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

БАГАТОРІВНЕВЕ ВЕБ-ЗАСТОСУВАННЯ НА DOCKER-ПЛАТФОРМІ

Текстова частина до курсової роботи
за спеціальністю „Комп'ютерні науки ” 6.050122

Керівник курсової роботи
к.т.н., доц. _____
(прізвище та ініціали)
_____
(підпис)
“____” _____ 2020 р.

Виконав студент _____
_____
(прізвище та ініціали)
“____” _____ 2020 р.

Київ 2021

Тема : Багаторівневе веб-застосування на docker-платформі.

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу.	05.11.2020	
2.	Огляд технічної літератури за темою роботи.	18.11.2020	
3.	Ознайомлення з літературою.	25.11.2020	
3.	Вивчення історії розробки та підтримки технологій Docker, дослідження останніх нововведень	15.12.2020	
4.	Аналіз деталей імплементації Docker-контейнерів, існуючих рішень по їх застосуванню.	07.01.2021	
5.	Вивчення документації для формування уявлення про роботу практичної частини.	15.01.2021	
6.	Початок проектування основної структури практичної частини.	11.02.2021	
7.	Програмування реалізації практичної частини.	7.03.2021	
8.	Створення слайдів для доповіді та написання доповіді.	22.03.2021	
8.	Аналіз отриманих результатів з керівником, закінчення роботи над практичною частиною, закінчення роботи над доповіддю.	31.03.2021	
10.	Корегування практичної роботи з керівником.	7.04.2021	
11.	Остаточне оформлення доповіді та слайдів.	11.04.2021	
12.	Захист курсової роботи.	20.04.2021	

АНОТАЦІЯ

У даній роботі розглядається технологія Docker-контейнерів та більш широке поняття контейнеризації, а також практичне використання цієї технології у поєднанні з мікросервісною архітектурою. Проводиться аналіз внутрішньої роботи контейнерів, процес побудови контейнерних зображень, а також роботи Docker Server з ними. Також у роботі представлена внутрішня структура багаторівневого веб-застосування що використовує Docker для деяких функцій, зокрема деплойменту та CI/CD.

Зміст

ВСТУП

Розробка програмного забезпечення є складною та ресурсоємкою діяльністю, це просто об’єктивний факт реальності. Чим більше сфера комп’ютерних технологій розвивається, тим більшими та складнішими стають програми, і з часом внесення змін або навіть доставка віртуальних продуктів до цільової аудиторії стає клопіткою працею. Що гірше, цей процес ускладнюється навіть більше коли потрібно враховувати розбіжність програмного забезпечення між розробниками та основним сервером – проблема, яка здатна, часом, повністю «заморозити» процес розгортання нових змін. Саме для вирішення цієї проблема була придумана контейнеризація. Контейнери, для яких Docker є фактичним галузевим стандартом на момент написання цієї роботи, є невеликими незалежними від мови програмування та ізольованими від ОС пакетами програм, суворо необхідних для коректного функціонування продукту. Цей підхід дозволяє усунути програмні відмінності між розробником та сервером, що, в свою чергу, значно полегшує розгортання застосувань. Крім того, вони є легко поєднуються з мікросервісами та хмарними обчисленнями, що зробило їх дуже популярною технологією у світі, де велетенські сервіси на кшталт Amazon та Netflix потроху стають стандартом.

Отже, темою цієї роботи є огляд поточних галузевих тенденцій щодо контейнерів в цілому та Docker зокрема, наявної інформації про їх застосування, а також спроба застосувати зазначені тенденції на практиці, для перевірки того, наскільки такі підходи можуть бути застосовані для ситуацій з значно більш обмеженими ресурсами. Метою цієї роботи є провести глибоке вивчення внутрішніх механік технології контейнеризації (особливо Docker-контейнерів та інших сервісів, наданих цією компанією), дослідити причин буму мікросервісів, спричинених її поширенням та популяризацією, та використати цю інформацію для розробки веб-застосування для публікування та перегляду новин.

РОЗДІЛ 1: Контейнери та Docker. Концепти та існуючі рішення

1.1 Базове поняття контейнеризації

Згідно з офіційною документацією Docker, контейнер – це стандартизована одиниця програмного забезпечення, у якій запаковано код та всі його залежності, що дозволяє програмі працювати швидко та надійно незалежно від обчислювального середовища. Зображення контейнера Docker – це малий, автономний та готовий до виконання пакет програмного забезпечення, що включає все необхідне для запуску програми: код, робочий, системні інструменти, системні бібліотеки та налаштування. [1]

Роль, яку виконує контейнер, подібна до ролі віртуальної машини (VM). Аналогічно до віртуальної машини, контейнер – це програмна абстракція, яка має на меті ізолювати всі необхідні компоненти програми чи програм, щоб усунути деякі розбіжності та розпаралелювати процеси. Однак є важливі відмінності:

- Рівень абстракції – контейнер є абстракцією програмного рівня, що означає відсутність власного ядра ОС та підпорядкованість процесам ОС машини. На відміну від цього, ВМ є абстракцією рівня обладнання та використовує власну ОС та бінарні файли, що дозволяє їй бути повністю окремою від головної ОС.
- Розмір – контейнер містить лише бінарні файли, необхідні для запуску програми (наприклад лише JVM та стандартних бібліотек Java), і тому зазвичай займає лише кілька десятків Мб. ВМ, з іншої сторони, можуть вимагати значно більше ресурсів – іноді декілька Гб.
- Швидкість – кілька віртуальних машин під контролем гіпервізора часто вимагають більше ресурсів для підтримки паралельних операцій, необхідних для роботи кожної ОС. Контейнери виконують лише операції, необхідні для функціонування та контролю програм всередині, зменшуючи додаткові витрати ресурсів системи.

Хоча контейнери не є стовідсотковою заміною віртуальних машин – останні все ще є корисними для окремих користувачів або робочих станцій з декількома ОС, та можуть бути використаними у поєднанні з контейнерами – вони роблять паралельний запуск декількох ізольованих сервісів на одній машині значно більш зручним та швидким, ніж це було можливо до їх появи.

1.2 Існуючі використання технології, переваги та проблеми

Microservices architecture

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities
- Owned by a small team

The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack [7].
Мікросервісна архітектура не обов’язково вимагає використання Docker контейнерів, але популярність мікросервісів сильно пов’язана з популярністю Docker та навпаки, та більшість мікросервісів зараз використовують Docker для CI/CD, оскільки ідея малих ізольованих контейнерів працює надзвичайно добре з ідеєю малих ізольованих сервісів, які розгортаються в окремому порядку. Завдяки цій синергії, Docker є практично синонімічним з мікросервісами[8], тому розгляд існуючих використань цієї технології має бути проведений саме у контексті мікросервісів.
Загальна картина переваг та проблем цієї архітектури виглядає так [9][10][11][12]:
Переваги:

- Простота трасування запитів
- Обмеженість внутрішніх структур даних одним мікросервісом
- Більша продуктивність за рахунок автономії команд
- Підтримка комунікації між мовами та розширення робочого стеку внаслідок цього
- Ефективне розширення сервісу
- Простота оновлення

Проблеми:

- Підвищення часу обробки запитів
- Проблеми з кешуванням

- Складність в знаходженні потрібного сервісу
- Більше точок доступу – більше помилок
- Більше систем – більше тестування
- Проблематично для малих компаній

Набір технологій

Точні дані по популярності певних технологій є невідомими, але, базуючись на кількості позначок “Star” для образів на Docker Hub, можна виділити такі технології як найбільш популярні в своєму роді:

- Ubuntu серед операційних систем (також популярний: centOS)
- Node.js серед мов програмування бізнес-логіки (також популярні: PHP, python, Go)
- Nginx серед засобів балансування нагрузок та доступу до статичного контенту (також популярні: httpd, Apache Maven)
- MySQL серед СУБД (також популярні: MongoDB, Postgres, MariaDB)

Фреймворки для фронтенду зазвичай не мають власних образів, оскільки їх неможливо запустити окремо. По цій причині, фронтенд зазвичай парується з бекендом чи статичними серверами, такими як nginx чи Apache Maven.

Висновки розділу

Docker контейнери мають багато властивостей, що роблять їх корисними для застосування в розробці застосувань, але основним їх використанням є Мікросервіси. Мікросервісна архітектура стала практично синонімічною з контейнеризацією, оскільки саме вона отримує найбільше від використання Docker чи інших контейнерів під час розробки.

Мікросервіси є складною технологією зі значною кількістю нюансів, а також своїми як перевагами, так і недоліками. Основних з цих недоліків є висока складність використання мікросервісів для підтримки застосувань малого розміру – при відсутності великої команди, вони активно сповільнюють розробку, роблячи створення прототипу нового стартапу практично неможливим. Застосування Docker в такій ситуації є можливим, але не дуже практичним – додаткова складність у побудові надає певні переваги у вигляді портативності та простоти розширення, але ці переваги

не є дуже актуальними для маленької компанії з сервісом, що не отримує сотні тисяч запитів щодня.

По цій причині, будь-який аналіз використання Docker неминуче перетворюється на розгляд деталей мікросервісної архітектури, де його використання є найбільш актуальним. По цій причині, у наступному розділі буде описано процес побудови багаторівневого веб-застосування на основі цієї архітектури.

РОЗДІЛ 2: Структурна розробка власного рішення

3.1 Постановка цілі

Проаналізувавши попередні використання технології та мікросервісну архітектуру в цілому, стає очевидною складність використання цієї технології (принаймні у практичному, актуальному вигляді) для маленької команди розробників. Оскільки формат курсової роботи не дозволяє існування більше ніж одного автора, постала потреба у побудуванні застосування таким чином, щоб дозволити побудову сервісу, функціонал якого може бути реалістично розробленим та підтримуваним командою з одного розробника, у досить невеликий час, тим не менш отримуючи усі можливі плюси від використання Docker контейнерів. Також, враховуючи предметну область веб-застосування (а саме, сайт новин), потрібен зручний інтерфейс як для користувачів сервісу, так і для авторів, для забезпечення простоти і процесу створення контенту, і процесу його сприйняття.

Необхідний функціонал – інтерфейс для перегляду новин, можливість залишати коментарі під новинами, інтерфейс для створення новин та сервер бізнес-логіки для роботи з базою даних (необхідною для забезпечення збереження даних).

Отже, ціль – створити структуру застосування таким чином, щоб мінімізувати додаткові витрати людського ресурсу та максимізувати портативність та поширюваність застосування, у той же час даючи якомога простіший доступ для авторів та якомога більш стабільний сервіс для користувачів.

3.2 Структура застосунку

Мікросервіси, в момент їх створення, вимагають додаткові зусилля від команди розробників – монолітний сервіс, хоч він і є менш портативним, вимагає менше роботи від команди розробників. Отже, логічно намагатись зробити структуру мікросервісів якнайбільш наближеною до монолітної – ділити продукти на сервіси тільки якщо це забезпечить вищий степінь поширення застосування. Як наслідок, для проектування мікросервісів застосовувався патерн «Self-contained service» [13] (англ. самодостатній сервіс). Є інші варіанти поділу на мікросервіси:

- «Decompose by subdomain» [14] (англ. поділ за субдоменами) – кожен мікросервіс має відповідати одному набору бізнес-правил. Він не підходить для

задачі у зв’язку з невідповідністю вимоги до мінімальної кількості поділів, що буде описано пізніше.

- «Decompose by business capability» [15] (англ. поділ за потребами бізнесу) – кожен мікросервіс має відповідати одній ділянці, яка потребує окремого нагляду бізнесу (наприклад, продаж, купівля, менеджмент та інвентар є окремими ділянками, якими займаються окремі люди, а отже кожен з них формує окремий мікросервіс). Цей патерн не підходить, оскільки всі бізнес-процеси лежать в руках однієї людини.
- «Service per team» [16] (англ. сервіс на команду) – кожним мікросервісом займається одна команда спеціалістів. Використання патерну неможливе через відсутність команд.

Для функціонування застосування необхідні принаймні три структури даних – Article, чи Стаття (для відображення новин), User, чи Користувач (для додавання коментарів та забезпечення автентифікації та авторизації) та Comment, чи Коментар (для операцій з коментарями). Ці структури даних формують прості субдомени, оскільки всі з них описують різні дії та моди відображення та керуються різними бізнес-правилами, тому за застосування поділу за субдоменами, кожен з них би формував власний мікросервіс. Проте, відображення коментарів без відображення статей не є можливим, а отже, формування з них окремого мікросервісу результувало б у порушенні масштабування процесів. У зв’язку з цим, коментарі та статті були об’єднані у один мікросервіс, NewsAPI. Фінальний поділ відображений у схемі у додатку Е, та формується такими мікросервісами:

- UserFrontEnd – інтерфейс для звичайних користувачів. Очікується, що цей сервіс буде вимагати найбільше ресурсів (та, як наслідок, потребує окремого масштабування), та поєднання з інтерфейсом для адміністраторів системи та авторів порушує загальну безпеку застосування. Він ізолює відображення контенту для користувачів.
- AdminFrontEnd – інтерфейс для авторів та адміністраторів системи. Очікуване коло користувачів значно менше, ніж у UserFrontEnd. Ізолює відображення

меню для додавання нового контенту, видалення чи зміни існуючих статей та регулювання акаунтів користувачів.

- NewsAPI – бекенд, що відповідає на запити, пов’язані з новинами чи коментарями для новин. Відповідає за основну частину функціоналу веб-застосування разом з UserFrontEnd. Ізолює доступ до даних про новини та коментарі, та комунікує з UsersAPI для перевірки авторизації, необхідної для створення нових новин чи редагування вже існуючих.
- UsersAPI – бекенд, що відповідає на запити, пов’язані з користувачами. Відповідає за безпеку застосунку, особистих даних користувачів, а також за збереження даних про використання користувачами застосунку (тобто, їх коментарів та статей).

Окрім основних структурних компонентів, варто зауважити ізоляцію баз даних. У той час, як для монолітного застосування була б використана лише одна СУБД та одна база даних, для максимального масштабування та виконання вимог патерну поділу на самодостатні сервіси, кожен з бекендів працює з власною базою даних.

Це рішення мало наступні переваги:

- СУБД можуть бути змінені в процесі розробки без впливу на інші сервіси
- Помилки в роботі однієї бази даних не сповільнюють роботу інших сервісів

Але призвело до загального сповільнення розробки у зв’язку з потребою робити перевірки між сервісами, замість того щоб, допустимо, перевіряти рівень доступу користувача у мікросервісі новин.

3.3 Детальний опис API

За схемою, описаною вище, у застосуванні існує два великих бекенд компоненти, один з яких відповідає за більшу частину використання сайту, а другий – за його безпеку. Очевидно, що для кожного з цих компонентів мали бути визначені численні точки доступу зовні, для обробки запитів від існуючих та майбутніх інтерфейсів.

2.3.1 NewsAPI

NewsAPI визначає такі точки доступу:

- “/articles/:page” (GET) – повертає посилання на усі існуючі на сервісі новини, впорядковані за датою їх додавання на сервер. З метою економії інтернет-

трафіку та забезпечення максимального масштабування, повертаються лише заголовки новин (що реалізовані у вигляді окремої сутності Headline, та містять лише час новини, її назву та короткий опис). Заголовки групуються у сторінки по 20 заголовків кожна – параметр page відповідає за номер сторінки.

- “/article/:id” (GET) – повертає інформацію про одну окрему новину, позначену параметром id (що має вигляд цілого числа). Інформація є майже повною (включає дату та час додавання на сервіс, час останньої редакції, назву, ключові слова, повний текст та автора) та не вимагає додаткової перевірки на автентифікацію.
- “/articles/” (POST) – дозволяє створення нових статей. Передає токен авторизації запиту до UsersAPI для перевірки наявності у користувача прав створення нових статей (такі права є лише у ролей Admin та Author). У випадку відсутності таких прав кидає помилку.
- “/article/:id” (PUT) – дозволяє редакцію існуючих статей. Перевіряє токен авторизації запиту через UsersAPI (право на редакцію є лише у адміністратора та автора статті). Не змінює оригінальні час та дату публікації статті, натомість зберігає час запиту як «час та дату останньої зміни». У разі відсутності статті з таким id, створює нову.
- “/article/:id” (DELETE) – дозволяє видалення існуючих статей. Працює за механікою, аналогічною до попереднього методу. У разі відсутності статті, нічого не робить.
- “/article/:id/comments” (GET) – повертає усі коментарі, що були залишені під цією статтею. Не вимагає авторизації чи автентифікації.
- “/article/:id/comments” (POST) – дозволяє додавати нові коментарі до статті. Користувачі будь-якої ролі мають доступ до цієї функції, але для неї все одно необхідна автентифікація.
- “/comment/:id” (GET) – повертає інформацію про окремий коментар. Не вимагає авторизації чи автентифікації.

- “/comment/:id” (PUT) – дозволяє редагування коментарів. Така можливість є лише у користувача, що залишив цей коментар, модератора (роль MOD) та адміністратора, інакше повертає помилку.
- “/comment/:id” (DELETE) – дозволяє видалення коментарів.

Для взаємодії з СУБД використовується фреймворк Java Spring Boot. Він був обраний завдяки простоті розробки та де факто є не найкращим можливим вибором для розробки мікросервісів. У разі подальшої розробки та можливих проблем, він буде замінений на засіб розробки з меншим додатковим використанням ресурсів, наприклад, Node.js express чи Go Gin Gonic.

На даний момент розробки, цей API використовує СУБД MySQL. Більш складні запити до БД виконуються за допомогою механізмів Spring Boot, тому найбільш важливим фактором є швидкість та доступ для багатьох користувачів. MySQL є гарним балансом між малими однопоточними СУБД SQLite та H2, та більш функціональною але повільнішою PostgreSQL. Обраний формат застосування дозволяє з часом мігрувати на NoSQL базу даних у випадку катастрофічного збільшення розмірів бази даних.

2.3.2 UsersAPI

UsersAPI визначає наступні точки доступу:

- “/login” (POST) – не вимагає авторизації чи автентифікації (оскільки є точкою входу). Якщо надані у запиті поля «email» та «password» співпадають з відповідними полями одного з користувачів, збережених у базі даних, повертає JWT (JSON Web Token), яким можна користуватись для доступу до даних у системі.
- “/register” (POST) -- не вимагає авторизації чи автентифікації (оскільки є точкою входу). Створює нового користувача з рівнем доступу USER (доступ лише до власного акаунту користувача та можливості створювати коментарі), якщо надані нікнейм та адреса електронної пошти не присутні у базі даних (вони задекларовані як унікальні), повертає JWT, що відповідає цьому користувачеві, та проводить редирект на сторінку користувача.

- “/user/:id” (GET) – перевіряє авторизацію та повертає сторінку користувача. Лише сам користувач та адміністратор системи мають доступ до сторінки користувача.
- “/user/:id” (PUT) – перевіряє авторизацію та заміняє дані користувача з id, вказаним у запиті, на нові надані дані. Доступний лише для користувача та адміністратора. На відміну від звичайної імплементації відповіді на метод PUT, повертає помилку, якщо вказаного користувача не існує у системі.
- “/user/:id” (DELETE) – перевіряє авторизацію та видаляє вказаного користувача. Доступний лише для користувача та адміністратора. Оскільки перевірка на JWT працює через базу даних, токен, вже отриманий клієнтом, буде вважатись недійсним для наступного запиту.
- “/verify/” (POST) – перевіряє дійсність JWT, надісланого разом з запитом, та повертає ролі користувача у разі його дійсності. Зроблено здебільшого для використання іншим бекендом, NewsAPI.

Для взаємодії з СУБД використовується фреймворк Java Spring Boot. Окрім попередньо зазначених пунктів, Spring Security є доволі зручним фреймворком для проведення операцій безпеки, для яких цей бекенд був створений, та використання одного формату на обох бекендах дозволяє дещо пришвидшити розробку.

На даний момент розробки, цей API використовує СУБД H2, СУБД для проектів на JVM з відкритим кодом. Вона є досить швидкою, щоб певний час не перейматись проблемами з її масштабуванням, та працює особливо добре Hibernate, що входить до складу залежностей Spring.

РОЗДІЛ 3: Детальна розробка UsersAPI

3.1 Постановка проблеми

За наданим вище описом, UsersAPI є основним компонентом, відповідальним за доступ до бази даних користувачів (що само по собі робить безпеку дуже високим пріоритетом в його розробці), та загальну безпеку застосунку. Для звичайного застосунку на Spring Boot, створення такого компоненту було б простим – фреймворк Spring Security дає доступ до значної кількості спрощень, що дозволяють захистити застосування від проблем з безпекою з мінімальною витратою людських годин.
На жаль, ці спрощення не зроблені з врахуванням архітектури мікросервісів – у разі використання, вони б працювали лише на точках доступу, розгорнутих на цьому сервері. Це б означало повну відсутність перевірок для іншого бекенду, що є звичайно не настільки ж руйнівним, як доступ до особистих даних користувачів, але все одно руйнує будь-яку перспективу роботи застосування в умовах реального світу.
Було необхідне створення системи, що не тільки дозволяє іншому серверу слугувати посередником у перевірці, а й здатна працювати у декількох копіях – однією з головних причин створення веб-застосування була перспектива майбутнього масштабування з допомогою інструментів оркестрування контейнерів Docker, таких як Docker Swarm чи Kubernetes.
Існує декілька методів вирішення цієї проблеми:

- Secure cookies – збереження стану сесії та використання його лише під час обміну даними за допомогою HTTPS. Це рішення є доволі очевидним та розповсюдженим, але оскільки обидва наші API функціонують за принципами REST, які вимагають відсутність сесій, та мають комунікувати одне з одним, це рішення не є оптимальним варіантом.
- HTTP Basic – передача даних автентифікації у хедері Authentication, закодованих у Base64. Проблема у використання цього стандарту полягає у тому, що кодування можна обернути, і тому без доступу до HTTPS він є абсолютно неефективним. HTTPS є де-факто стандартом сучасної інтернет комунікації, але оскільки створені автором мікросервіси спроектовані для максимального потенціалу майбутньої розробки, він не є гарантованим

варіантом. До того ж, втрата конфіденційності користувачів, навіть всередині системи, є поганою ідеєю.

- HTTP Digest – передача даних автентифікації у хедері Authentication, захешованих за допомогою MD5. MD5 є застарілим методом хешування, що є доведено вразливим до зламу, і не має використовуватись для забезпечення безпеки у будь-якому застосуванні.
- JSON Web Token – це запропонований Інтернет-стандарт для створення даних з необов'язковим підписом та / або необов'язковим шифруванням, що містять у собі JSON з певними даними, зазвичай необхідними для автентифікації. Токени були спроектовані для того, щоб бути компактними, URL-безпечними та корисними для схем SSO (Single Sign-On). Заяви JWT, як правило, можуть використовуватися для передачі ідентифікації автентифікованих користувачів між постачальником ідентифікації та постачальником послуг. Загалом, вони є ідеальними для вирішення сформульованої проблеми.

Отже, після вибору програмного рішення, лишилось лише спроектувати необхідні класи, необхідні для формування та розпізнавання JWT, а також решту бізнес-логіки.

3.2 Реалізація стандартних функцій

Оскільки однією з основних функцій мікросервісу є збереження особистих даних користувачів, варто вказати, яким саме чином це було зроблено.

Клас User

```
@Entity
@Table(name = "users", uniqueConstraints = {
    @UniqueConstraint(columnNames = {
        "email",
        "nickname"
    })
})
public class User {

    @JsonIgnore
    private static final BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
```

```
    @Enumerated(EnumType.STRING)
    private HashSet<Role> roles;

    @JsonIgnore
    private String encodedPassword;
    private String email;
    private String nickname;

    protected User() {}

    public User(String password, String nickname, String email){
        this.roles = new HashSet<Role>();
        this.roles.add(Role.USER);
        encodedPassword = encoder.encode(password);
        this.email = email;
        this.nickname = nickname;
    }

… getters, setters…

    public enum Role {
        ADMIN,
        MODERATOR,
        AUTHOR,
        USER
    }
```

Варто підмітити додатковий рівень захисту паролів користувачів – вони збережені лише в зашифрованому вигляді, тому їх неможливо дізнатись навіть при наявності повного доступу до бази даних.

Шифрування відбувається за допомогою класу BCrypt в фреймворку Spring Security, що імплементує однойменний алгоритм шифрування. Метод є повільним (що не дає зловмисникам отримати список паролів методом перебору) та має регульований рівень складності, що робить його гарним вибором для довготривалої підтримки.

Шифрування паролю відбувається одразу при його отриманні, та він передається за допомогою JWT, що робить процес безпечним від початку до кінця навіть якщо використання HTTPS не є можливим.

У користувача може бути будь-яка комбінація з наступних ролей:

- USER є найбільш базовою роллю та надає доступ лише до власного профіля, можливості створювати нові коментарі та можливості редагувати чи видаляти

власні коментарі. Ця роль надається усім користувачам під час створення облікового запису і не може бути забрана.

- Профіль, видалений користувачем, не видаляється з бази даних, а оголошується «деактивованим», та може бути відновленим з допомогою адміністратора.

- AUTHOR є розширенням USER в області створення контенту – вона дозволяє створювати нові статті та редагувати чи видаляти власні статті.
- MODERATOR є розширенням USER в області модерування контенту – вона дозволяє редагувати чи видаляти чужі коментарі (окрім коментарів інших модераторів чи адміністратора).
- ADMIN надає повний доступ до системи і не може бути надана за допомогою наявних API. Єдиний спосіб зробити користувача адміністратором лежить через консоль бази даних.

  В адміністратора є доступ до наступних привілегій:

  - Видалення користувачів, надання їм чи позбавлення їх ролей (окрім адміністратора), зміна їх особистих даних (з цілями технічної підтримки) та
  - Створення, редагування чи видалення статей будь-яких користувачів.
  - Створення, редагування чи видалення коментарів будь-яких користувачів.

  З ціллю уникнення потенційного зловживання владою, записи всіх дій адміністратора зберігаються.

Клас UserDetailsImpl

```
public class UserDetailsImpl implements UserDetails{

    private static final long serialVersionUID = 1L;

    private User user;

    private boolean locked = false;
    private boolean expired = false;
    private boolean enabled = true;

    private Collection<? extends GrantedAuthority> authorities;

    public UserDetailsImpl(User user, Collection<? extends GrantedAuthority> authorities)
{
```

```
        this.user = user;
        this.authorities = authorities;
    }

    public static UserDetailsImpl build(User user) {
        List<GrantedAuthority> authorities = user.getRoles().stream()
            .map(role -> new SimpleGrantedAuthority(role.name()))
            .collect(Collectors.toList());

        return new UserDetailsImpl(
            user,
            authorities);
    }

… getters and setters…
```

UserDetailsImpl існує для того, щоб прив’язати існуючих користувачів до контексту безпеки у Spring Security. Таким чином, будь-яка зміна в базі даних відображається на дозволах користувача без жодних проблем.

3.3 Реалізація парсування та видачі JWT

У Spring 5 була введена функція роботи сервера у якості генератора токенів OAuth 2.0, що базуються на JWT, за допомогою фреймворку Keycloak, проте використання цілого фреймворку для цих потреб витрачає багато часу розробки та сповільнює роботу застосування, тому було реалізовано системи для з’ясування цих проблем вручну. UsersAPI створює простий JWT токен, що не включає в себе жодних заяв – лише суб’єкт, підписанта та час закінчення роботи.

```
public class JwtUtils implements Serializable{

    private static final long serialVersionUID = 1L;
    private static final long VALIDITY = 3600;

    @Value("jwt.secret")
    private String secret;

    public String getUsername(String token) {
        return JWT.decode(token).getSubject();
    }

    public Date getExpiration(String token) {
        return JWT.decode(token).getExpiresAt();
    }

    public String generate(UserDetails userDt) {
```

```
        return JWT.create()
            .withIssuer("auth")
            .withSubject(userDt.getUsername())
            .withIssuedAt(new Date())
            .withExpiresAt(new Date(System.currentTimeMillis() + VALIDITY*1000))
            .sign(Algorithm.HMAC512(secret));
    }

    public boolean validate(String token, UserDetails userDt) {
        JWT.require(Algorithm.HMAC512(secret)).withIssuer("auth").build().verify(token);
        return getUsername(token).equals(userDt.getUsername());
    }
}
```

Клас JwtUtils існує для реалізації процесу створення JWT токену, а також їх розшифрування та перевірки.

```
public class JwtRequestFilter extends OncePerRequestFilter{

    @Autowired
    private UserDetailsServiceImpl service;

    @Autowired
    private JwtUtils utils;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse respon
se, FilterChain filters)
            throws ServletException, IOException {
        final String requestTokenHeader = request.getHeader("Authorization");
        UserDetails user = null;
        if (requestTokenHeader!=null && requestTokenHeader.startsWith("Bearer ")) {
            String nickname = utils.getUsername(requestTokenHeader.substring(7));
            user = service.loadUserByUsername(nickname);
        }
        UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new User
namePasswordAuthenticationToken(user, null,
            user.getAuthorities());
        usernamePasswordAuthenticationToken.setDetails(new WebAuthenticationDetailsSource(
).buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticatio
nToken);
        filters.doFilter(request, response);
    }
```

JwtFilterRequest – це фільтр, що застосовується для всіх вхідних запитів через SecurityConfig (окрім “/login” та “/register”). Він перевіряє наявність токену JWT та оновлює контекст сесії до рівня користувача, юзернейм котрого надано в цьому токені.

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    … Autowired values…

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf()
            .and()
              .authorizeRequests()
                .antMatchers(HttpMethod.POST, "/login", "/register")
                    .permitAll()
                .anyRequest()
                    .authenticated()
            .and()
                .exceptionHandling()
                    .authenticationEntryPoint(entry)
            .and()
                .sessionManagement()
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
            http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
    }
```

SecurityConfig – це клас, в якому вказано зміни до звичайної політики безпеки Spring Security. Наданий приклад сконфігуровано таким чином, щоб:

- Дозволити запити Cross Site Request Form – оскільки сайт опрацьовує запити від іншого серверу, технічно усі запити надходять з іншого сайту, тому цей дозвіл є необхідним.
- Дозволити усі POST запити до “/login” та “/register” – це відкриті точки доступу, що дозволяють почати взаємодію з сервером.
- Заборонити неавтентифікований доступ до усіх інших точок доступу.
- Заборонити створення сесій – автентифікація відбувається лише за допомогою JWT
- Додати описаний вище фільтр до усіх запитів

Висновки

Темою цієї роботи була перевірка можливості та доцільності застосування існуючих технологій розробки веб-застосувань з Docker для малих команд на власному досвіді розробки багаторівневого веб-застосування «сайт новин». На думку автора, не буде помилкою стверджувати, що цей експеримент був поганою ідеєю.

Обрані методи та технології розробки призвели до наступних проблем у розробці:

- Підвищення складності розробки в декілька разів унаслідок ізоляції сервісів та потреби створення додаткових точок доступу для об’єднання фрагментованих даних – перетворення комбінації фронтенд + бекенд на два фронтенди та два бекенди
- Підвищення складності структур та, як наслідок, часу який вони вимагають для запуску та обміну інформацією
- Створення вимоги до використання більш складних технологій обміну інформацією між частинами застосування – без використання мікросервісів, не виникло б потреби у використанні JWT, що дуже добре б позначилось на часі розробки

Переваги від використання технології є, але вони є перспективними, а не реальними – переваги у портативності дуже складно відчути за збереження розміру команди в одну особу, а у масштабуванні – без закінчення розробки застосування та його несподіваної популярності.

Отже, висновок з цієї роботи є таким – Docker та архітектура мікросервісів, для якої він переважно застосовується, є корисними для команд великого та середнього розміру – команд, що можуть бути поділені на менші, тісні команди, кожна з яких буде працювати над окремим бізнес-процесом. Для малої команди (особливо команди з одного чоловіка), переваги Docker є невеликими, а проблеми з використанням архітектури мікросервісів – руйнівними.

Список використаної літератури

1. What is a container: офіційний сайт Docker [Електронний ресурс]: стан на 5 квітня 2021 // Режим доступу: https://www.docker.com/resources/what-container
2. Understanding the Docker internals [Електронний ресурс] : стаття від 7 січня 2017 / Nitin Agarwal // Режим доступу: https://medium.com/@BeNitinAgarwal/understanding-the-docker-internals-7ccb052ce9fe
3. Docker Internals [Електронний ресурс] : блог пост від 29 лютого 2016 / автори Docker Saigon // Режим доступу: http://docker-saigon.github.io/post/Docker-Internals/
4. Pods: офіційний сайт Kubernetes Docker [Електронний ресурс]: стан на 5 квітня 2021 // Режим доступу: https://kubernetes.io/docs/concepts/workloads/pods/
5. What is Docker and why is it so darn popular? [Електронний ресурс] : стаття від 27 березня 2018 / Steven J. Vaughan-Nichols // Режим доступу: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/
6. Industry Leaders Unite to Create Project for Open Container Standards: офіційний сайт Docker [Електронний ресурс]: пост від 22 червня 2015 / PR департамент Docker // Режим доступу: https://www.docker.com/docker-news-and-press/industry-leaders-unite-create-project-open-container-standards
7. What are microservices? [Електронний ресурс] : стан на 12 квітня 2021 / Chris Richardson // Режим доступу: https://microservices.io
8. What Are Containerized Microservices? [Електронний ресурс] : стаття від 25 лютого 2021 / Jeremy H // Режим доступу: https://blog.dreamfactory.com/what-are-containerized-microservices/
9. Seemlessly swapping the backend of the Netflix Android app [Електронний ресурс] : стаття від 8 вересня 2020 / Rohan Dhruva, Ed Ballot // Режим доступу: https://netflixtechblog.com/seamlessly-swapping-the-api-backend-of-the-netflix-android-app-3d4317155187
10. Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow [Електронний ресурс] : стаття від 5 вересня 2015 / Einas Haddad // Режим доступу: https://eng.uber.com/service-oriented-architecture/
11. Building Products at SoundCloud—Part III: Microservices in Scala and Finagle [Електронний ресурс] : стаття від 13 червня 2014 / Phil Calçado // Режим доступу: https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-3-microservices-in-scala-and-finagle
12. How we build microservices at Karma [Електронний ресурс]: стаття від 4 квітня 2016 / офіційний акаунт Karma // Режим доступу: https://blog.karmawifi.com/how-we-build-microservices-at-karma-71497a89bfb4
13. Self-contained service [Електронний ресурс]: стан на 5 квітня 2021 / Chris Richardson // Режим доступу: https://microservices.io/patterns/decomposition/service-per-team.html
14. Decompose by subdomain [Електронний ресурс]: стан на 5 квітня 2021 / Chris Richardson // Режим доступу: https://microservices.io/patterns/decomposition/decompose-by-subdomain.html
15. Decompose by business capability [Електронний ресурс]: стан на 5 квітня 2021 / Chris Richardson // Режим доступу: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html
16. Service per team [Електронний ресурс]: стан на 5 квітня 2021 / Chris Richardson // Режим доступу: https://microservices.io/patterns/decomposition/service-per-team.html