Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

Розробка онлайн середовища розробки мовою Java

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення”

Керівник курсової роботи
к.т.н., доц. Олецький О.В.

____
(підпис)
“____” __________ 2020 р.

Виконав студент Лелюк О.О.
“____” __________ 2020 р.

Київ 2020

Зміст

Вступ 5
Анотація 6
1. Аналіз предметної області та наявних проблем і задач 7
1.1 Процес створення веб-застосунку 7
1.2 Проблеми, що виникають при створенні веб-застосунку 9
2. Теоретичні засади розробки веб-застосунків 10
2.1 Поняття клієнт-серверної архітектури 10
2.2 Типи клієнт-серверної архітектури 12
2.2.1 Дворівневий тип 12
2.2.2 Трирівневий тип 12
2.2.3 N-рівневий тип 13
2.3 Поняття веб-застосунок та веб-сервіс 14
2.4 Протокол SOAP 15
2.5 Протокол REST 16
2.6 Протокол RPC 17
3. Реалізація практичної частини 18
3.1 Опис застосунку 18
3.2 Обґрунтування обраних інструментів та структури 19
3.2 Опис розробки програми 22
3.2.1 Клієнтська частина 22
3.2.2 Серверна частина 24
3.3 Компіляція та запуск коду 29
3.4 Опис файлів даних та інтерфейсу програми 30
Висновки 34
Список джерел 35
Додаток А 37
Додаток Б 38

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних систем,

доцент, к.ф.-м.н.

____________ О. П. Жежерун

(підпис)

"____" _______________ 2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Лелюку Олександру

3-го курсу факультету інформатики

ТЕМА: Розробка онлайн середовища розробки мовою Java

Вихідні дані:

- Веб-додаток, який надає можливість користувачам зберігати, редагувати та виконувати вихідний код написаний мовою Java.

Зміст ТЧ до курсової роботи:

Вступ
Анотація
1. Аналіз предметної області та наявних проблем і задач
2. Теоретичні засади розробки веб-додатків
3. Реалізація практичної частини
Висновки
Список джерел
Додатки (за необхідністю)

Дата видачі "___" _________ 201_ р.

Керівник _______________ Завдання отримано __________

Календарний план виконання курсової роботи

Тема: Розробка онлайн середовища розробки Java

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	жовтень 2018 р.	
2.	Огляд літератури за темою роботи	грудень 2019р. – лютий 2020р.	
3.	Огляд технологій потрібних для реалізації практичної частини	лютий – березень 2020р.	
4.	Створення практичн ої частини роботи	березень – травень 2020р.	
5.	Створення текстової частини роботи	квітень – травень 2020р.	
6.	Надання роботи керівнику для перевірки	травень 2020р.	
7.	Корегування роботи за результатами перевірки керівником	травень 2020р.	
8.	Створення презентації	травень 2020р.	
9.	Захист курсової роботи		

Студент ________Лелюк О.О________ ________

Керівник ________Олецький О.В.________ ________

“_____” ________________ ________ р.

Вступ

Для того, щоб розробляти програмне забезпечення мовою програмування Java, програмісту потрібно мати встановлений JDK (Java Developer Kit) та IDE (інтегровану середу розробки). У людей, які тільки починають своє знайомство з програмуванням, у процесі встановлення цих компонентів виникають труднощі через відсутність досвіду та розуміння певних нюансів під час встановлення компонентів. До того ж, сучасні середовища розробки потребують великої обчислювальної потужності комп'ютера. Під час розробки масштабних проектів ці потужності є справді затребуваними, проте вони не потрібні для навчальних проектів.

Ці обидві проблеми будуть вирішені, якщо середовище розробки буде створено у вигляді веб-застосунку, який матиме всі потрібні налаштування і забезпечуватиме весь потрібний функціонал. Тоді комп'ютер користувача буде тільки відображати інтерфейс застосунку й не витрачатиме зайвих ресурсів.

Тому цю роботу присвячено вивченню процесу розробки веб-застосунку, який забезпечує функціональність онлайн середовища розробки мовою Java.

Анотація

Роботу присвячено вивченню процесу розробки веб-сервісу мовою Java за допомогою фреймворків Spring та Angular. На основі результатів отриманих з дослідження було створено веб-сервіс, який являє собою онлайн середовище розробки мовою Java.

1. Аналіз предметної області та наявних проблем і задач

1.1 Процес створення веб-застосунку

Веб-застосунки є дуже важливою частиною сучасного світу. Майже все, з чим ми зустрічаємось у сфері інформаційних технологій є веб-застосунком.

Більшість веб-застосунків поділяють на три частини:

1. Користувацька частина – частина, яку бачить користувач та за допомогою якої обмінюється інформацією з застосунком і отримує послуги.
2. Бізнес частина – частина, яка відповідає за надання послуг.
3. Частина даних – частина, яка відповідає за збереження даних, які потрібні для коректної роботи застосунку.

Користувацька частина може надаватися у вигляді HTML для відображення у браузері, а може й бути звичайним застосунком, який відображає операційна система.

Процес створення веб-застосунку складається з таких етапів [1],[2]:

1. Етап визначення головних цілей та призначення.
2. Етап визначення майбутніх користувачів. Використовуючи ці дані треба особливо звернути увагу на безпеку збереження даних у сервісі та можливість користувачів доступу до сервісу.
3. Етап формування функціонального документу, в якому обумовлюються функціональні можливості та технічні уточнення майбутнього сервісу. Варто зазначити, що веб-застосунок може плануватися як частина вже існуючої сервісно-орієнтованої архітектури. У такому випадку у функціональному документі додатково визначається протокол обміну інформації в цій архітектурі.
4. Етап визначення постачальників послуг, потрібних для коректної роботи веб-сервісу

5. Етап визначення технологій, які будуть використовуватись для реалізації сервісу.
6. Етап визначення користувацького інтерфейсу, якщо він потрібен.
7. Етап розробки. На цьому етапі розробники створюють архітектуру сервісу, структуру бази даних та реалізують функціонал, який було визначено у функціональному документі.
8. Етап тестування.

Після виконання цих етапів сервіс стає готовим до розгортання.

1.2 Проблеми, що виникають при створенні веб-застосунку

При створенні веб застосунку розробники зіштовхуються з багатьма викликами, найбільш поширеними з яких є:

- Веб-застосунок повинен мати зручний інтерфейс. Якщо веб-застосунок має користувацький інтерфейс, то дуже важливо, щоб робота цього інтерфейсу була коректною на будь-якому пристрої.
- Веб-застосунок має легко масштабуватися. Більшість веб-сервісів після розгортання продовжують підтримуватись та вдосконалюватись розробниками. Якщо виникає потреба додати новий функціонал до веб-застосунку, то цей сервіс має бути спроектований таким чином, щоб додавання нового функціоналу не конфліктувало з вже робочим функціоналом.
- Веб-застосунок має бути продуктивним. Веб-застосунки обслуговують багато користувачів, тому вони мають використовувати доступні ресурси виважено. Цю проблему частково вирішують обрані технології, але розробник все одно має звертати увагу на можливі проблеми з продуктивністю застосунку.
- Веб-застосунок повинен мати гарний захист. Веб-застосунки можуть працювати з конфіденційною інформацією, тому дуже важливо надати безпечну передачу та збереження цієї інформації.

2. Теоретичні засади розробки веб-застосунків

2.1 Поняття клієнт-серверної архітектури

Клієнт-серверна архітектура – це обчислювальна або мережева архітектура, в якій завдання або мережеве навантаження розподілені між постачальниками послуг (сервер) і замовниками послуг (клієнт). [4]

API (application programming interface) – набір операцій, за допомогою яких можна обмінюватись інформацією з програмою.

Сервер – це компонент клієнт-серверної архітектури, який має певний API, за допомогою якого клієнти можуть надавати запити до сервера задля отримання деякої інформації або виконання сервером певних дій. Сервери розподіляють за типами. Наприклад, сервери, які надають дані для відображення в браузері називають веб-серверами, сервери, які використовують протокол FTP (file transfer protocol), називають FTP серверами і т.д.

Сервери також розділяють за способом обробки запитів: [3]

- Ітеративний – отримує та обробляє запити по черзі. Сервер отримує запит і оброблює його, а інші запити, які надходять у той самий час, заносяться в чергу.
- Багатопотоковий – сервер має певний набір процесів, які працюють паралельно і використовуються задля того, щоб мати можливість відповідати на декілька запитів одночасно. Вирішує проблеми ітеративного серверу, але є досить складним в реалізації.

Клієнт – це компонент клієнт-серверної архітектури, який надає запити до сервера і обробляє відповіді, які надає сервер.

Отже, обмін інформацією між клієнтом і сервером відбувається таким чином: сервер очікує запити від клієнтів, клієнти відправляють запити, сервер їх оброблює і відправляє відповіді на ці запити, а ці відповіді вже отримують та оброблюють клієнти.

Щоб клієнти і сервери могли належним чином обмінюватись інформацією, використовуються протоколи обміну інформації.

Протокол обміну інформації – це набір правил, які описують шаблон, який буде використовуватись для формування повідомлень у клієнт-серверній комунікації. Прикладами протоколів є HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol) та багато інших.

2.2 Типи клієнт-серверної архітектури

Клієнт-серверну архітектуру поділяють на такі типи: [3]

- Дворівнева
- Трирівнева
- N-рівнева

2.2.1 Дворівневий тип

Архітектуру застосунку розділено на дві частини – клієнтський рівень та рівень даних.

Має такі переваги:

- Процес підтримки є легким завдяки простоті структури застосунку.
- Завдяки тому, що база даних та серверний код знаходяться фізично близько, операції виконуються досить швидко.

Має такі недоліки:

- Продуктивність дуже сильно залежить від кількості користувачів, бо для підтримки цілісності даних рівень даних використовує блокування.
- Набір можливих операцій клієнта повністю залежить від виробника бази даних.
- Складний процес переходу на іншу СУБД (*тут і далі* – система управління базами даних).

Дворівневу архітектуру можна використовувати, коли можлива кількість користувачів є досить невеликою, а функціонал застосунку повністю задовольняється можливостями СУБД, що буде використана.

2.2.2 Трирівневий тип

Архітектуру застосунку поділяють на три частини:

- Клієнтський рівень або рівень представлення. Являє собою частину застосунку, яка відповідає за відображення користувацького інтерфейсу.

- Бізнес рівень. Являє собою частину застосунку, яка відповідає за валідацію, передачу даних на рівень даних, обчислення та інші операції, які не відносяться до інших рівнів.
- Рівень даних. Являє собою частину застосунку, який відповідає за підключення до бази даних та маніпуляцію даних у цій базі.

Такий тип має такі характеристики:

- Покращена продуктивність застосунку.
- Кожний рівень може горизонтально розширюватись.
- Можливість валідувати дані.
- Покращена безпека застосунку, бо користувач тепер не має прямого доступу до рівня даних.
- Застосунок стає більш комплексним і потребує більше зусиль для реалізації.

2.2.3 N-рівневий тип

Архітектура застосунку має змінну кількість рівнів. Застосунки з таким типом архітектури мають дуже гнучку структуру. Коли потрібно внести певні зміни до програми, то створюється новий рівень застосунку. Таким чином переписувати реалізацію не потрібно, але чим більшу кількість рівнів має застосунок, тим більше зростає комплексність застосунку та важче його підтримувати.

2.3 Поняття веб-застосунок та веб-сервіс

Веб-застосунок [3] – це застосунок, який має клієнт-серверну архітектуру, і в якому обмін інформації між клієнтом та сервером відбувається за допомогою веб-браузера.

У випадку веб-застосунку, веб-браузер виступає як універсальний клієнт. Клієнт обмінюється інформацією з сервером в більшості випадків використовуючи протокол HTTP або HTTPS. Інтерфейс користувача у веб-застосунку формується сервером у форматах, які підтримуються веб браузерами. Найпоширенішим з таких форматів є HTML.

Сервісно-орієнтована архітектура [5] – стиль розробки програмного забезпечення, у якому програмне забезпечення ділиться на незалежні один від одного компоненти, кожен з яких надає певні послугу. Ці компоненти називаються сервісами.

Веб-сервіс – незалежний веб-застосунок, який знаходиться в сервісно-орієнтованій архітектурі.

Звичайним веб-застосункам достатньо використовувати HTTP протокол для обміну інформацією з клієнтом, але веб-сервісам цього вже недостатньо, бо їх клієнтами є не тільки веб-браузери, але й інші веб-сервіси в тій самій архітектурі.

Тому, зазвичай, веб-сервіси використовують такі протоколи як HTTP саме для транспортування інформації між клієнтом та сервером, а структуру інформації, якою обмінюється клієнт та сервер описують інші протоколи. Ці протоколи використовують протоколи більш низького рівня для транспортування даних і описують, яку структуру повинна мати інформація, що передається між клієнтом та сервером. Одними з найбільш відомих на даний момент таких протоколів є SOAP, REST та RPC.

2.4 Протокол SOAP

SOAP (Simple Object Access Protocol) [6] – протокол, який використовує XML (eXtensible Markup Language) для кодування запитів та відповідей. Він використовує HTTP для обміну інформації.

SOAP був спроектований таким чином, щоб його можна було легко розширювати і користувач використовував певні частини цього протоколу для певних задач. Через це він став дуже комплексним і більшість можливостей, які він надає, не потрібні звичайному веб-сервісу.

Також побудова запитів та відповідей за протоколом SOAP може бути достатньо складною на деяких мовах програмування, які не мають стандартних інструментів для роботи з XML.

Ці проблеми вирішує інший протокол - REST, а SOAP зараз використовують у веб-сервісах, де потрібна вся комплексність, яку він пропонує.

2.5 Протокол REST

REST (REpresentational State Transfer) [7] – stateless (той, що не має стану) протокол, який на відміну від SOAP надає більшу свободу у виборі структури для обміну даних між клієнтом та сервером. Дані, у випадку REST, можуть бути у форматі XML, JSON, HTML та ін. Для створення запиту використовується URI (Universal Resource Identifier) і, якщо потрібно, додаткові дані в обраному форматі.

Веб-сервіси, що притримуються REST-протоколу, називаються RESTful веб-сервісами. [8]

RESTful веб-сервіси в порівнянні з веб-сервісами, що використовують SOAP:

- Більш продуктивні, бо дані можуть кешуватися та передаються у простому форматі, на аналіз якого витрачається менше ресурсів, ніж в SOAP.
- Мають менше залежностей від інших веб-сервісів завдяки тому, що REST – stateless.
- Легші в розробці, через простоту протоколу.

Завдяки своїм характеристикам REST використовують наразі в більшості сервісно-орієнтованих архітектурах.

2.6 Протокол RPC

RPC (Remote Procedure Call) [9] – у парадигмі клієнт-серверної архітектури є протоколом, який визначає структуру запиту як виклик процедури (методу у випадку об'єктно-орієнтованої мови програмування).

Коли використовується протокол RPC, веб-сервіс віддалено викликає метод іншого сервісу і при цьому на час виконання цього методу блокується або продовжує свою роботу, залежно від реалізації протоколу.

Запити в RPC описують за принципом "Що треба зробити", а в REST, для порівняння, – "Яку інформацію треба отримати або зберегти".

Фактично, REST є спеціальною формою RPC, тому коли не можна використовувати REST, бо обмін інформацією між сервісами занадто комплексний, використовують RPC. До речі, SOAP має функціональність RPC, проте коли комплексність SOAP не потрібна, то просто використовується RPC.

3. Реалізація практичної частини

3.1 Опис застосунку

У результаті аналізу предметної області та дослідження теоретичних відомостей було створено веб-сервіс, який працює за протоколом REST, що виконує роль онлайн середовища розробки Java.

Веб-сервіс написаний мовою Java. Для зменшення кількості шаблонного коду була використана бібліотека Lombok. Для реалізації серверної частини використовувався фреймворк Spring [12], а для реалізації клієнтської частини використовувався фреймворк Angular [11]. Веб-сервіс збирається у jar-файл за допомогою Apache Maven.

Веб-сервіс було розгорнуто на PaaS (Platform as a Service) Heroku. Процес розгортання було автоматизовано завдяки використанню системи контролю версій Git та сервісу GitHub.

Веб-сервіс зберігає дані для аутентифікації користувачів, інформацію про проекти користувачів та структуру проектів з їх вихідним кодом. Для збереження даних, потрібних для аутентифікації та даних про проекти використовувалась СУБД PostgreSQL, яка теж була розгорнута на Heroku. Структура проектів та вихідний код цих проектів зберігається в Amazon S3 (Amazon Simple Storage Service).

3.2 Обґрунтування обраних інструментів та структури

Було вирішено створювати не просто веб-застосунок, а веб-сервіс як частину сервісно-орієнтованої архітектури, оскільки остання забезпечує гнучкість структури, до якої можна легко додати новий функціонал. До того ж, у такому випадку можливе розширення структури з додаванням нових веб-сервісів під час майбутніх розробок.

Було обрано фреймворк Spring, адже він надає повний інструментарій для побудови повноцінного серверної частини веб-сервісу. Були використані такі компоненти цього фреймворку:

- Spring Core – компонент, що надає IoC (Inversion of Control) контейнер та багато інших інструментів, які спрощують розробку застосунків з комплексною архітектурою.
- Spring WEB – компонент, що надає інструментарій для побудови веб-застосунків та сервісів. Використовує технологію сервлетів.
- Spring Security – компонент, що надає інструментарій для налаштування аутентифікації та авторизації користувачів та багатьох інших аспектів пов'язаних із захищеністю веб-сервісу.
- Spring Data – компонент, що надає модель доступу до даних. Він дозволяє з легкістю використовувати технології доступу до даних такі як JDBC (Java Database Connectivity) або ORM (Object Relational Mapping). У даному випадку було використано технологію ORM [13], бо вона зменшує кількість шаблонного коду та прискорює процес розробки.

Також були використані такі інструменти, що належать до Spring:

- Spring Email – надає інструментарій для підключення та використання SMTP-серверу (Simple Mail Transport Protocol).
- Profiles – для запуску веб-застосунку у різних конфігураціях.
- Configuration Processor – для валідації використання конфігурацій проекту.

Фреймворк Angular було обрано, у першу чергу, через те, що він надає широкий інструментарій для побудови візуальної та логічної частини веб-застосунку. Він спрощує розробку комплексної клієнтської архітектури завдяки використанню Typescript [10] та його можливості до статичної типізації. Також, як і Spring, Angular надає можливість робити ін'єкцію залежностей, що ще більше спрощує написання коду.

Клієнтська частина, побудована за допомогою Angular, зазвичай запускається на окремому сервері. Таким чином структура веб-сервісу буде складатися з UI-серверу та серверу, що оброблюватиме запити. Було вирішено, що Angular буде використаний саме для побудови клієнтської частини, а структура веб-сервісу буде складатися з одного серверу, що буде надавати користувацький інтерфейс та відповідатиме на запити від нього. Angular має інструментарій для побудови клієнтської частини у вигляді статичних файлів, але цей процес треба автоматизувати для формування коректного jar-файлу, який буде використаний для розгортання сервісу. Для автоматизації цього процесу було використано Apache Maven. Проект було поділено на три модулі:

- online-java-ide-front – виконує побудову клієнтської частин проекту у вигляді статичних ресурсів.
- online-java-ide-back – виконує побудову серверної частини, що має використовувати статичні ресурси для клієнтської частини.
- online-java-ide – визначає online-java-ide-back та online-java-ide-front як одне ціле. Саме цей модуль запускається для коректної побудови веб-сервісу.

Проект використовує систему контролю версій Git та сервіс GitHub, у якому репозиторій було налаштовано таким чином, що при завантаженні змін у гілку master оновлений код автоматично починає розгортатися на PaaS "Heroku".

PaaS Heroku пропонує не тільки можливість розгорнути власний веб-сервіс, але й підключити до свого застосунку інші сервіси. З додаткових сервісів було використано:

- Heroku Postgres – для доступу до серверу PostgreSQL.
- Mailgun – для доступу до послуг SMTP-серверу.

Було вирішено зберігати структуру проектів не за допомогою бази даних, а за допомогою звичайної файлової системи, бо структура проекту саме це з себе і представляє. Amazon S3 надає цю структуру, до того ж існує зручне API [14] для роботи з цим сервісом, тому було обрано саме його для збереження структури проектів.

3.2 Опис розробки програми

3.2.1 Клієнтська частина

Структуру проекту клієнтської частини було вирішено поділити на такі частини:

- Компоненти – являють собою графічні частини застосунку. Кожний компонент складається з трьох частин:
    - Верстка – визначає графічну структуру.
    - Стилізація – визначає графічну стилізацію.
    - Логічна частина – визначає логіку компоненту.
- Моделі – являють собою контейнери для передачі інформації.
- Модулі – являють собою інструменти, які треба підключити до проекту.
- Сервіси – являють собою частину застосунку, які надають інтерфейс для обміну інформації з серверною частиною застосунку.
- Утиліти – являють собою шаблони коду, які використовуються для спрощення написання коду.

Завдяки цій структурі, у більшості випадків, процес створення нового функціоналу зводиться до таких кроків:

1. Створення компоненту.
2. Створення моделі, за допомогою якої буде передана інформація з сервера.
3. Якщо потрібно, створення сервісу.
4. Визначення взаємодії між компонентом та сервісом.

Стандартне використання на прикладі функціоналу реєстрації користувача:

```
export class RegistrationComponent {

public registrationForm: FormGroup;

constructor(private formBuilder: FormBuilder,
        private authService: AuthenticationService,
        private router: Router) {...}


 public onSubmit(): void {
  const user: User = new User(this.email.value, this.nickname.value, this.password.value);
  this.authService.register(user).subscribe(
   (_) => this.router.navigate(['/login']),
   (err: HttpErrorResponse) => {
    if (err.status === 400) {
     window.alert('User with such email already exists. Please choose another email.');
    } else {
     console.error(`Unexpected error occurred while registering: ${JSON.stringify(err)}`);
     window.alert('Unexpected error occurred ! Please try again.');
    }
   }
  );
 }

get email() {...}
get nickname() {...}
get password() {...}
get confirmPassword() {...}
}
```

Компонент

```
export class AuthenticationService {

  private static readonly REGISTER_ENDPOINT: string = environment.api_path +
'api/v1/auth/register';
  private static readonly LOGIN_ENDPOINT: string = environment.api_path + 'api/v1/auth/login';
  private static readonly ACTIVATION_ENDPOINT: string = environment.api_path +
'api/v1/auth/activateUser';

  private isAuthenticated: boolean;

  constructor(private http: HttpClient) {
  }

  public login(email: string, password: string): Observable<any> {...}

  public setAuthenticated(value: boolean): void {...}
  public getIsAuthenticated(): boolean {...}

  public register(user: User): Observable<any> {
    return this.http.post<any>(AuthenticationService.REGISTER_ENDPOINT, user);
  }

  public activateUserByToken(token: string): Observable<any> {...}
}
```

Сервіс

```
export class User {
  readonly id: number;
  readonly email: string;
  readonly nickname: string;
  readonly password: string;


  constructor(email: string, nickname: string, password: string) {...}
}
```

Модель користувача

3.2.2 Серверна частина

Структуру проекту серверної частини було вирішено поділити на такі частини:

- Конфігурація – складається з класів, які відповідають за конфігурацію проекту і визначення перехоплювачів запитів та подій.
- Контролери – являють собою API, за допомогою якого можна обмінюватись інформацією з серверною частиною.
- Моделі – являють собою контейнери для передачі інформації. У цій частині також визначаються об’єкти для ORM.

- Репозиторії – визначають API для обміну інформацією з базою даних. Реалізація репозиторіїв потребує невеликих зусиль завдяки використанню технології ORM.
- Сервіси – визначають всю бізнес логіку застосунку. Уся логіка застосунку закладена в цій частині.

Ми дотримувалися SOLID-принципів, тому для кожного контролера та сервісу створений інтерфейс і відповідна реалізація. Завдяки такому розподіленню серверної частини додання нового функціоналу не є складним і не вимагає багато часу. Це особливо помітно при використанні IoC контейнеру, який надає Spring.

Приклади використання:

```
package com.leliuk.onlinejavaideback.configuration;

import {...};

@Getter
@AllArgsConstructor
@ConstructorBinding
@ConfigurationProperties(prefix = "amazon-bucket-configuration")
public class AmazonS3Configuration {
   private final String accessKeyId;
   private final String secret;
   private final String bucketName;
   private final Regions region;

   @Bean
   public AmazonS3 getAmazonS3Client() {
      AWSCredentials credentials = new BasicAWSCredentials(accessKeyId, secret);
      return AmazonS3ClientBuilder.standard()
            .withCredentials(new AWSStaticCredentialsProvider(credentials))
            .withRegion(region)
            .build();
   }
}
```

Конфігурація підключення до Amazon S3

```java
package com.leliuk.onlinejavaideback.controller;

import {...};

@Validated
@RequestMapping("api/v1/projects/items")
public interface ProjectItemsController {

   @GetMapping("/all")
   List<ProjectItem> getProjectStructure(@RequestParam(name = PROJECT_ID_PARAMETER_NAME)
long projectId) throws ProjectItemsRetrievalException;

   @PostMapping("/createOrSave")
   void createOrSaveItem(@RequestParam(name = PROJECT_ID_PARAMETER_NAME) long projectId,
                 @Valid @RequestBody ProjectItem item) throws UnsupportedEncodingException;

   @PostMapping("/delete")
   void deleteItem(@RequestParam(name = PROJECT_ID_PARAMETER_NAME) long projectId, @Valid
@RequestBody ProjectItem item);
}
```

Інтерфейс контролеру

```java
package com.leliuk.onlinejavaideback.controller.impl;

import {...};

@Slf4j
@RestController
@RequiredArgsConstructor
@RequestMapping("api/v1/projects/items")
public class ProjectItemsControllerImpl implements ProjectItemsController {

   private final ProjectStructureService service;

   @Override
   public List<ProjectItem> getProjectStructure(long projectId) throws ProjectItemsRetrievalException
{
      return service.getProjectStructure(projectId, false);
   }

   @Override
   public void createOrSaveItem(long projectId, ProjectItem item) throws
UnsupportedEncodingException {
      service.createOrSave(projectId, item);
   }

   @Override
   public void deleteItem(long projectId, ProjectItem item) {
      service.delete(projectId, item);
   }

}
```

Реалізація контролеру

```java
package com.leliuk.onlinejavaideback.service;

import {...};

public interface ProjectStructureService {

   List<ProjectItem> getProjectStructure(long projectId, boolean withRoot) throws
ProjectItemsRetrievalException;

   void createOrSave(long projectId, ProjectItem item) throws UnsupportedEncodingException;

   void delete(long projectId, ProjectItem item);
}
```

Інтерфейс сервісу

```
package com.leliuk.onlinejavaideback.service.impl;

import {...};

@Slf4j
@RequiredArgsConstructor
@Service
public class ProjectStructureServiceImpl implements ProjectStructureService {

  private final BucketService bucketService;

  @Override
  public List<ProjectItem> getProjectStructure(long projectId, boolean withRoot) throws
ProjectItemsRetrievalException {
    String path = withRoot ? Long.toString(projectId) : projectId + BucketService.FOLDER_DELIMETER;
    return bucketService.getItemsFromFolder(path).stream()
        .map(obj -> s3ObjectToProjectItem(obj, withRoot))
        .collect(Collectors.toList());
  }

  @Override
  public void createOrSave(long projectId, ProjectItem item) throws UnsupportedEncodingException
    {...}

  @Override
  public void delete(long projectId, ProjectItem item) {...}

  private static String convertToFullPath(long projectId, String path) {...}

  private ProjectItem s3ObjectToProjectItem(S3ObjectSummary obj, boolean withRoot) {
    String key = obj.getKey();
    String projectPath = withRoot ? key :
key.substring(key.indexOf(BucketService.FOLDER_DELIMETER) + 1);
    if (key.endsWith(BucketService.FOLDER_DELIMETER)) {
      return new ProjectItem(true, projectPath, null);
    }

    try {
      return new ProjectItem(false, projectPath, bucketService.getTextContent(key));
    } catch (IOException e) {
      log.error("Can't read contents of S3Objcect:", e);
      throw new ProjectItemsRetrievalException();
    }
  }
```

Реалізація сервісу

```
package com.leliuk.onlinejavaideback.repository;

import {...};

@Repository
public interface UserRepository extends JpaRepository<Account, Long> {
}
```

Інтерфейс репозиторію

3.3 Компіляція та запуск коду

Коли користувач створив проект та структуру проекту, яка складається з пакетів та класів, у нього з'являється можливість створювати та виконувати вихідний код.

Для того, щоб виконати вихідний код користувачеві треба:

1. Створити клас, якщо він ще не створений.
2. Створити вихідний код в цьому класі.
3. Зберегти зміни.
4. Скомпілювати проект.
5. Виконати код обраного класу.

У майбутньому кроки 3, 4, 5 планується звести до одного.

Збереження структури проекту відбувається за допомогою сервісу Amazon S3. Під час запуску сервісу створюється сховище, в якому будуть зберігатися структури проектів всіх користувачів. Кожній структурі відповідає унікальний ID і саме за цим ID відбувається пошук потрібної структури проекту.

Компіляція вихідного коду у Java була стандартизована у версії 6.0, результатом чого став Java 6.0 Compiler API. Саме цей API використовується для компіляції вихідного коду. Компіляція в сервісі складається з таких кроків:

1. Усі класи завантажуються з Amazon S3.
2. Відбувається перевірка, чи правильно вказаний пакет на початку кожного з файлів.
3. Усі завантажені класи з текстового вигляду перетворюються у підкласи класу javax.tools.SimpleJavaFileObject.
4. Створюється екземпляр класу javax.tools.DiagnosticCollector, для того, щоб зібрати діагностику процесу компіляції.
5. За допомогою javax.tools.JavaCompiler створюється екземпляр класу javax.tools.JavaCompiler.CompilationTask.
6. Починається процес компіляції. У випадку, якщо компіляція відбудеться з помилками, то за допомогою створеного екземпляру

javax.tools.DiagnosticCollector вся потрібна інформація про помилки збирається і відправляється на клієнтську частину.

Вихідний код, що відповідає за процес компіляції надано у додатку А.

Запуск вихідного коду можна виконати декількома способами. Одним з найвідоміших є використання Reflection API. У цьому випадку створюється екземпляр класу java.lang.ClassLoader, за допомогою якого клас підвантажується, а потім використовується Reflection API для того, щоб дістати метод main класу та виконати. Іншим способом є створення окремого процесу і запуску обраного класу в цьому процесі. Було вирішено використовувати другий спосіб, бо завдяки створенню окремого процесу, виконання коду користувача ізолюється від виконання коду серверу.

Reflection API було вирішено використовувати для перевірки наявності коректного методу main для запуску коду.

Отже, процес запуску сервісом вихідного коду користувача складається з таких кроків:

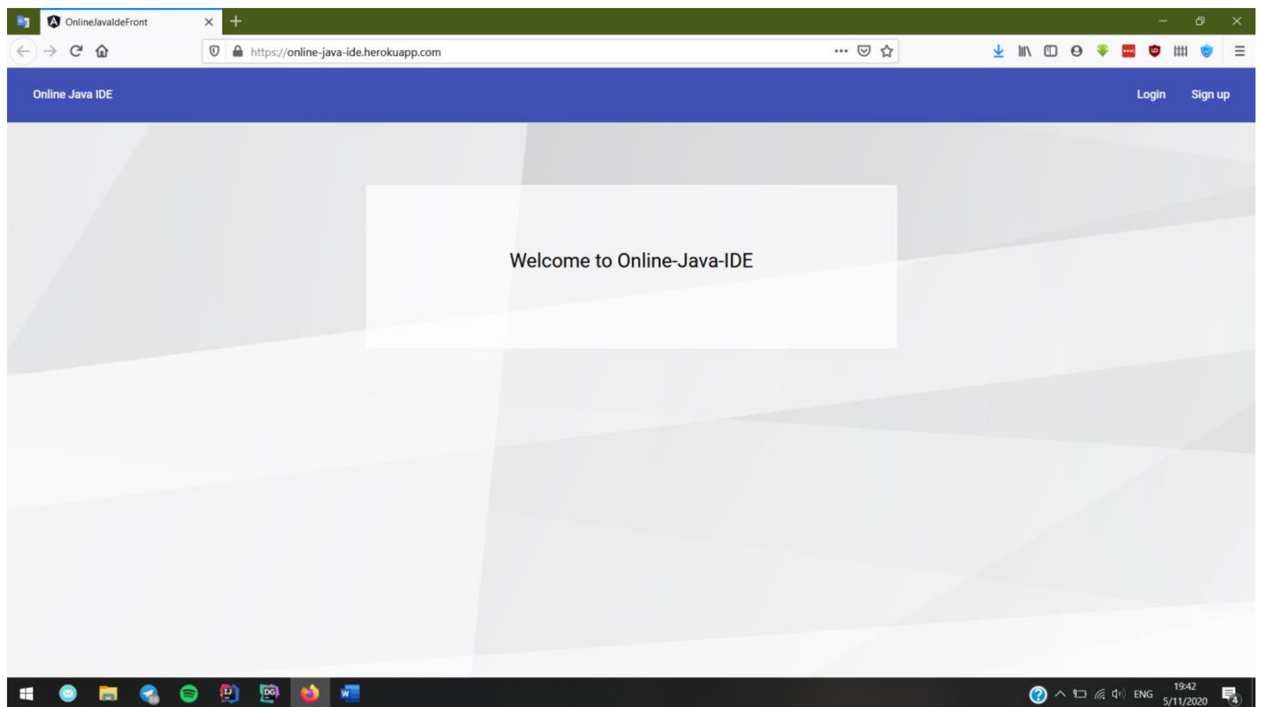
1. За допомогою java.lang.ClassLoader потрібний клас підвантажується і таким чином перевіряється його наявність.
2. Якщо клас існує, у його реалізації відбувається пошук методу main із потрібною сигнатурою.
3. Якщо метод існує, то створюється окремий процес та відбувається запуск класу.
4. Усі дані, що були надіслані у вихідний потік процесу, збираються та відправляються на клієнтську частину.

Вихідний код, що відповідає за процес компіляції надано у додатку Б.

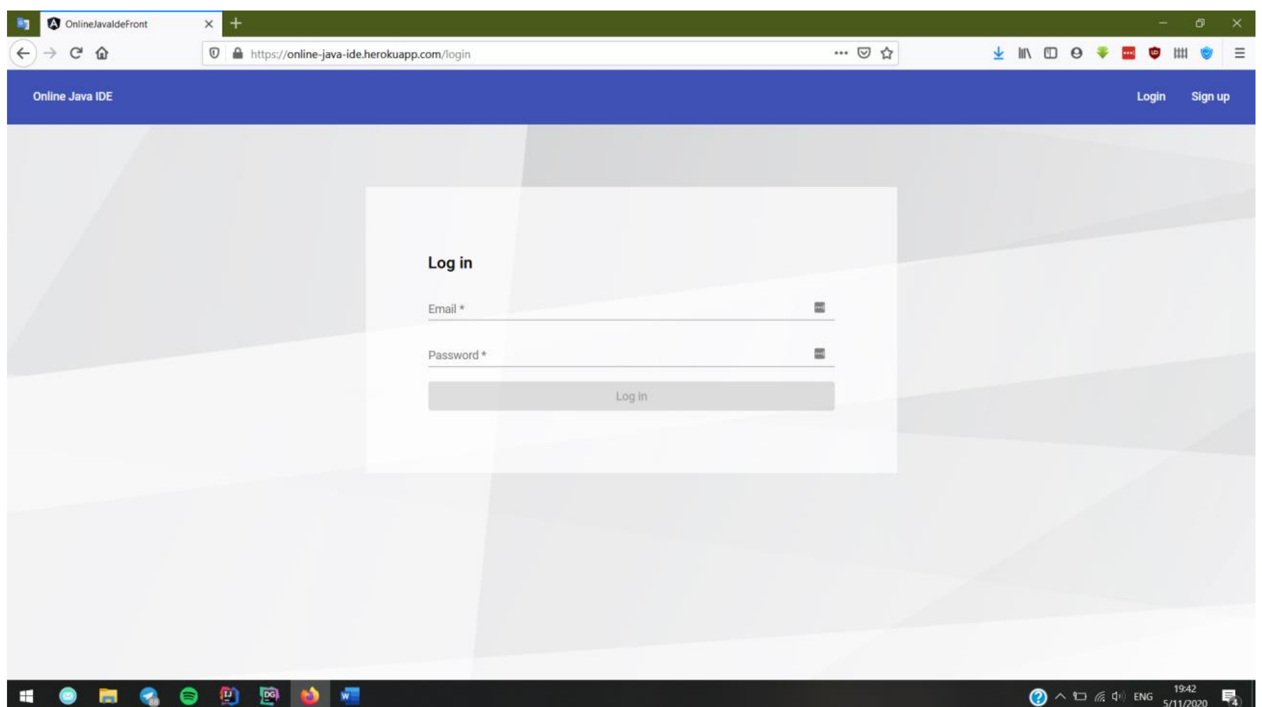
3.4 Опис файлів даних та інтерфейсу програми

Серверна частина програми створює папку під назвою compiled, де знаходяться скомпільовані класи з проектів користувачів.

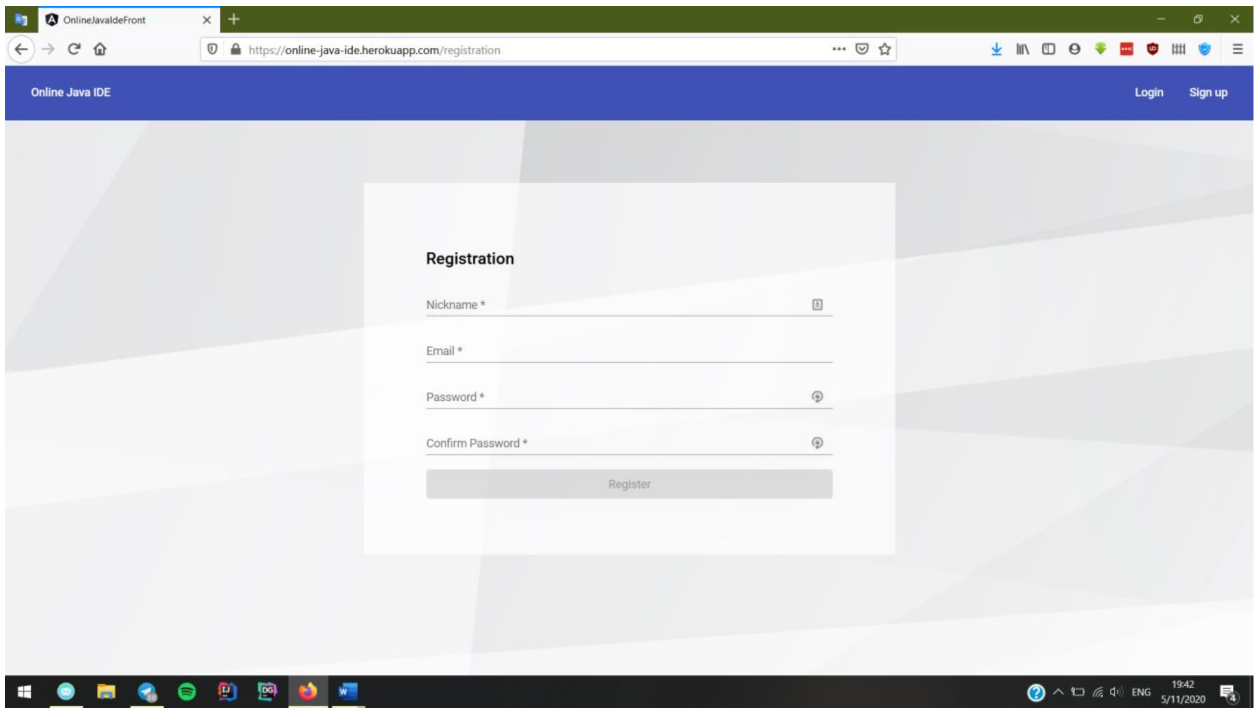
Інтерфейс веб-сервісу має такий вигляд:



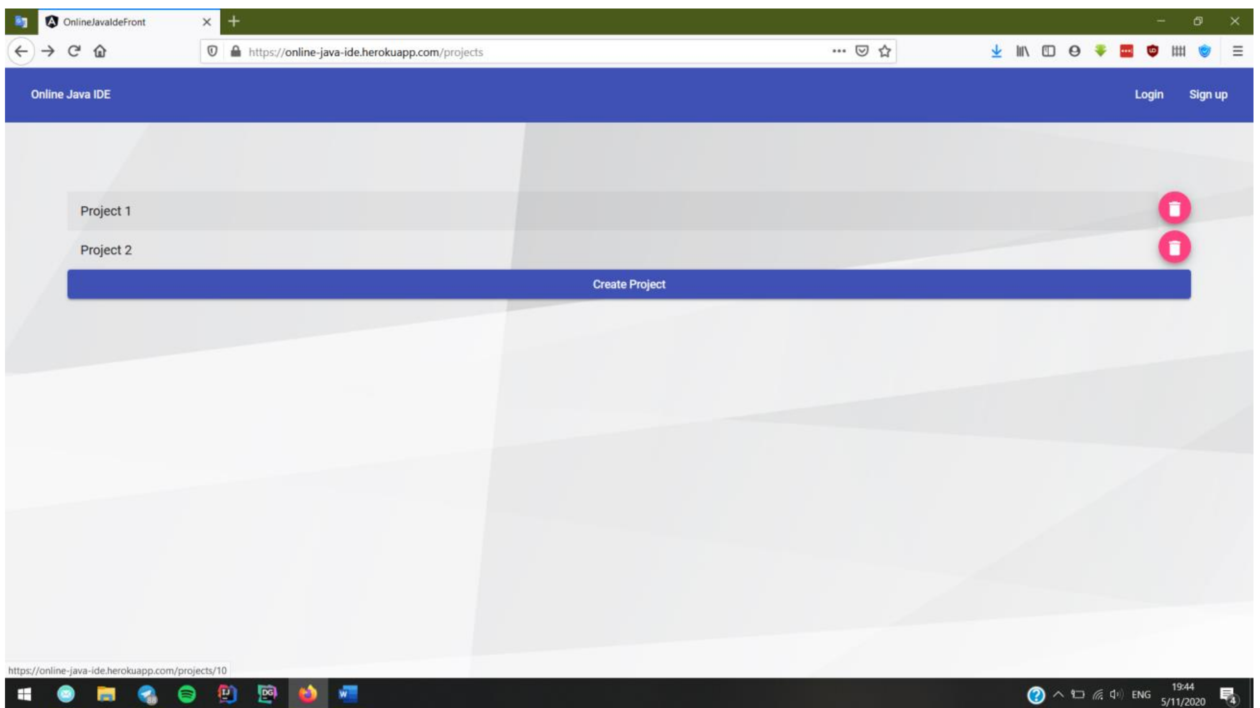
Початкова сторінка



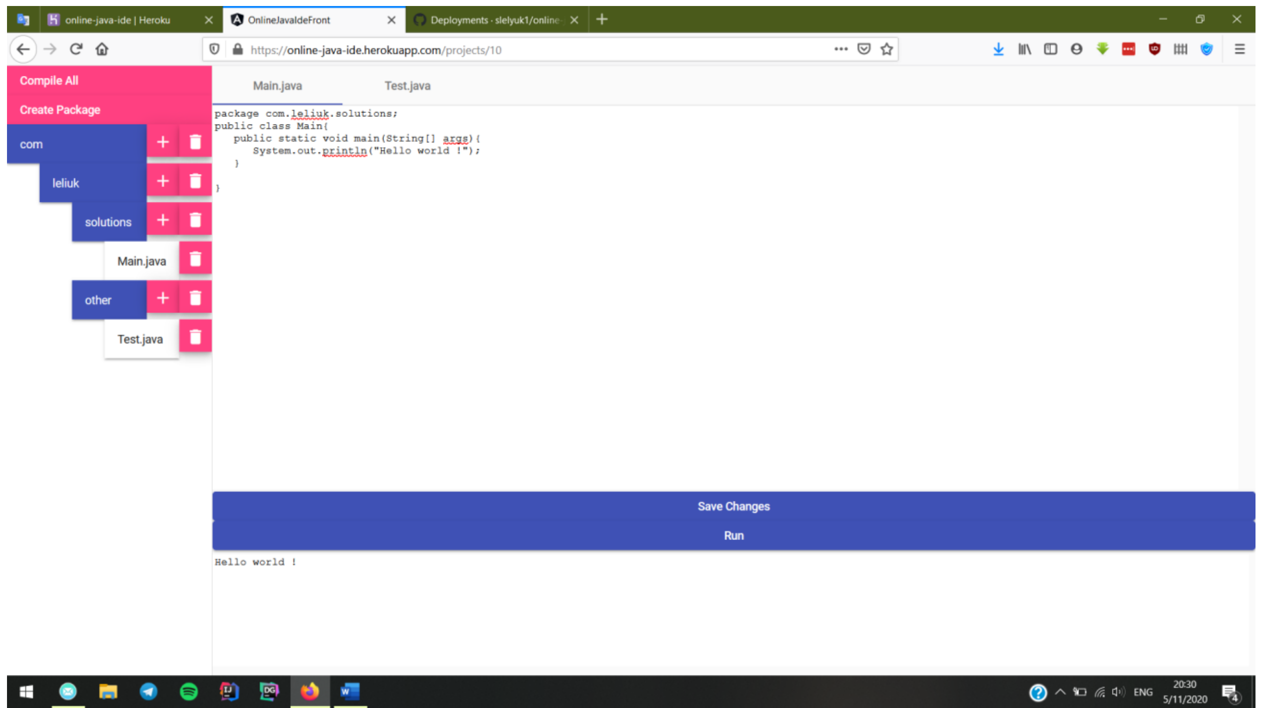
Сторінка входу



Сторінка реєстрації



Сторінка створених проектів



Сторінка структури проекту

Висновки

Було досліджено процес створення веб-сервісу, теорію зі створення веб-сервісу, технології, що полегшують процес створення клієнтської та серверної частини та поліпшують якість продукту. У результаті створено веб-сервіс, який являє собою онлайн середовище розробки мовою Java за допомогою фреймворків Spring та Angular. Створений веб-сервіс виконує такі функції: створення та збереження проектів, створення, збереження та редагування структури проектів, компіляція проектів та виконання коду. При належному покращенні й розвитку веб-сервісу, він може стати повноцінним продуктом, що являтиме собою аналог десктопних середовищ розробки мовою Java для розробки невеликих проектів.

Майбутнє вдосконалення веб-сервісу варто почати з покращення користувацького інтерфейсу, аби він був більш user-friendly. Розвиток веб-сервісу можна здійснювати в багатьох аспектах: додати функціонал імпорту вихідного коду з різних джерел; реформувати веб-сервіс у платформу з можливостями соціальної взаємодії та освітніх функцій. Веб-сервіс планується використовувати в освітніми цілями.

Список джерел

1. Web Application Development Process. Автор Бернард Коган. https://www.comentum.com/web-application-development-process.html
2. 8 Essential Phases in Web Application Development Process. Автор Арвінд Ронгала. Рік видання – 2015. https://www.invensis.net/blog/it/8-essential-phases-in-web-application-development-process/
3. Курс “An introduction to web applications architecture”. https://www.open.edu/openlearn/science-maths-technology/introduction-web-applications-architecture/content-section-1.1
4. Client-Server Model. Автор Шурікат Харун Суліман. Рік видання - 2014. https://www.researchgate.net/publication/271295146_Client-Server_Model
5. Service-oriented architecture (SOA). Автор IBM Corporation. Рік видання - 2016 https://www.ibm.com/support/knowledgecenter/en/SSMQ79_9.5.1/com.ibm.egl.pg.doc/topics/pegl_serv_overview.html
6. SOAP Tutorial. https://www.tutorialspoint.com/soap/index.htm
7. What is REST?. https://www.codecademy.com/articles/what-is-rest
8. RESTful Web Services Tutorial. https://www.tutorialspoint.com/restful/index.htm
9. RPC Tutorial https://www.tutorialspoint.com/xml-rpc/xml_rpc_intro.htm
10. TypeScript Handbook. https://www.typescriptlang.org/docs/handbook
11. Angular Documentation. https://angular.io/docs
12. Spring in Action 5th Edition Автор Крейг Волс. Рік видання - 2019
13. Hibernate Tutorial.

https://www.tutorialspoint.com/hibernate/index.htm

14. AWS SDK for Java 2.0 Developer Guide.
https://docs.aws.amazon.com/sdk-for-java/v2/developer-guide/welcome.html

Додаток А

Метод, що виконує компіляцію вихідного коду

```
@Override
public List<CustomDiagnostic> compile(long projectId) {
   List<StringJavaFileObject> toCompile = new ArrayList<>();
   for (ProjectItem item : projectStructureService.getProjectStructure(projectId, true)) {
      if (!item.getIsPackage()) {
         if (!hasValidPackageHeader(item)) {
            return Collections.singletonList(CustomDiagnostic.packageInvalid(item));
         }
         toCompile.add(StringJavaFileObject.of(item.getRelativePath(), item.getContent()));
      }
   }

   DiagnosticCollector<JavaFileObject> diagnosticCollector = new DiagnosticCollector<>();
   JavaFileManager fileManager = compiler.getStandardFileManager(diagnosticCollector, null, null);
   String compiledPath = COMPILED_FOLDER_PATH + File.separator + projectId;
   new File(compiledPath).mkdirs();
   List<String> params = Arrays.asList("-d", compiledPath);
   boolean successful = compiler.getTask(null, fileManager, diagnosticCollector, params, null,
toCompile).call();
   if (!successful) {
      return diagnosticCollector.getDiagnostics().stream()
            .map(CustomDiagnostic::from)
            .collect(Collectors.toList());
   }
   return Collections.emptyList();
}

private boolean hasValidPackageHeader(ProjectItem item) {
   String[] relativePath = item.getRelativePath().split(BucketService.FOLDER_DELIMETER);
   String[] correctPackagePath = Arrays.copyOfRange(relativePath, 1, relativePath.length - 1);
   if (item.getIsPackage() || correctPackagePath.length == 0) {
      return true;
   }
   String[] lines = item.getContent().trim().split("\n");
   if (lines.length < 1) {
      return false;
   }
   String packageLine = lines[0].trim();
   int indexOfEnd = packageLine.lastIndexOf(';');
   if (indexOfEnd == -1) {
      return false;
   }
   String[] packageLineParts = packageLine.substring(0, indexOfEnd).split(" ");
   return packageLineParts.length >= 2
         && packageLineParts[0].equals("package")
         && Arrays.equals(correctPackagePath, packageLineParts[1].split("\\."));
}
```

Додаток Б

Метод, що виконує запуск скомпільованого коду

```
@Override
public List<CustomDiagnostic> compile(long projectId) {
   List<StringJavaFileObject> toCompile = new ArrayList<>();
   for (ProjectItem item : projectStructureService.getProjectStructure(projectId, true)) {
      if (!item.getIsPackage()) {
         if (!hasValidPackageHeader(item)) {
            return Collections.singletonList(CustomDiagnostic.packageInvalid(item));
         }
         toCompile.add(StringJavaFileObject.of(item.getRelativePath(), item.getContent()));
      }
   }

   DiagnosticCollector<JavaFileObject> diagnosticCollector = new DiagnosticCollector<>();
   JavaFileManager fileManager = compiler.getStandardFileManager(diagnosticCollector, null, null);
   String compiledPath = COMPILED_FOLDER_PATH + File.separator + projectId;
   new File(compiledPath).mkdirs();
   List<String> params = Arrays.asList("-d", compiledPath);
   boolean successful = compiler.getTask(null, fileManager, diagnosticCollector, params, null,
toCompile).call();
   if (!successful) {
      return diagnosticCollector.getDiagnostics().stream()
            .map(CustomDiagnostic::from)
            .collect(Collectors.toList());
   }
   return Collections.emptyList();
}
```