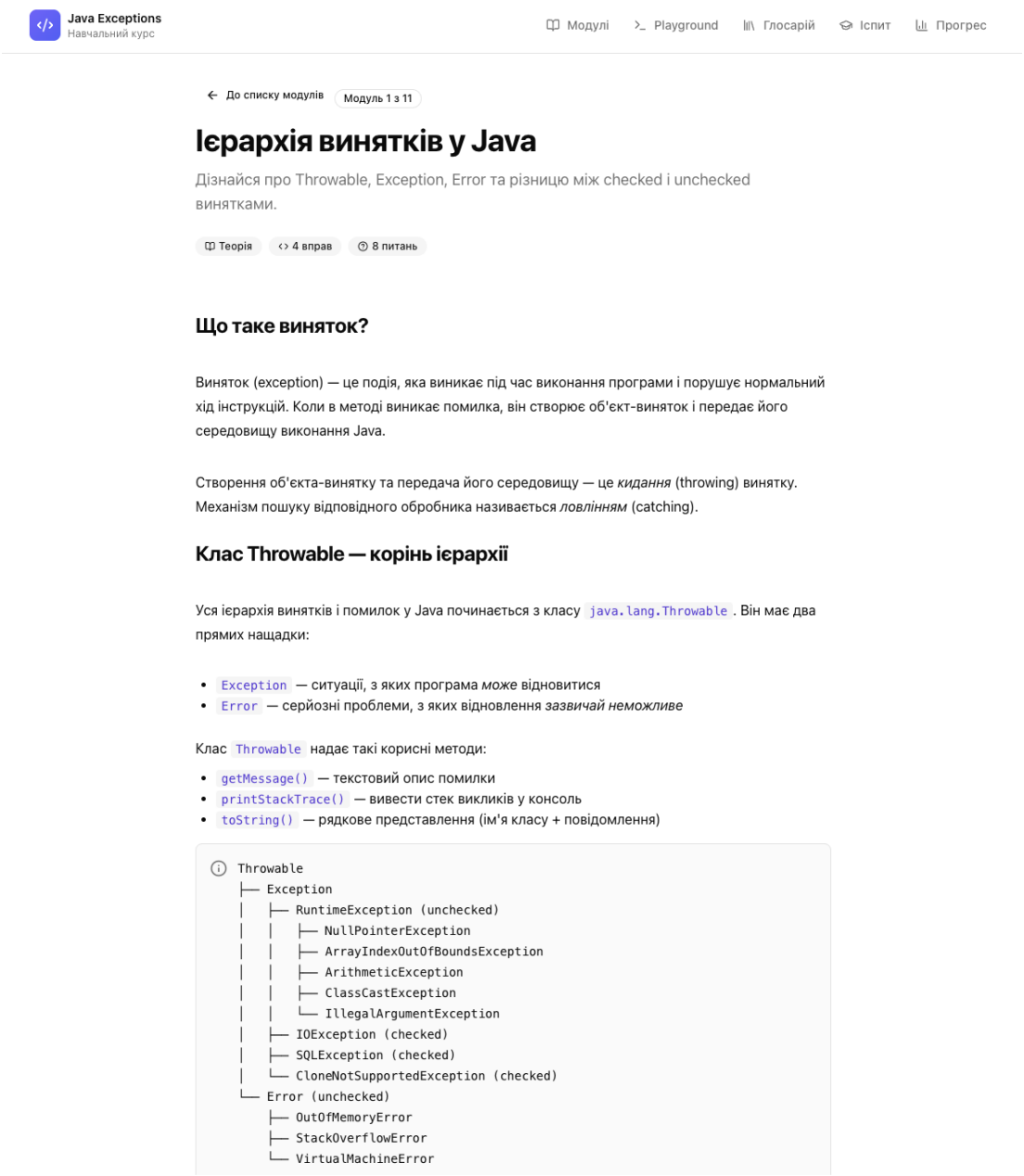

Рисунок 3.7 — Сторінка теоретичної частини модуля

Кнопка «Перейти до вправ» з'являється внизу теоретичної частини і викликає метод markTheoryRead глобального сховища, що встановлює прапорець ознайомлення для відповідного модуля. Перехід до сторінки вправ виконується програмно через хук useRouter Next.js. Аналогічно, кнопка «Перейти до тесту» на сторінці вправ стає активною лише після завершення всіх чотирьох вправ модуля.

Компонент ProgressRing — кільцева діаграма прогресу, реалізована через SVG-елементи. Компонент приймає числове значення відсотка (0–100) і параметр розміру у пікселях. Ефект заливки кільця досягається через властивість stroke-dasharray, яку обчислюють як добуток довжини окружності на частку прогресу. Анімація плавного переходу під час зміни значення забезпечується через CSS-властивість transition. ProgressRing використовується на лендінговій сторінці (як індикатор загального прогресу), на сторінці прогресу (для кожного модуля) та у компоненті ModuleCard (для статусу окремого модуля).

Стилізація компонентів повністю побудована на utility-класах Tailwind CSS 4. Це усуває потребу у написанні власних CSS-файлів і забезпечує єдину дизайн-систему. Кольорова схема узгоджена з поняттями Tailwind: основний акцент — індиго-600 (#4F46E5), додатковий — синій-500, фон у світлій темі — gray-50, у темній — gray-900, успіх — зелений-500, помилка — червоний-500. Перемикання між світлою і темною темами відбувається автоматично відповідно до системних налаштувань користувача через CSS-медіазапит prefers-color-scheme, реалізований у Tailwind через стратегію media для модифікатора dark.

Адаптивність інтерфейсу забезпечується префіксами Tailwind sm:, md:, lg: та xl:, що дозволяють декларативно змінювати класи відповідно до ширини viewport. Для перевірки адаптивності проведено візуальне тестування інтерфейсу на трьох контрольних розмірах: 1440×900 (desktop), 768×1024 (tablet) та 375×667 (mobile). Усі сторінки коректно адаптуються до мобільних екранів: багатоколонкові сітки перетворюються на однойменну колонку, заголовки зменшуються відповідно до брейкпоінтів, а навігаційне меню зберігає компактний вигляд за рахунок прихованих текстових міток.

Узагальнюючи, реалізована компонентна модель інтерфейсу забезпечує модульну, добре структуровану та адаптивну архітектуру, що задовольняє сучасним вимогам користувацького досвіду й зручна для подальшого розширення.

3.5. Інтеграція редактора Monaco та сервісу віддаленого запуску коду

Інтеграція редактора Monaco та сервісу Piston становить технічне ядро навчальної платформи, оскільки забезпечує можливість виконання Java-коду користувача безпосередньо у браузері. Реалізація цього функціоналу складається з трьох взаємодоповнювальних рівнів: клієнтський компонент-обгортка CodeEditor, що інкапсулює налаштування редактора Monaco; функціональний компонент-контролер ExerciseRunner, що поєднує редактор, керування станом виконання та валідацію результату; серверний маршрут /api/run/route.ts, що проксує запити до публічного API Piston.

Перший рівень — компонент CodeEditor — є тонкою обгорткою навколо базового компонента Editor з пакета @monaco-editor/react. Обгортка приховує надлишкову конфігурацію, що повторюється у всіх місцях використання редактора, та надає простий інтерфейс з трьох пропсів: value (поточний код), onChange (обробник зміни) та опціональні параметри height і readOnly. Усередині обгортки виставлено такі сталі налаштування: defaultLanguage = «java» — явне позначення Java для коректного підсвічування синтаксису; theme = «vs-dark» — темна тема, узгоджена з оформленням блоків коду у теоретичній частині; fontSize = 14, lineNumbers = «on», tabSize = 4, padding.top = 12 — налаштування для зручного читання коду; minimap.enabled = false — вимкнена мінімапа для збереження горизонтального простору; scrollBeyondLastLine = false — вимкнена прокрутка за межі останнього рядка; automaticLayout = true — автоматичне підлаштування розмірів при зміні розмірів контейнера. Скорочений вихідний код компонента подано у Додатку А.

Другий рівень — компонент ExerciseRunner — є відповідальним за повний цикл виконання вправи. Компонент приймає об'єкт Exercise, числові пропси

exerciseNumber та totalExercises (для відображення прогресу «Вправа N з M»), кількість попередніх спроб attempts та два колбеки: onComplete та onAttempt. Внутрішньо компонент керує сімома локальними станами через хуки useState: code (поточний код у редакторі), stdin (введення зі стандартного потоку), output (комбінований текст stdout/stderr), isRunning (прапорець виконання), isCorrect (тристанний результат: null/true/false), showHint (видимість підказки), showSolution (видимість еталонного розв'язку), showStdin (видимість блоку введення).

Інтерфейс компонента ExerciseRunner містить кілька зон: верхній блок з бейджами «Вправа N з M», типом вправи (укр.: «Доповни код», «Виправ помилку», «Передбач вивід», «Напиши з нуля») та лічильником попередніх спроб; заголовок та опис вправи; редактор коду CodeEditor з кнопками «Stdin», «Підказка», «Скинути»; опціональний блок введення stdin; область підказки; кнопки «Запустити» та «Показати рішення»; блок виводу з бейджем правильності; блок очікуваного виводу. Основний інтерфейс показано на рисунку 3.8 (див. рис. 3.8).

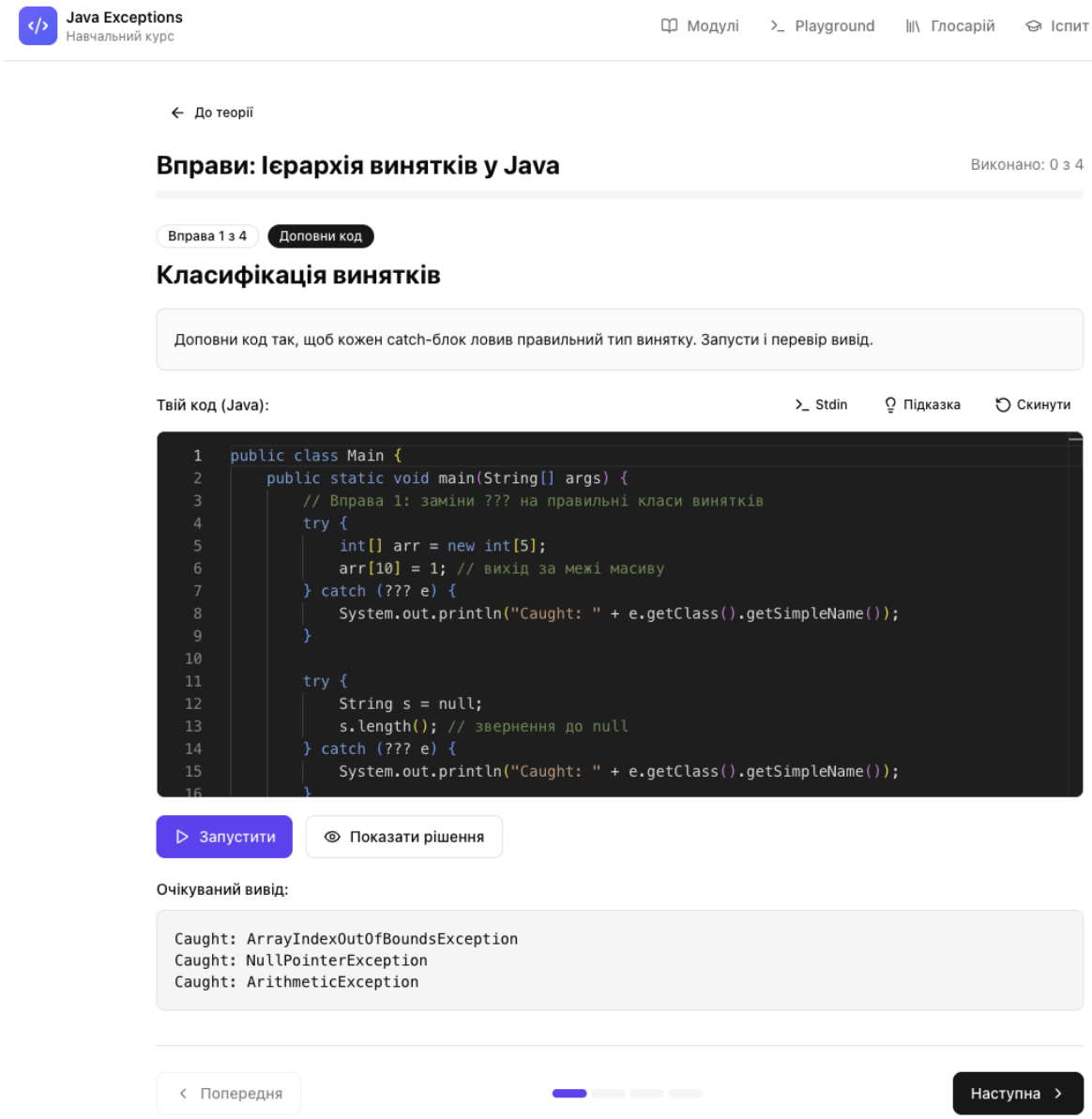

Рисунок 3.8 — Інтерфейс виконання практичної вправи

Логіка функції handleRun компонента ExerciseRunner виконує асинхронну послідовність дій. Спочатку у локальний стан виставляється isRunning = true, у блок виводу — повідомлення «Виконується...», а індикатор isCorrect — у null. Потім виконується спроба викликати функцію runJavaCode з модуля lib/piston.ts, передаючи поточний код та stdin. Результат — об'єкт PistonResult з полями stdout, stderr та code — формує комбінований вивід для відображення. Далі виконується нормалізація результатів через допоміжну функцію normalize і порівняння з очікуваним виводом. У разі збігу викликаються колбеки onAttempt(«success») та

onComplete; у разі розбіжності з наявним stderr — onAttempt з результатом detectErrorType(stderr); у разі розбіжності без stderr — onAttempt(«wrong-output»). Усі асинхронні помилки fetch обгорнено блоком try/catch, що формує дружнє повідомлення «Помилка запуску» та реєструє спробу як runtime-error.

Третій рівень — серверний маршрут app/api/run/route.ts, описаний як обробник POST-запитів. Маршрут спочатку виконує елементарну валідацію тіла запиту: за відсутності або некоректного типу поля code повертає HTTP 400 з повідомленням «Missing code». Далі з'єднується з API Piston через fetch на адресу, що читається зі змінної середовища PISTON_URL (з типовим значенням http://localhost:2000 для локального тестування проти приватного інстансу). Тіло запиту містить ідентифікатор мови «java», версію «15.0.2», масив files з єдиним файлом «Main.java» та значення stdin. Якщо API Piston повертає помилку, маршрут зберігає її текст і повертає клієнту HTTP 502 з відповідним повідомленням. У штатному випадку результат API трансформується у спрощену структуру { stdout, stderr, code } і повертається з кодом 200.

Клієнтський хелпер runJavaCode у lib/piston.ts виконує POST-запит на /api/run, обробляє статус відповіді та повертає об'єкт PistonResult. Перевага такої архітектури — клієнтський код жодного разу прямо не звертається до зовнішнього сервісу: всі мережеві деталі інкапсульовані у серверному маршруті. Це означає, що зміна сервісу виконання коду (наприклад, перехід на JDoodle або self-hosted Judge0) вимагатиме змін лише у одному файлі route.ts і не торкнеться клієнтських компонентів. Додатково передбачено демо-режим розгортання: змінна середовища NEXT_PUBLIC_DISABLE_CODE_RUN, виставлена у значення «true», вимикає звернення до сервісу виконання та виводить користувачу повідомлення з рекомендацією локального запуску через docker compose. Цей режим дозволяє розміщувати статичну демо-версію платформи на хостингах без виконавця коду, не порушуючи інтерфейсу.

Особливу увагу приділено обробці випадку, коли користувач використовує введення зі stdin. Така можливість необхідна для значної частини вправ — зокрема для модулів 3 (валідація віку через Scanner.nextInt()), 4 (парсинг числа з обробкою

NumberFormatException), 8 (рефакторинг коду з вводом). У ExerciseRunner блок stdin за замовчуванням приховано і відкривається або при наявності exercise.stdin (попередньо визначеного у вправі), або при натисканні кнопки «Stdin» з піктограмою терміналу. Введені дані передаються у Java-програму як вміст стандартного потоку — кожен рядок інтерпретується як окреме введення для Scanner або BufferedReader.

Реалізація детектора типу помилки (функція detectErrorType) використовує простий регулярний вираз для розрізнення помилок компіляції та виконання. Якщо у тексті stderr виявляється послідовність виду «error:» у поєднанні з координатами «.java:номер_рядка», помилка класифікується як compile-error; в інших випадках — як runtime-error. Така евристика покриває переважну більшість реальних випадків і дозволяє вести деталізовану статистику типів помилок без додаткового парсингу складних виводів JVM.

У результаті, інтеграція редактора Monaco та сервісу Piston реалізована як трирівнева архітектура з чіткими межами відповідальності. Клієнтська обгортка CodeEditor забезпечує уніфіковане налаштування Monaco; компонент-контролер ExerciseRunner керує станом виконання та валідацією; серверний маршрут /api/run проксує запити до Piston з повним приховуванням мережевих деталей від клієнта. Така архітектура поєднує надійність, простоту супроводу та гнучкість до подальшої заміни компонентів.

3.6. Реалізація системи прогресу користувача та фінального екзамену

Система прогресу є наскрізним компонентом навчальної платформи: вона збирає інформацію про навчальну активність користувача, забезпечує її збереження між сесіями та формує аналітичні метрики, які відображаються на дашборді прогресу. Реалізація системи побудована навколо одного глобального сховища Zustand, описаного у файлі lib/progress.ts. Сховище інкапсулює стан, методи його модифікації, обчислені властивості та налаштування персистентного зберігання у локальному сховищі браузера.

Структура стану складається з двох основних частин: modules — словник, що зіставляє ідентифікатор модуля з об'єктом ModuleProgress; exam — об'єкт типу ExamProgress, що описує стан фінального екзамену. ModuleProgress містить п'ять полів: theoryRead (булевий прапорець ознайомлення з теорією), exercisesCompleted (масив індексів виконаних вправ), exerciseStats (словник статистики спроб для кожного індексу вправи), quizScore (число правильних відповідей у тесті або null), quizCompleted (булевий прапорець завершення тесту). ExerciseStats для окремої вправи містить: загальну кількість спроб attempts; кількість помилок типу compile, runtime та wrongOutput; позначку часу першої успішної спроби firstSuccessAt. ExamProgress містить найкращий результат bestScore, загальну кількість спроб attempts та позначку часу останнього проходження lastCompletedAt.

Методи модифікації стану утворюють зрозумілий API з шести функцій: markTheoryRead встановлює прапорець ознайомлення; markExerciseCompleted додає індекс вправи у список виконаних, уникаючи дублювання; recordExerciseAttempt інкрементує лічильник спроб і відповідний лічильник помилок, оновлює firstSuccessAt; setQuizScore зберігає результат тесту і прапорець завершення; recordExamResult оновлює рекордний результат фінального екзамену та збільшує лічильник спроб. Усі методи реалізовані як іммутабельні оновлення з використанням spread-оператора, що відповідає рекомендованому стилю Zustand та React.

Обчислені властивості getModuleProgress, getExerciseStats, getTotalProgress, getTotalAttempts та getTotalErrors дозволяють отримати агреговані показники без дублювання логіки у компонентах. Зокрема, getTotalProgress реалізовано динамічно: метод перебирає всі модулі курсу через імпорт з модуля curriculum, обчислює кількість завершених активностей (теорія, вправи, тест) та повертає процентне співвідношення відносно загальної кількості, що обчислюється як кількість модулів, помножена на три (для поточних одинадцяти модулів — 33 активності). Така динамічна формула забезпечує коректність показника прогресу у разі подальшого розширення курсу новими модулями без жодних змін у коді сховища. getTotalAttempts і getTotalErrors агрегують відповідні лічильники з усіх ExerciseStats для всіх модулів.

Персистентне зберігання реалізовано через мідлвар persist бібліотеки Zustand. Налаштування мідлвару включають: ім'я ключа у localStorage — «java-exceptions-progress»; номер версії схеми — 2; функцію міграції migrate, що забезпечує безболісний перехід між версіями. Поточна функція міграції обробляє перехід з версії 1 на версію 2: якщо у збережених даних відсутнє поле exerciseStats всередині об'єктів модулів, воно ініціалізується порожнім словником; якщо відсутній корінь exam, він заповнюється значеннями за замовчуванням. Така архітектура дозволяє розширювати модель прогресу у майбутньому без ризику псування даних користувачів, які встигли почати курс на попередніх версіях продукту.

Сторінка прогресу (app/progress/page.tsx) є основним виглядом системи аналітики для користувача. У верхній частині розміщено великий компонент ProgressRing з відсотковим показником загального прогресу, що отримується через виклик getTotalProgress. Поряд відображаються ключові метрики: загальна кількість спроб виконання вправ, загальна кількість помилок, відсоткове співвідношення «успішних» спроб. Нижче розгорнуто перелік усіх восьми модулів з індивідуальними картками, кожна з яких містить заголовок, прогрес-індикатор, кількість виконаних вправ із чотирьох та результат тесту. Загальний вигляд сторінки прогресу подано на рисунку 3.9 (див. рис. 3.9).

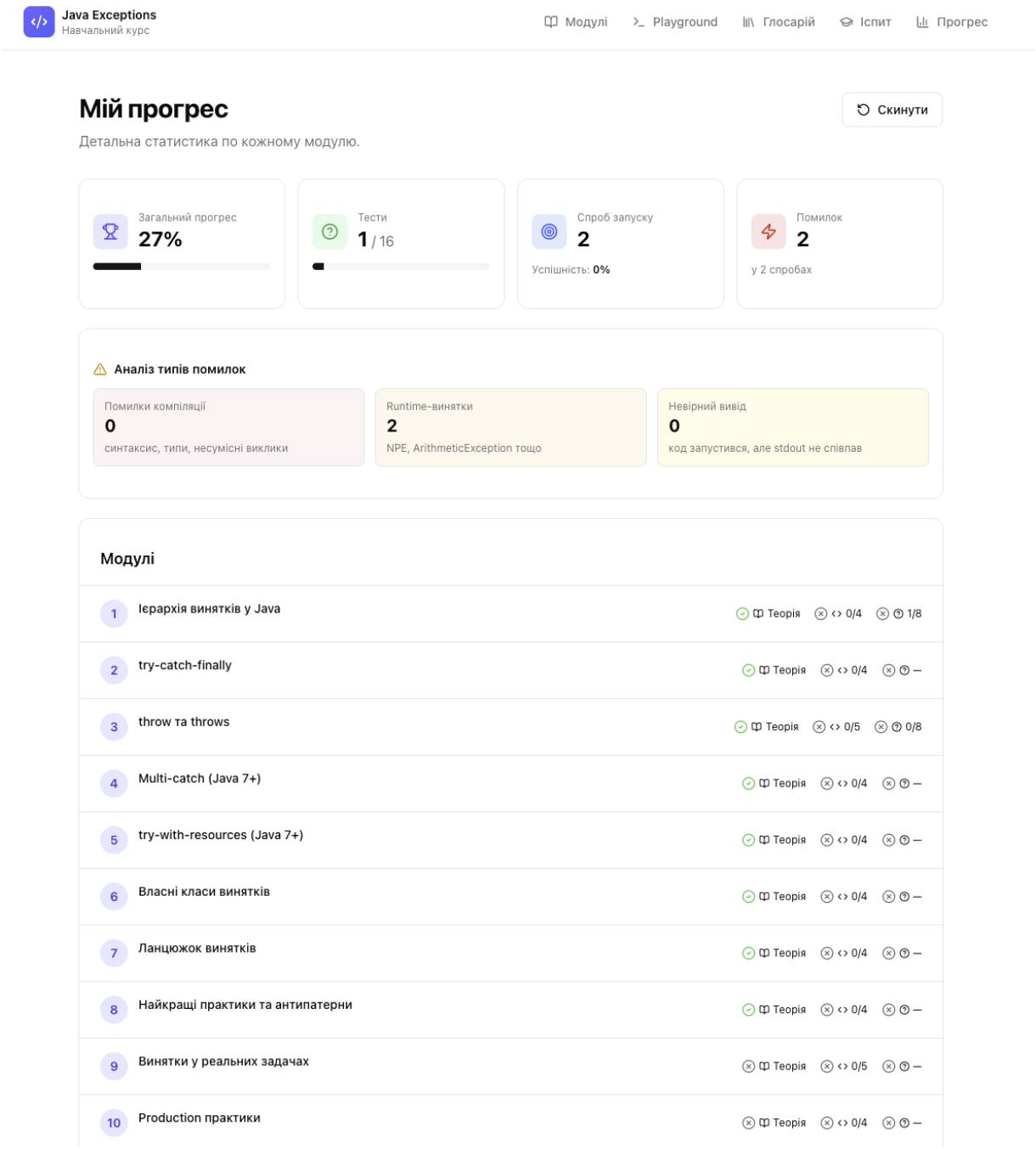

Рисунок 3.9 — Сторінка прогресу користувача

Фінальний екзамен (app/exam/page.tsx) реалізує підсумкову перевірку знань після проходження всіх модулів і побудований як трифазовий компонент із власною машиною станів: «intro» — стартовий екран з умовами проходження, «running» — активне складання екзамену з відліком часу, «finished» — підсумковий звіт. Алгоритм формування екзамену передбачає об'єднання всіх 88 тестових питань курсу у єдиний пул, перемішування його алгоритмом Фішера-Єйтса та вибір перших 20 питань (стала EXAM_QUESTION_COUNT = 20). Такий підхід

забезпечує рівномірний розподіл, виключає повторення та робить кожну спробу унікальною. На відміну від модульних тестів, фінальний екзамен обмежений у часі: загальний таймер становить 20 хвилин (стала EXAM_TIME_SECONDS = 1200), його зворотний відлік виконується через хук useEffect з інтервалом setInterval; за досягнення нульового значення поточний результат автоматично фіксується методом finishExam незалежно від кількості відповідей.

Інтерфейс екзамену побудовано на спеціалізованому компоненті, аналогічному за структурою компоненту Quiz, але з низкою відмінностей: видимий лічильник часу у форматі «хв:сс»; навігаційна сітка з номерами всіх 20 питань, що дозволяє переходити між ними у довільному порядку та повертатися до пропущених; відсутність негайного зворотного зв'язку — пояснення до питань не виводяться під час складання, а демонструються лише на підсумковому екрані після завершення. Така зміна підкреслює серйозність екзамену порівняно з тренувальними тестами окремих модулів. Після завершення викликається метод recordExamResult, що оновлює рекордний результат у локальному сховищі. Якщо набрано не менше 70 % правильних відповідей, на екрані з'являється поздоровлення з піктограмою «Trophy», за нижчого показника — заохочення «Можна краще — спробуй ще раз» та кнопка перепроходження. Інтерфейс підсумків фінального екзамену показано на рисунку 3.10 (див. рис. 3.10).

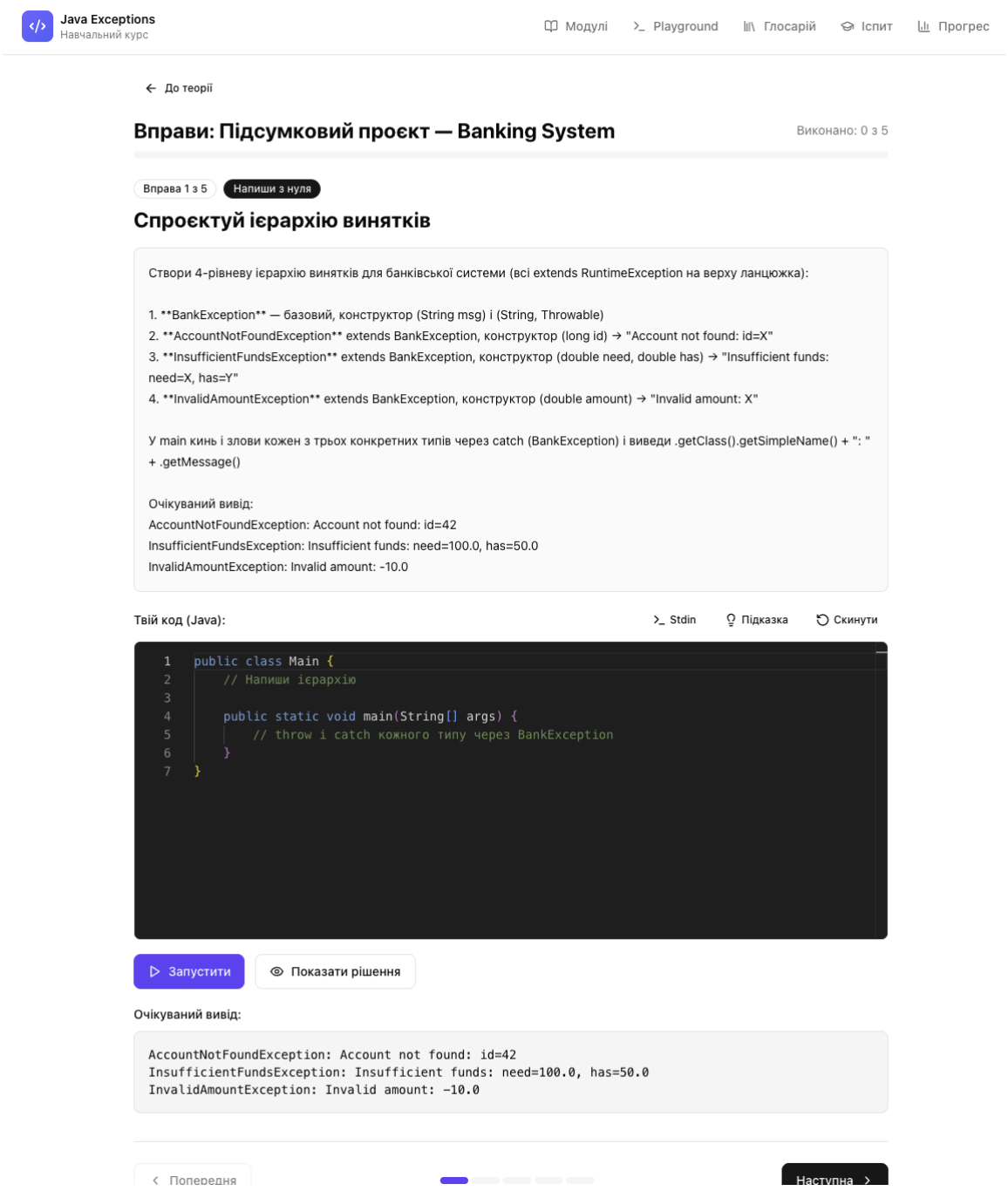

Рисунок 3.10 — Підсумковий екран фінального екзамену

Корисною деталлю для user-experience є реалізація хука useHydrated. Цей хук вирішує типову для Next.js проблему «гідратаційного зсуву»: серверний рендеринг не має доступу до localStorage, тому при першому відображенні сторінки прогрес дорівнює нулю; після завантаження клієнтського JavaScript стан відновлюється, і відсоток змінюється. Без додаткового захисту React зафіксує неузгодженість між серверним і клієнтським рендерингом, видавши попередження у консоль браузера. Хук useHydrated повертає прапорець завершення гідратації, що дозволяє у

компонентах прогресивно «розкривати» дані з localStorage без створення помилок гідратації. Атрибут suppressHydrationWarning використовується точково лише для невеликих текстових фрагментів (наприклад, тексту головної кнопки лендінга) — це краща практика, ніж глобальне відключення попереджень.

Окремою важливою функцією системи прогресу є можливість скидання у налаштуваннях. На сторінці прогресу присутня кнопка «Скинути прогрес», що викликає підтверджувальний діалог і у разі підтвердження повністю очищає localStorage під ключем «java-exceptions-progress». Така функція дозволяє користувачу почати курс з нуля без потреби у ручному видаленні даних через DevTools браузера.

У сукупності система прогресу та фінального екзамену надає здобувачу освіти прозорий зворотний зв'язок про рівень засвоєння матеріалу, мотивує до завершення всіх модулів та забезпечує об'єктивний інструмент підсумкової самоперевірки. При цьому, завдяки локальному характеру зберігання, реалізація системи не потребує бекенд-інфраструктури і повністю відповідає дипломному характеру проєкту.

3.7. Тестування навчальної платформи

Тестування є невід'ємним етапом життєвого циклу розробки програмного забезпечення, що дозволяє виявити дефекти, перевірити відповідність функціональних і нефункціональних вимог та підвищити надійність кінцевого продукту. Для розроблюваної навчальної платформи було розроблено комплексну стратегію тестування, що охоплює чотири основні види перевірок: функціональне тестування, інтеграційне тестування, юзабіліті-тестування та крос-браузерне і адаптивне тестування. Перевірки проводилися на зібраній виробничій версії застосунку, розгорнутій локально через команду npm run build та npm start.

Функціональне тестування виконано за принципом «чорної скриньки»: для кожної функціональної вимоги, сформульованої у постановці задачі, було сформульовано тест-кейс із вхідними даними, очікуваним результатом та

критеріями успішності. Головні групи тест-кейсів охоплюють такі сценарії: запуск Java-коду через редактор Monaco та сервіс Piston; валідація результатів виконання; ведення статистики спроб і помилок; проходження тестів модуля; складання фінального екзамену; персистентне збереження прогресу між сесіями. Перелік типових тест-кейсів подано у таблиці 3.2.

Таблиця 3.2 — Перелік ключових тест-кейсів функціонального тестування

№	**Сценарій**	**Очікуваний результат**	**Статус**
1	Запуск коректної Java-програми	Виведення stdout у блок результату	Пройдено
2	Запуск коду з помилкою компіляції	Класифікація як compile-error, +1 у лічильник	Пройдено
3	Запуск коду з runtime-винятком	Класифікація як runtime-error, +1 у лічильник	Пройдено
4	Збіг stdout з очікуваним	Бейдж «Правильно!», виклик onComplete	Пройдено
5	Розбіжність stdout з очікуваним	Класифікація як wrong-output	Пройдено
6	Введення зі stdin (Scanner)	Програма зчитує дані без помилок	Пройдено
7	Натискання «Скинути»	Код повертається до starterCode	Пройдено
8	Натискання «Показати рішення»	Код заміщується на solution	Пройдено
9	Завершення тесту з ≥70 %	Бейдж «Чудовий результат», збереження	Пройдено
10	Перезавантаження сторінки	Прогрес відновлюється з localStorage	Пройдено
11	Очищення прогресу	Стан повертається до початкового	Пройдено
12	Перехід на неіснуючий модуль	Сторінка not-found.tsx	Пройдено

Інтеграційне тестування зосереджене на правильності взаємодії між клієнтською частиною додатку та зовнішнім сервісом Piston. Для відтворюваності перевірок підготовлено набір контрольних Java-програм: «Hello, World!» — для перевірки базового потоку; програма з навмисно пропущеною крапкою з комою — для перевірки помилок компіляції; програма з NullPointerException — для перевірки runtime-помилок; програма з Scanner.nextInt() — для перевірки введення зі stdin; програма з некоректним перетворенням типу — для перевірки ClassCastException. Запит до публічного API Piston виконується через серверний маршрут /api/run і повертає результат у форматі { stdout, stderr, code }. Усі виклики у тестових сценаріях успішно завершилися протягом часу, що не перевищує дві секунди, що

відповідає очікуванням до інтерактивного навчального інструменту. Час відповіді сервісу контролювався через вкладку Network у DevTools браузера (див. рис. 3.11).

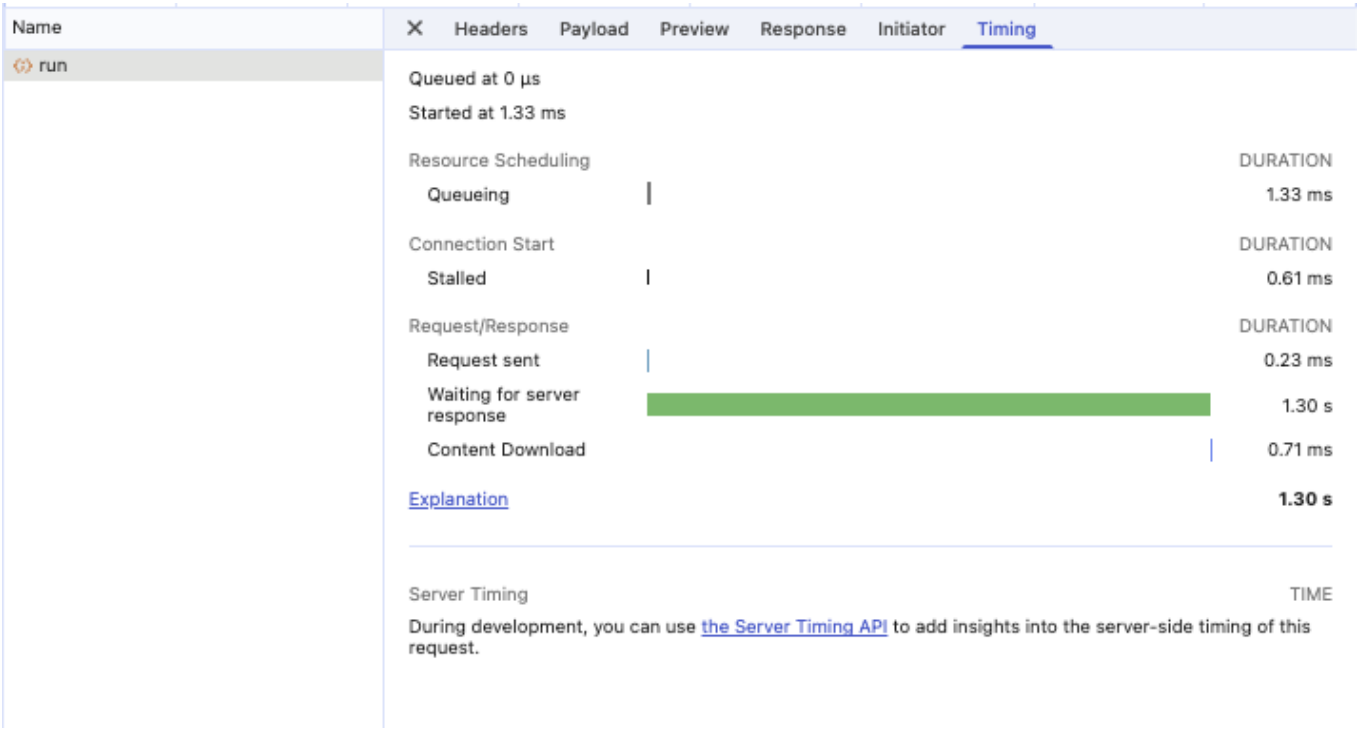

Рисунок 3.11 — Моніторинг часу відповіді сервісу виконання коду

Особлива увага приділялася перевірці нормалізації виводу. Для тестування підготовлено програми, що виводять результат із різними варіантами завершення рядка («\n», «\r\n», без завершального переносу), а також з пробілами на початку та у кінці. Усі ці варіанти після нормалізації функцією normalize коректно зіставляються з очікуваним виводом, що підтверджує вірність обраної стратегії порівняння.

Юзабіліті-тестування проведено за участі п'яти студентів спеціальності 122 «Комп'ютерні науки» ПУЕТ, які раніше не мали досвіду роботи з платформою. Кожному учаснику було поставлено завдання: зайти на платформу, опрацювати перші три модулі (теорію, всі вправи і тести) та поділитися враженнями. За результатами тестування зроблено такі висновки: інтерфейс інтуїтивно зрозумілий — усі учасники без додаткових інструкцій знайшли потрібні розділи; редактор коду сприйнятий позитивно завдяки звичному вигляду на основі Monaco; функція

«Підказка» оцінена як корисна, особливо для вправ типу «Напиши з нуля»; функція «Показати рішення» викликала зауваження про можливість «жульництва», тому надалі реалізовано поведінку, при якій після перегляду рішення відповідна вправа не позначається як виконана. Усі п'ятеро учасників успішно довели роботу до кінця і подали позитивну оцінку загальної зручності інтерфейсу.

Крос-браузерне тестування виконано на чотирьох найпопулярніших браузерах настільної операційної системи: Google Chrome 124, Mozilla Firefox 125, Microsoft Edge 124 та Safari 17. У всіх протестованих браузерах інтерфейс відображається коректно, виконуються всі ключові сценарії, локальне сховище працює без особливостей. Окремо перевірено мобільні браузери Chrome для Android та Safari для iOS — усі модулі, тести і фінальний екзамен працюють у мобільному форм-факторі, редактор Monaco адаптується до маленького екрану через властивість automaticLayout.

Адаптивне тестування проведено на трьох контрольних розмірах viewport: 1440×900 (desktop), 768×1024 (tablet) та 375×667 (mobile). На розмірі desktop інтерфейс розкривається у трьохколонкові сітки на лендингу та переліку модулів. На розмірі tablet — переходить у двоколонкове відображення з коректним зменшенням розмірів шрифтів. На мобільному розмірі усі сітки колапсують у однойменну колонку, у навігаційному меню приховуються текстові мітки, кнопки набувають повної ширини. Результати адаптивного тестування підтверджують, що інтерфейс зберігає функціональність і читабельність на всіх протестованих розмірах.

Перевірка персистентного зберігання прогресу проведена за таким сценарієм: користувач проходить кілька вправ і тест у Модулі 1; перезавантажує сторінку; перевіряє, що прогрес відновлюється у такому самому стані. Окремо перевірено, що при очищенні даних сайту через налаштування браузера або через вбудовану кнопку «Скинути прогрес» стан повертається до початкового. Тестування міграції схеми проведено вручну: у localStorage записувалися дані старої версії 1 (без полів exerciseStats та exam), після чого відкривався застосунок — функція migrate автоматично доповнила відсутні поля без втрати наявних даних.

У підсумку проведене тестування підтвердило, що розроблена платформа відповідає сформульованим у постановці задачі функціональним і нефункціональним вимогам, коректно взаємодіє з зовнішнім сервісом виконання коду, забезпечує надійне збереження прогресу та зручний користувацький досвід на широкому спектрі пристроїв і браузерів.

3.8. Інструкція для користувача

Цей підрозділ містить покрокові інструкції для здобувача освіти, який починає роботу з навчальною платформою «JavaExceptions». Інструкція охоплює системні вимоги, перший вхід на платформу, навігацію по основних розділах, виконання типових сценаріїв (опрацювання теорії, виконання вправи, проходження тесту, складання фінального екзамену) та користування допоміжними інструментами (ігровий майданчик, глосарій).

Системні вимоги до робочої станції користувача мінімальні: будь-який сучасний веб-браузер (Chrome 100+, Firefox 100+, Edge 100+, Safari 16+) на персональному комп'ютері або мобільному пристрої; постійне підключення до мережі Інтернет (потрібне для запуску Java-коду через сервіс Piston та для початкового завантаження сторінки); увімкнена підтримка JavaScript; дозвіл на використання локального сховища для збереження прогресу між сесіями. Жодного додаткового програмного забезпечення (JDK, IDE, плагінів) встановлювати не потрібно.

Перший вхід на платформу. Користувач відкриває у браузері адресу розгорнутого застосунку (за замовчуванням http://localhost:3000 при локальному запуску). На лендінговій сторінці у верхній частині розташовано вітальний заголовок з градієнтним виділенням слова «Java», коротке формулювання призначення платформи, дві кнопки переходу — «Почати навчання» та «Переглянути прогрес» — і блок зведеної статистики курсу. Якщо це перше відвідування, кнопка переходу матиме напис «Почати навчання»; за наявності

збереженого прогресу — «Продовжити навчання». Натискання головної кнопки переводить користувача до переліку модулів.

Навігація по платформі здійснюється через верхню панель, що містить такі пункти: «Модулі» — перехід до переліку всіх одинадцяти модулів курсу; «Playground» — перехід до ігрового майданчика для вільного експериментування з кодом; «Глосарій» — перехід до словника термінів; «Іспит» — перехід до фінального екзамену; «Прогрес» — перехід до сторінки статистики. На широких екранах поряд з піктограмами відображаються текстові мітки, на мобільних — лише піктограми для збереження компактності.

Опрацювання модуля. На сторінці переліку модулів обирається картка цікавлячого модуля (наприклад, «Модуль 1. Ієрархія винятків у Java»). На сторінці модуля розгорнуто структурований теоретичний матеріал з блоками коду, примітками та діаграмами. Після ознайомлення з теорією, у нижній частині сторінки користувач натискає кнопку «Перейти до вправ», що веде на окрему сторінку модулі-вправ. Поточний прогрес «теорію прочитано» автоматично фіксується у локальному сховищі при переході.

Виконання практичної вправи. На сторінці вправ відкривається перша з чотирьох вправ модуля. У верхній частині розташовано бейджі з номером вправи (вправа 1 з 4), типом вправи («Доповни код», «Виправ помилку», «Передбач вивід» або «Напиши з нуля») та лічильником зроблених спроб. Нижче подано опис завдання, після чого розташовано редактор Monaco з початковим кодом вправи. У редакторі підтримуються звичні комбінації клавіш: Ctrl+S — без дії (зберігати окремо не потрібно), Ctrl+Z/Ctrl+Y — скасувати/повторити, Ctrl+/ — закоментувати рядок, Ctrl+F — пошук у коді.

Над редактором розташовано три допоміжні кнопки: «Stdin» — відкриває блок введення зі стандартного потоку (потрібен для вправ, що використовують Scanner або BufferedReader); «Підказка» — показує короткий натяк, як підійти до розв'язку; «Скинути» — повертає код до початкового стану вправи. Після того, як користувач написав свій варіант розв'язку, він натискає кнопку «Запустити». Кнопка тимчасово дезактивується, на її місці з'являється індикатор завантаження

«Виконується...» та обертовий кружок. Після виконання у блоці «Вивід програми» з'являється результат: при коректному розв'язку — бейдж «Правильно!» зеленого кольору, при помилці — бейдж «Не збігається з очікуваним» з виведенням stdout та повідомленням компілятора або середовища виконання у stderr.

Якщо вправа не вдається, рекомендується скористатися кнопкою «Підказка» для отримання спрямовуючої поради; уважно перечитати опис завдання та очікуваний вивід. Якщо знайти розв'язок не вдається, можна натиснути кнопку «Показати рішення», що замістить ваш код еталонним розв'язком — однак при цьому вправа не буде зарахована як виконана. Після успішного виконання вправи користувач переходить до наступної через кнопку «Наступна вправа» або повертається до сторінки модуля.

Проходження тесту модуля. Після завершення усіх чотирьох вправ модуля стає активною кнопка «Перейти до тесту». Тест містить вісім питань множинного вибору з чотирма варіантами відповіді кожне. Після обрання варіанта правильна відповідь автоматично підсвічується зеленим кольором, неправильна — червоним. Поряд з'являється пояснення обраної відповіді. Кнопка «Наступне питання» переводить користувача до наступного. Після завершення тесту відображається підсумок із кількістю правильних відповідей, відсотком успіху та кільцевим індикатором. Тест можна пройти повторно за кнопкою «Пройти знову».

Перегляд прогресу. На сторінці «Прогрес» розгорнуто загальний показник просування курсом у вигляді кільцевої діаграми, агреговані метрики (загальна кількість спроб, кількість помилок різних типів) та індивідуальні картки кожного модуля з прогресом. На сторінці присутня кнопка «Скинути прогрес», що після підтвердження повністю очищає збережений стан і повертає користувача до початкового.

Фінальний екзамен. Після завершення всіх одинадцяти модулів стає рекомендованим складання фінального екзамену. Екзамен містить двадцять випадково обраних питань з єдиного пулу всіх 88 тестових питань курсу та обмежений у часі — на проходження надається 20 хвилин, які відлічуються видимим таймером у верхній частині екрана. Користувач може переходити між

питаннями у довільному порядку через навігаційну сітку та повертатися до пропущених. На відміну від тренувальних тестів, у режимі екзамену пояснення не виводяться одразу — підсумок з усіма поясненнями з'являється лише після завершення всіх питань або після вичерпання часу. Прохідним вважається результат від 70 % правильних відповідей. Найкращий результат екзамену зберігається у статистиці прогресу.

Допоміжні інструменти. Сторінка «Playground» надає вільний редактор коду без перевірки результату — корисний для самостійного експериментування з мовою Java і дослідження поведінки конструкцій обробки винятків. Сторінка «Глосарій» містить алфавітний словник термінів предметної області з пошуком та прив'язкою кожного запису до відповідного модуля курсу. Тема оформлення (світла або темна) автоматично визначається на основі системних налаштувань операційної системи користувача — окрема кнопка перемикання у платформі не передбачена, оскільки це дозволяє зберігати єдиний користувацький досвід між системою і застосунком.

Розв'язання типових проблем. Якщо запуск коду повертає повідомлення про мережеву помилку, слід перевірити підключення до Інтернету: сервіс Piston потребує мережевого доступу. Якщо прогрес не зберігається між сесіями, слід переконатися, що у браузері увімкнене локальне сховище для відповідного сайту. У випадку очевидної помилки інтерфейсу можна оновити сторінку клавішею F5 — це не призведе до втрати збереженого прогресу, оскільки він зберігається у локальному сховищі браузера.

Подана інструкція описує стандартний сценарій роботи з платформою «JavaExceptions» і дозволяє здобувачу освіти самостійно опанувати весь курс з обробки виняткових ситуацій у Java без додаткових пояснень з боку викладача.

ВИСНОВКИ

У межах кваліфікаційної роботи розроблено інтерактивну веб-платформу «JavaExceptions», призначену для навчання обробки виняткових ситуацій у мові програмування Java у курсі дисципліни «Об'єктно-орієнтоване програмування». Поставлену мету повністю досягнуто: створено повноцінне навчальне програмне забезпечення, що поєднує структурований теоретичний матеріал, практичні вправи з виконанням Java-коду у браузері та автоматизовану перевірку знань через тести і фінальний екзамен.

У ході виконання роботи розв'язано всі поставлені задачі та отримано такі результати:

- проаналізовано предметну область навчання обробки винятків у Java, систематизовано основні концепції механізму та обґрунтовано важливість інтерактивних форм навчання у сучасному закладі вищої освіти;
- здійснено огляд п'яти найпопулярніших аналогів (Codecademy, JetBrains Academy, W3Schools, JavaTpoint, Codingame), складено порівняльну таблицю за дев'ятьма критеріями та обґрунтовано доцільність розробки власного програмного продукту;
- розглянуто теоретичні засади ієрархії Throwable, конструкцій try-catch-finally, throw/throws, multi-catch, try-with-resources, ланцюжків винятків та власних класів винятків;
- обґрунтовано вибір технологічного стеку: Next.js 16 з App Router, TypeScript, Tailwind CSS 4, власні UI-примітиви на базі class-variance-authority, Zustand з персистентним зберіганням у localStorage, Monaco Editor як редактор коду та публічне API Piston як сервіс віддаленого виконання Java-коду;
- спроєктовано архітектуру навчальної платформи з поділом на серверні та клієнтські React-компоненти, побудовано діаграму використання, файлову структуру та концептуальну модель даних;
- розроблено блок-схеми загального алгоритму роботи системи та детального

алгоритму виконання практичної вправи з обробкою всіх типів помилок;

- підготовлено навчальний контент обсягом одинадцяти тематичних модулів — від базових концепцій до продакшн-практик і підсумкового банківського проєкту, — що містять понад 330 теоретичних блоків, 47 практичних вправ з еталонними розв'язками та 88 тестових питань з поясненнями;
- реалізовано клієнтську частину веб-додатку з адаптивним інтерфейсом, підтримкою світлої та темної теми та власною системою компонентів інтерфейсу;
- інтегровано редактор Monaco з підтримкою синтаксису Java та сервіс Piston через серверний API-маршрут /api/run з підтримкою введення зі стандартного потоку;
- реалізовано модуль практичних вправ з чотирма типами завдань і автоматичною перевіркою результатів через нормалізацію та порівняння stdout;
- реалізовано модуль тестування з підтримкою питань множинного вибору, негайним зворотним зв'язком, поясненнями та обчисленням відсотка успіху;
- реалізовано модуль фінального екзамену з випадковим відбором двадцяти питань з єдиного пулу 88 тестових питань курсу, обмеженим у часі (20 хвилин) проходженням і підсумковим звітом;
- реалізовано систему збереження прогресу через мідлвар persist бібліотеки Zustand з підтримкою версіонування та міграції схеми;
- проведено функціональне, інтеграційне, юзабіліті, крос-браузерне та адаптивне тестування платформи з участю п'яти студентів спеціальності 122 «Комп'ютерні науки»;
- підготовлено детальну інструкцію для користувача, що охоплює усі сценарії роботи з платформою.

Розроблений програмний продукт повністю відповідає сформульованим у постановці задачі функціональним і нефункціональним вимогам. Архітектурно платформа реалізована як гібридний веб-додаток з серверним рендерингом, мінімальним обсягом клієнтського JavaScript та зовнішньою інтеграцією виключно

з безкоштовним публічним API виконання коду. Така архітектура усуває необхідність у власній серверній інфраструктурі, базі даних і системі автентифікації, що особливо важливо для дипломних і навчально-методичних проєктів.

Практичне значення роботи полягає у можливості безпосереднього використання розробленої платформи у навчальному процесі ПУЕТ та інших закладів вищої освіти під час викладання дисциплін, пов'язаних з об'єктно-орієнтованим програмуванням на Java. Платформа може застосовуватися як основний навчальний інструмент під час дистанційного навчання, як допоміжний матеріал для самостійної роботи студентів або як інструмент об'єктивної перевірки знань через систему тестів та фінальний екзамен.

Перспективи подальшого розвитку розробленого продукту включають: розширення курсу новими тематичними модулями (узагальнення, потоки, лямбда-вирази, функціональні інтерфейси, колекції); додавання режиму викладача з можливістю агрегованого моніторингу прогресу групи студентів; інтеграцію з системами дистанційного навчання Moodle або Google Classroom за допомогою стандарту LTI; підтримку інших мов програмування у складі курсу обробки винятків; розробку мобільного застосунку на базі React Native; реалізацію підтримки спільного режиму, у якому викладач та студент бачать однаковий стан коду у редакторі.

Виконана кваліфікаційна робота продемонструвала здатність здобувача освіти самостійно реалізовувати повний цикл розробки сучасного веб-додатку: від аналізу предметної області та проєктування архітектури до програмної реалізації, тестування і документування продукту. Засвоєні компетенції відповідають кваліфікаційним вимогам спеціальності 122 «Комп'ютерні науки» та можуть бути застосовані у подальшій професійній діяльності у галузі веб-розробки та освітніх технологій.

СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. Oracle. The Java Tutorials. Lesson: Exceptions. URL: https://docs.oracle.com/javase/tutorial/essential/exceptions/
2. Oracle. Java Platform, Standard Edition Documentation. URL: https://docs.oracle.com/en/java/javase/21/docs/api/index.html
3. Wirfs-Brock R., Yoder J., Roberts D. Patterns for Exception Handling. Communications of the ACM. 2022. Vol. 65, No. 3. URL: https://dl.acm.org/doi/10.1145/3490099
4. Goetz B., Bowbeer J., Bloch J. Java Concurrency and Exception Patterns: Modern Best Practices. ACM Queue. 2023. Vol. 21. URL: https://queue.acm.org/detail.cfm?id=3578265
5. Freeman A. Active Learning in Computing Education: Evidence and Practice. Education and Information Technologies. 2022. Vol. 27. URL: https://link.springer.com/journal/10639
6. Codecademy. Learn Java Course. URL: https://www.codecademy.com/learn/learn-java
7. JetBrains. JetBrains Academy. Java Developer Track. URL: https://www.jetbrains.com/academy/
8. W3Schools. Java Exceptions Tutorial. URL: https://www.w3schools.com/java/java_try_catch.asp
9. JavaTpoint. Exception Handling in Java. URL: https://www.javatpoint.com/exception-handling-in-java
10. Codingame. Practice and Learn Programming Through Games. URL: https://www.codingame.com/
11. Vercel. Next.js Documentation. App Router and Server Components. URL: https://nextjs.org/docs
12. Meta. React Documentation. URL: https://react.dev/

13. EngineerMan. Piston: a high performance general purpose code execution engine. URL: https://github.com/engineer-man/piston
14. Microsoft. Monaco Editor Documentation. URL: https://microsoft.github.io/monaco-editor/
15. Tailwind Labs. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs
16. Pmnd.rs. Zustand: a small, fast and scalable state-management solution. URL: https://github.com/pmndrs/zustand
17. Joachimiak D. class-variance-authority: variant API for designing component variants. URL: https://cva.style/docs
18. Ольховська О. В. Методичні рекомендації до виконання, оформлення та захисту кваліфікаційних робіт здобувачами вищої освіти спеціальності 122 «Комп'ютерні науки». Полтава : ПУЕТ, 2024. 67 с.
19.
20.
21.
22.

ДОДАТОК А

А.1 — Інтерфейси типів навчального контенту (файл lib/curriculum/types.ts).

```
export type ExerciseType = "fill-in" | "fix-bug" | "predict-output" |
"write-from-scratch";

export interface Exercise {
id: string;
type: ExerciseType;
title: string;
description: string;
starterCode: string;
expectedOutput: string;
hint: string;
solution: string;
stdin?: string;
}

export interface QuizQuestion {
id: string;
question: string;
options: [string, string, string, string];
correctIndex: 0 | 1 | 2 | 3;
explanation: string;
}

export interface TheoryBlock {
type: "text" | "code" | "note" | "warning" | "diagram";
content: string;
language?: string;
}

export interface Module {
id: number;
```

```
slug: string;
title: string;
shortDescription: string;
theory: TheoryBlock[];
exercises: Exercise[];
quiz: QuizQuestion[];
}
```

А.1 містить чотири основні інтерфейси, що формують концептуальну модель навчального контенту: тип Exercise описує структуру практичної вправи; QuizQuestion — структуру тестового питання; TheoryBlock — окремий блок теоретичного матеріалу; Module — навчальний модуль як композицію всіх перерахованих елементів. Така типізація забезпечує консистентність даних у клієнтській частині та контроль на етапі компіляції.

А.2 — Серверний API-маршрут запуску Java-коду (файл app/api/run/route.ts).

```
import { NextResponse } from "next/server";

export const runtime = "nodejs";

const PISTON_URL = process.env.PISTON_URL || "http://localhost:2000";

export async function POST(req: Request) {
try {
const { code, stdin } = await req.json();
if (typeof code !== "string") {
return NextResponse.json({ error: "Missing code" }, { status: 400 });
}

const res = await fetch(`${PISTON_URL}/api/v2/execute`, {
method: "POST",
headers: { "Content-Type": "application/json; charset=utf-8" },
body: JSON.stringify({
language: "java",
version: "15.0.2",
files: [{ name: "Main.java", content: code, encoding: "utf8" }],
```

```
stdin: typeof stdin === "string" ? stdin : "",
}),
});

if (!res.ok) {
const text = await res.text();
return NextResponse.json(
{ error: `Piston returned ${res.status}: ${text}` },
{ status: 502 }
);
}

const data = await res.json();
return NextResponse.json({
stdout: data.run?.stdout ?? "",
stderr: data.run?.stderr ?? "",
code: data.run?.code ?? null,
});
} catch (err) {
return NextResponse.json(
{ error: err instanceof Error ? err.message : String(err) },
{ status: 500 }
);
}
}
```

У А.2 представлено серверний обробник POST-запитів. Маршрут зчитує тіло запиту, перевіряє наявність поля code, проксує запит до публічного API Piston з параметрами language="java" та version="15.0.2", після чого повертає клієнту спрощену структуру з результатом виконання. Усі мережеві помилки обгортаються у блок try/catch, що гарантує коректну відповідь з відповідним HTTP-статусом.

А.3 — Клієнтський хелпер виклику серверного маршруту (файл lib/piston.ts).

```
export interface PistonResult {
stdout: string;
stderr: string;
```

```
code: number | null;
}

export async function runJavaCode(code: string, stdin: string = ""):
Promise<PistonResult> {
const response = await fetch("/api/run", {
method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify({ code, stdin }),
});

const data = await response.json();

if (!response.ok) {
throw new Error(data.error || `HTTP ${response.status}`);
}

return {
stdout: data.stdout ?? "",
stderr: data.stderr ?? "",
code: data.code ?? null,
};
}
```

У А.3 наведено клієнтську функцію runJavaCode, яка інкапсулює логіку звернення до серверного маршруту /api/run. Функція повертає Promise<PistonResult> із полями stdout, stderr та code. Уся складність мережевих деталей прихована від компонентів, що використовують цю функцію.

А.4 — Компонент-обгортка редактора Monaco (файл components/CodeEditor.tsx).

```
"use client";

import Editor from "@monaco-editor/react";

interface CodeEditorProps {
```

```
value: string;
onChange: (value: string) => void;
height?: string;
readOnly?: boolean;
}

export function CodeEditor({ value, onChange, height = "320px",
readOnly }: CodeEditorProps) {
return (
<div className="overflow-hidden rounded-lg border">
<Editor
height={height}
defaultLanguage="java"
value={value}
onChange={(v) => onChange(v ?? "")}
theme="vs-dark"
options={{
minimap: { enabled: false },
fontSize: 14,
lineNumbers: "on",
scrollBeyondLastLine: false,
automaticLayout: true,
tabSize: 4,
readOnly,
padding: { top: 12 },
}}
/>
</div>
);
}
```

У А.4 показано тонку обгортку CodeEditor над компонентом Editor пакета @monaco-editor/react. Обгортка приймає три обов'язкові пропси (value, onChange) та два опціональні (height, readOnly), сталими залишаючи всі інші налаштування редактора: темну тему vs-dark, шрифт 14 пунктів, нумерацію рядків, табуляцію 4 пробіли, вимкнену мінімапу та автоматичне підлаштування розмірів.

А.5 — Картка модуля з індикатором прогресу (файл components/ModuleCard.tsx).

```
"use client";

import Link from "next/link";
import { CheckCircle2, Circle, BookOpen, Code, HelpCircle } from
"lucide-react";
import { Card, CardContent, CardHeader, CardTitle } from
"@/components/ui/card";
import { Badge } from "@/components/ui/badge";
import { useProgressStore, useHydrated, EMPTY_MODULE_PROGRESS } from
"@/lib/progress";
import type { Module } from "@/lib/curriculum";
import { ProgressRing } from "./ProgressRing";

interface ModuleCardProps {
module: Module;
}

export function ModuleCard({ module }: ModuleCardProps) {
const hydrated = useHydrated();
const stored = useProgressStore((s) => s.modules[module.id]);
const progress = hydrated && stored ? stored : EMPTY_MODULE_PROGRESS;

const totalSteps = 3;
let completed = 0;
if (progress.theoryRead) completed++;
if (progress.exercisesCompleted.length >= module.exercises.length)
completed++;
if (progress.quizCompleted) completed++;
const percent = (completed / totalSteps) * 100;

const status =
completed === 0 ? "Не розпочато" : completed === totalSteps ?
"Завершено" : "В процесі";
```

```
return (
<Link href={`/modules/${module.id}`}>
<Card className="group h-full cursor-pointer transition-all
hover:border-indigo-400 hover:shadow-lg">
<CardHeader>
<div className="flex items-start justify-between gap-4">
<div className="flex-1">
<Badge variant="outline">{status}</Badge>
<CardTitle className="text-lg leading-tight group-hover:text-indigo-
600">
{module.title}
</CardTitle>
</div>
<ProgressRing value={percent} size={52} strokeWidth={5} />
</div>
</CardHeader>
<CardContent>
<p className="mb-4 text-sm text-muted-
foreground">{module.shortDescription}</p>
</CardContent>
</Card>
</Link>
);
}
```

А.5 ілюструє типовий клієнтський компонент платформи. ModuleCard читає стан модуля з глобального сховища, обчислює відсоток прогресу за трьома активностями (теорія, вправи, тест) і рендерить картку з заголовком, бейджем статусу та кільцевим індикатором ProgressRing. Хук useHydrated використовується для уникнення помилок гідратації між серверним і клієнтським рендерингом.

А.6 — Компонент-контролер виконання вправи (файл components/ExerciseRunner.tsx, фрагмент).

```
"use client";
```

```
import { useState } from "react";
import { Play, CheckCircle2, XCircle, Lightbulb, RotateCcw, Eye,
Loader2, Terminal } from "lucide-react";
import { Button } from "@/components/ui/button";
import { Badge } from "@/components/ui/badge";
import { CodeEditor } from "./CodeEditor";
import { runJavaCode } from "@/lib/piston";
import type { Exercise } from "@/lib/curriculum";

export type AttemptResult = "success" | "compile-error" | "runtime-
error" | "wrong-output";

interface ExerciseRunnerProps {
exercise: Exercise;
exerciseNumber: number;
totalExercises: number;
attempts?: number;
onComplete: () => void;
onAttempt?: (result: AttemptResult) => void;
}

const EXERCISE_TYPE_LABELS: Record<Exercise["type"], string> = {
"fill-in": "Доповни код",
"fix-bug": "Виправ помилку",
"predict-output": "Передбач вивід",
"write-from-scratch": "Напиши з нуля",
};

function normalize(s: string): string {
return s.replace(/\r\n/g, "\n").trim();
}

function detectErrorType(stderr: string): "compile-error" | "runtime-
error" {
if (/error:/i.test(stderr) && /\.java:\d+/.test(stderr)) return
"compile-error";
```

```
return "runtime-error";
}

export function ExerciseRunner({
exercise,
exerciseNumber,
totalExercises,
attempts = 0,
onComplete,
onAttempt,
}: ExerciseRunnerProps) {
const [code, setCode] = useState(exercise.starterCode);
const [stdin, setStdin] = useState(exercise.stdin ?? "");
const [output, setOutput] = useState<string>("");
const [isRunning, setIsRunning] = useState(false);
const [isCorrect, setIsCorrect] = useState<boolean | null>(null);
const [showHint, setShowHint] = useState(false);
const [showSolution, setShowSolution] = useState(false);
const [showStdin, setShowStdin] = useState(!!exercise.stdin);

async function handleRun() {
setIsRunning(true);
setOutput("Виконується...");
setIsCorrect(null);
try {
const result = await runJavaCode(code, stdin);
const combined = (result.stdout || "") + (result.stderr ? "\n" +
result.stderr : "");
setOutput(combined || "(немає виводу)");
const correct = normalize(result.stdout) ===
normalize(exercise.expectedOutput);
setIsCorrect(correct);
if (correct) {
onAttempt?.("success");
onComplete();
} else if (result.stderr) {
```

```
onAttempt?.(detectErrorType(result.stderr));
} else {
onAttempt?.("wrong-output");
}
} catch (err) {
setOutput("Помилка запуску: " + (err instanceof Error ? err.message :
String(err)));
setIsCorrect(false);
onAttempt?.("runtime-error");
} finally {
setIsRunning(false);
}
}

function handleReset() {
setCode(exercise.starterCode);
setStdin(exercise.stdin ?? "");
setOutput("");
setIsCorrect(null);
setShowSolution(false);
}

function handleShowSolution() {
setCode(exercise.solution);
setShowSolution(true);
}
}
```

У А.6 наведено компонент-контролер ExerciseRunner, що поєднує редактор Monaco, керування станом виконання, асинхронний виклик сервісу Piston і валідацію результату. Метод handleRun реалізує повний цикл: переведення у стан виконання, асинхронний виклик runJavaCode, нормалізацію виводу, порівняння з очікуваним результатом і класифікацію типу помилки через функцію detectErrorType.

А.7 — Глобальне сховище прогресу на Zustand (файл lib/progress.ts, фрагмент).

```
"use client";

import { useEffect, useState } from "react";
import { create } from "zustand";
import { persist } from "zustand/middleware";

export type AttemptErrorType = "success" | "compile-error" | "runtime-
error" | "wrong-output";

export interface ExerciseStats {
attempts: number;
errors: { compile: number; runtime: number; wrongOutput: number; };
firstSuccessAt: number | null;
}

export interface ModuleProgress {
theoryRead: boolean;
exercisesCompleted: number[];
exerciseStats: Record<number, ExerciseStats>;
quizScore: number | null;
quizCompleted: boolean;
}

export interface ExamProgress {
bestScore: number | null;
bestTotal: number | null;
attempts: number;
lastCompletedAt: number | null;
}

export const useProgressStore = create<ProgressStore>()(
persist(
(set, get) => ({
```

```
modules: {},
exam: defaultExamProgress(),

markTheoryRead: (moduleId) =>
set((state) => ({
modules: {
...state.modules,
[moduleId]: {
...defaultModuleProgress(),
...state.modules[moduleId],
theoryRead: true,
},
},
})),

markExerciseCompleted: (moduleId, exerciseIndex) =>
set((state) => {
const current = state.modules[moduleId] ?? defaultModuleProgress();
const already = current.exercisesCompleted.includes(exerciseIndex);
return {
modules: {
...state.modules,
[moduleId]: {
...current,
exercisesCompleted: already
? current.exercisesCompleted
: [...current.exercisesCompleted, exerciseIndex],
},
},
};
}),

recordExerciseAttempt: (moduleId, exerciseIndex, result) =>
set((state) => {
const current = state.modules[moduleId] ?? defaultModuleProgress();
const prev = current.exerciseStats[exerciseIndex] ?? emptyStats();
```

```
const next: ExerciseStats = {
attempts: prev.attempts + 1,
errors: {
compile: prev.errors.compile + (result === "compile-error" ? 1 : 0),
runtime: prev.errors.runtime + (result === "runtime-error" ? 1 : 0),
wrongOutput: prev.errors.wrongOutput + (result === "wrong-output" ? 1
: 0),
},
firstSuccessAt:
prev.firstSuccessAt ??
(result === "success" ? prev.attempts + 1 : null),
};
return {
modules: {
...state.modules,
[moduleId]: {
...current,
exerciseStats: { ...current.exerciseStats, [exerciseIndex]: next },
},
},
};
}),

setQuizScore: (moduleId, score, _total, completed) =>
set((state) => ({
modules: {
...state.modules,
[moduleId]: {
...defaultModuleProgress(),
...state.modules[moduleId],
quizScore: score,
quizCompleted: completed,
},
},
})),
```

```
recordExamResult: (score, total) =>
set((state) => {
const current = state.exam;
const isBest = current.bestScore === null || score >
current.bestScore;
return {
exam: {
bestScore: isBest ? score : current.bestScore,
bestTotal: isBest ? total : current.bestTotal,
attempts: current.attempts + 1,
lastCompletedAt: Date.now(),
},
};
}),

getTotalProgress: () => {
const { modules } = get();
const totalModules = 8;
let completed = 0;
for (let i = 1; i <= totalModules; i++) {
const m = modules[i];
if (!m) continue;
if (m.theoryRead) completed++;
if (m.exercisesCompleted.length >= 4) completed++;
if (m.quizCompleted) completed++;
}
return Math.round((completed / (totalModules * 3)) * 100);
},
}),
{
name: "java-exceptions-progress",
version: 2,
}
)
);
```

```
export function useHydrated(): boolean {
const [hydrated, setHydrated] = useState(false);
useEffect(() => {
setHydrated(useProgressStore.persist.hasHydrated());
const unsub = useProgressStore.persist.onFinishHydration(() =>
setHydrated(true));
return unsub;
}, []);
return hydrated;
}
```

У А.7 наведено реалізацію глобального сховища прогресу на бібліотеці Zustand. Сховище інкапсулює модель стану (modules та exam), методи модифікації (markTheoryRead, markExerciseCompleted, recordExerciseAttempt, setQuizScore, recordExamResult), агрегаційну функцію getTotalProgress та налаштування мідлвару persist для збереження у localStorage під ключем «java-exceptions-progress». Хук useHydrated повертає прапорець завершення гідратації, що використовується клієнтськими компонентами для уникнення конфліктів між серверним і клієнтським рендерингом у Next.js.