ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра комп‘ютерних наук

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр

на тему: «РОЗРОБКА ПЛАТФОРМИ ДЛЯ ЕЛЕКТРОННХ ПУБЛІКАЦІЙ НА МОВІ ПРОГРАМУВАННЯ CLOJURE »

Виконав: студент 4 курсу, групи КНД–42
спеціальності
122 ____Комп’ютерні науки_____
(шифр і назва спеціальності)
_____Богдан М.П._____
(прізвище та ініціали)
Керівник Василенко В.В._____
(прізвище та ініціали)
Рецензент_____
(прізвище та ініціали)

Київ – 2021

ВСТУП

Актуальність дослідження. Розробка програмного забезпечення розвивається дуже швидкими темпами. Сьогодні, з приходом глобалізації треба забезпечити доступність свого застосунку в будь-якій точці земної кулі. Цей фактор спонукає використовувати хмарні рішення, що висувають значні вимоги до застосунку, зокрема вимоги до відтворюваності і переносимості.

З іншого боку популяризація інформаційних технологій призвела до полегшення інформатизацію компаній, створення своїх он-лайн бізнесів, а це призвело до посилення конкуренції. В таких умовах треба щоб застосунок міг швидко розвиватися та забезпечував надійну роботу на різних навантаженнях.

Об'єкт дослідження – процес розробки програмного продукту, що відповідає сучасним умовам на прикладі додатку для електронних публікацій

Предмет дослідження – технології та методи розробки сучасних веб-застосунків.

Мета роботи – розробити платформу для електронних публікацій, що відповідає вимогам сучасних веб-застосунків

Наукові завдання:

дослідити сучасні архітектури проектування програмних продуктів;

дослідити варіанти розгортання готового додатку;

дослідити інструменти розробки програмного забезпечення;

розробити програмний продукт, що використовує сучасну архітектуру та використовує новітні інструменти розробки програмного забезпечення;

розробити конфігурацію для системи розгортання програмного продукту на хмарній платформі.

Методи дослідження – опрацювання літератури за розробкою, експлуатацією й архітектурою програмних продуктів, аналіз та порівняння інструментів розробки програмних продуктів, розробка програмного продукту для електронних публікацій.

Практичне значення одержаних результатів полягає в створенні рекомендацій щодо створення та розгортання програмних продуктів, а також створення платформи електронних публікацій на мові програмування Clojure.

1 АНАЛІЗ ВИКЛИКІВ ПРИ ПОБУДОВІ СУЧАСНОГО ВЕБ-ЗАСТОСУНКУ

1.1. Аналіз проблеми внесення змін

Сьогодні ринок інформаційних технологій розвивається стрімко. Багато компаній відкриваються і закриваються, поглинаються, зливаються та купуються кожного дня змінюючи баланс на ринку. Уже не досить зробити гарний продукт, як це було ще нещодавно, гірші продукти, що вийшли раніше чи швидше розвиваються, мають більше шансів вижити. Інновації та зміни йдуть пліч-о-пліч та сприяють розвивитку людства.

Сучасні методології та фреймворки розробки, такі як SCRUM або Agile, передбачають часті невеликі зміни продукту. Це дозволяє швидше отримувати зворотній зв’язок від користувачів, дешевше і частіше випробовувати нові ідеї, а як результат – зменшувати ризик для бізнесу. В свою чергу такий підхід кидає виклики до інженерів, які створюють і підтримують код застосунку.

Змінення коду спричиняє наступні негативні ефекти:

- поява старого коду;
- ускладнення логіки;
- збільшення вартості внесення змін;
- зменшення надійності системи;
- складність підтримки;
- збільшення вартості обслуговування;
- та інше.

Такі виклики дуже часто вирішуються цілим комплексом рішень:

- використання фреймворків;
- своєчасний рефакторінг;
- використання паттернів.

Одним з базових принципів, який застосовується як для проектів з єдиною кодовою базою та застосунком (моноліт), так і для розподілених систем з багатьма сервісами, є слабке зв’язування (Loose Coupling). Цей підхід дозволяє

максимально гнучко створювати системи будь-якої складності, залишаючі можливості для легкої зміни продукту.

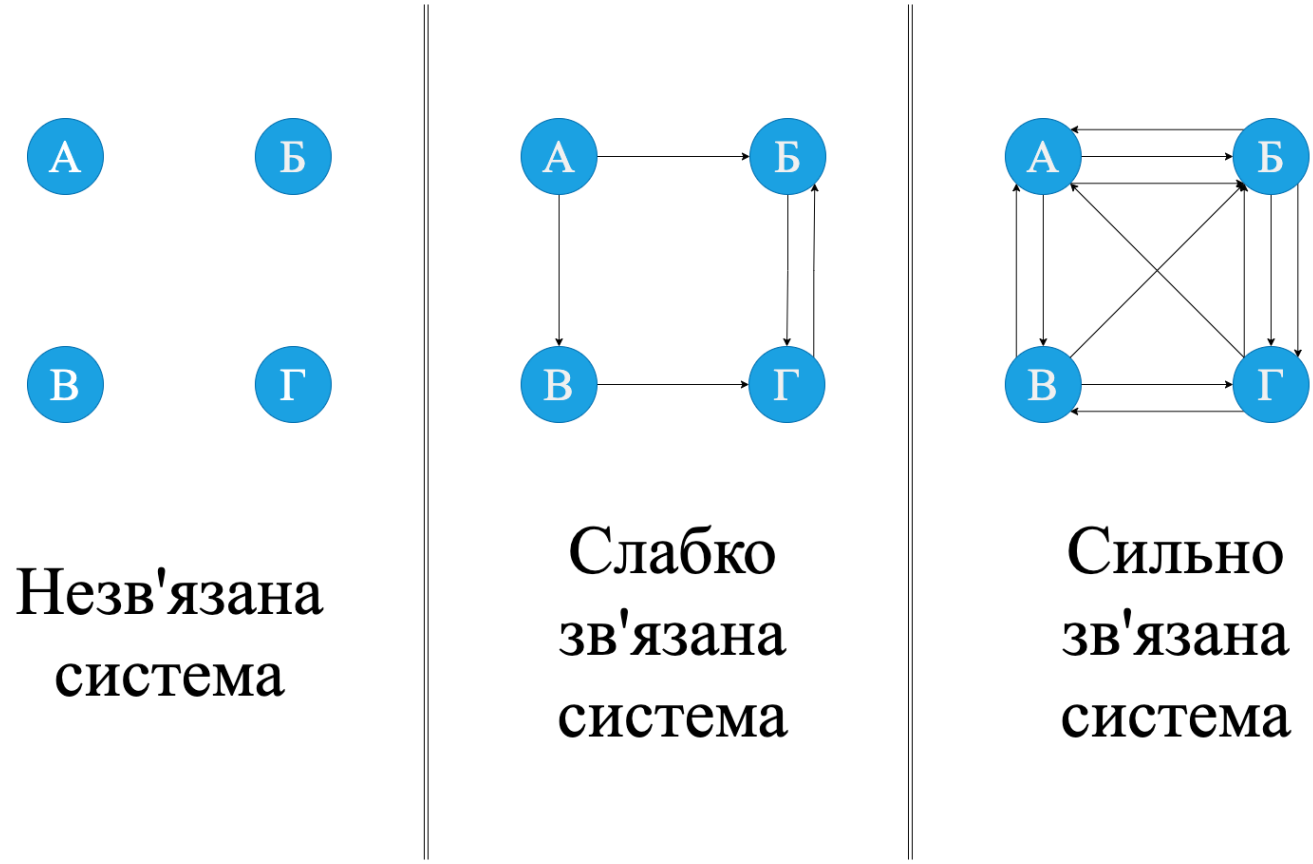

Рис. 1.1. Види зв’язаності систем

Розглядаючи різні види систем, які приведені на рис. 1.1, легко побачити, що кількість зв’язків, а значить і кількість місць, що потребують змін значно менша, і це дає нам більш гнучкий, легший у підтримці код, кожен компонент якого практично можливо розглядати у відриві від інших. Ця особливість є ключовим фактором при створенні сучасного ПЗ, адаптованого для існування в реаліях гнучких методологій розробки.

1.2. Сучасна архітектура програмних продуктів

Вимога слабкої зв’язаності висуває також вимоги і до архітектури проекту. Для забезпечення даних вимог є багато підходів та патернів розробки з різним ступенем свободи. Найбільш поширеною є багатошарова архітектура, яку описав Роберт Мартін. Він назвав цей підхід Clean Architecture, тобто Чиста Архітектура.

Ця архітектура поєднує у собі багато підходів що існували і до цього, але не є прив’язаною до якоїсь конкретної парадигми програмування. Серед рішень, що об’єднує в собі Чиста Архітектура можна виділити наступні:

- Hexagonal Architecture (гексагональна архітектура), автор Алістар Кокбурн
- Onion Architecture (багатошарова архітектура), автор Джефрі Палермо
- BCE (boundary-control-entity), автор Івар Якобсон

Всі ці рішення мають в основі розділення коду на компоненти і розподілення повноважень між ними. Майже кожне рішення є реалізацією ідеї відділення логіки програмного забезпечення від взаємодії з навколишнім світом.

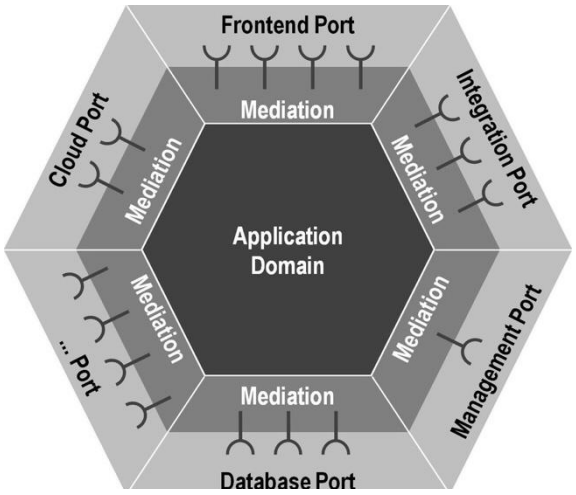

Рис. 1.2. Гексагональна архітектура

Якщо розглядати гексагональну архітектуру (рис 1.2) детальніше, то ми побачимо, що в її основі лежить ідея адаптерів і портів. В даному випадку адаптери це програмні компоненти, що інкапсулюють у собі логіку роботи з зовнішніми компонентами, базами даних, сервісами, користувацькими інтерфейсами і т.д.. У свою чергу порти – це інтерфейси для внутрішньої логіки для взаємодії зі світом. Такий підхід дозволяє абстаргуватися від конкретної реалізації зовнішньої залежності, щоб внутрішня логіка була якомога чистішою. Наприклад, якщо в нас зміниться версія сервісу, що ми використовуємо, або якщо ми вирішимо перейти з одної бази на іншу, всі наші зміни повинні бути обмежені кодом адаптера, у свою чергу логіка застосунку і порту є незмінною. Ця ідея є

дуже зручною і дає нам досить багато, але в ній не описано як повинна бути побудована логіка програмного продукту.

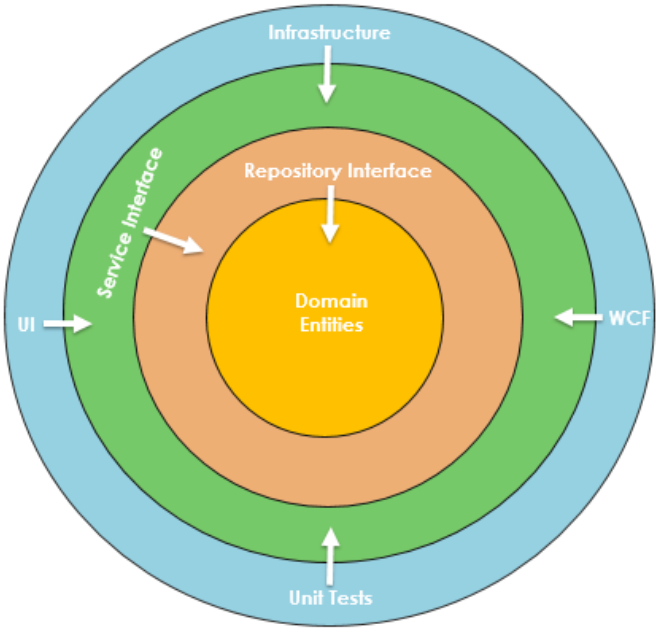

Рис. 1.3. Багатошарова архітектура

Багатошарова архітектура зазвичай протиставляється гексагональній. В цій ідеї вже є готові рішення для управління логікою застосунку. Застосунок зазвичай розділений на декілька шарів:

- Domain entities – шар, який містить у собі структури даних і логіку поведінки модельованої предметної області. Цей шар не повинен містити ніяких зовнішніх залежностей.
- Repository Interface – шар, що є клеєм між моделлю і бізнес логікою застосунку, може також містити взаємодію з базою даних.
- Service Interface – шар, що поєднує взаємодію між бізнес логікою і зовнішніми інтерфейсами
- API Layer – шар, що дає доступ зовнішньому світу до застосунку.

В цій моделі ми більш детально відокремили шар бізнес логіки від шару, що взаємодії з предметною областю. Проте є і недоліки, зокрема, неможливість замінити базу даних, або частково винести логіку в інший сервіс, замінити сервіс, що вже використовується, або ж обновити версію програмного інтерфейсу. В цьому підході зберігання даних є однією з основних операцій, а сервіси, що використовує наш продукт, частиною одного із внутрішніх шарів.

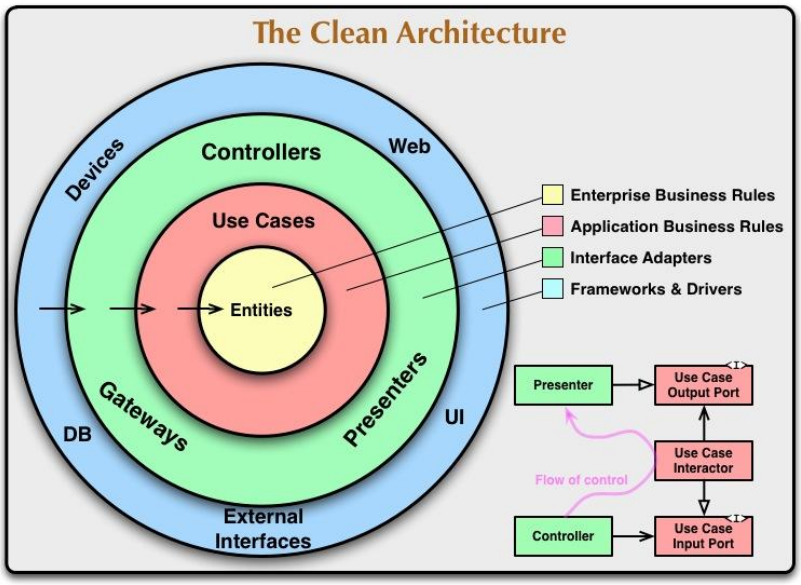

Рис. 1.4. Чиста архітектура

Розглядуючи більш детально чисту архітектуру (рис 1.4), ми побачимо, що вона позбавлена недоліків, що притаманні її ідейним попередницям. На відміну від багатошарового підходу всі залежності мають рівний пріорітет, як власний користувацький інтерфейс, так і робота з базами даних і зовнішніми сервісами, що притаманно гексагональній архітектурі. У порівнянні з останньою є чіткий підхід до організації внутрішньої логіки, який навіть є більш детальним.

Відповідно до діаграми шари відповідають за наступні частини:

- Entities – це структури даних і логіка що описують предметну область на високому рівні та змінюються дуже рідко.
- Use Cases – бізнес логіка, що описує конкретні ситуації і відповідні дії над предметною областю, змінюється не часто.
- Interface adapters – шар що відповідає за трансформацію даних від зовнішніх джерел в потрібні для використання бізнес логікою
- Frameworks and Drivers – шар який працює з зовнішнім світом: базами даних, користувацькими інтерфейсами, сторонніми сервісами, драйверами та іншим.

1.3. Вимоги інфраструктури до коду застосунку

Окрім частих змін також існує проблема експлуатації програмного забезпечення. Застосунок повинен уміти працювати як на комп'ютері розробника, тестувальника, так і на тестовому сервері, або в хмарній екосистемі. На кожному з них можуть бути установлені різні програми, драйвери, операційні системи, і все це не повинно впливати на те, як працює код.

Для рішення даної задачі розробниками хмарної платформи для запуску веб-застосунків Heroku були створенні 12 правил, названі 12 факторів. В основу цієї проблеми покладено і книги Мартіна Фаулера, який пропрацював в індустрії багато років і є знаним фахівцем, а також автором багатьох книг з архітектури програмних продуктів.

Дванадцять факторів:

1. Єдина кодова база для спрощення версіонування коду.
2. Зовнішні залежності мають бути явно оголошені та ізольовані.
3. Кофігурація має отримуватись з середовища виконання.
4. Сторонні служби – це підключені ресурси.
5. Збірка, реліз та виконання мають бути чітко розділені між собою.
6. Застосунок має бути запущений як один або декілька незалежних процесів, що не зберігають внутрішній стан.
7. Експортування сервісу має відбуватися через прив'язку портів
8. Масштабування має відбуватися за допомогою процесів.
9. Застосунок має підтримувати коректне завершення.
10. Середовища для розробки, тестування та виконання мають бути максимально схожі між собою.
11. Журналювання має бути неперервним потоком подій
12. Адміністрування застосунку має виконуватись разовими процесами.

При виконанні всіх цих пунктів ми отримуємо продукт, який не залежить від конкретних налаштувань операційної системи, встановлених пакетів, може бути безпечно запущений і завершений, а також легко-відслідковуваний через журналювання подій.

1.4. Аналіз проблеми доставки та запуску програмних продуктів

В історії програмного забезпечення було доволі багато прикладів реалізації незалежності від оточення, «заліза» та інших факторів. Основним гаслом і рекламним слоганом для мови програмування Java під час її виходу було: «Write once, run anywhere», що натякає на переносимість коду на JVM між різними реалізаціями «заліза». Віртуальна машина добре справлялася з проблемою залежності від апаратної платформи та частково брала на себе рішення проблем сумісності програмної, але повністю не вирішувала її.

Для рішення проблеми відтворюваності були створені контейнери. Їх існує досить багато: Zones (Solaris), Jails (BSD), rkt (Linux), Windows Containers та інші. Найпопулярнішою технологією на даний момент є Docker. Його портували на багато операційних систем, і зараз він доступний на всіх популярних ОС. Для роботи контейнеру потрібно ядро Linux, тому на інших системах він працює за рахунок віртуалізації.

Основні можливості контейнерів на основі Docker:

- Ізолювання середовища виконання включно з різноманітним набором сторонніх системних бібліотек потрібних для функціонування
- Незалежність від файлової системи, оскільки використовується своя
- Ізоляція файлової системи – процес має доступ тільки до явно вказаних файлів
- Ізоляція ресурсів – контейнер не може споживати більше ресурсів (процесор, оперативна та постійна пам’ять та інше) ніж вказано
- Ізоляція мережі – процес має доступ тільки до дозволеного простору імен та портів.
- Всі стандартні потоки виводу зберігаються як журналювання автоматично
- Налаштування одного контейнера може бути використана як базовий образ для інших контейнерів
- Інтеграція з багатьма хмарними провайдерами.

- Зберігання образів в централізованих репозиторіях.
- Один спільний публічний репозиторій для образів
- Підтримка приватних репозиторіїв

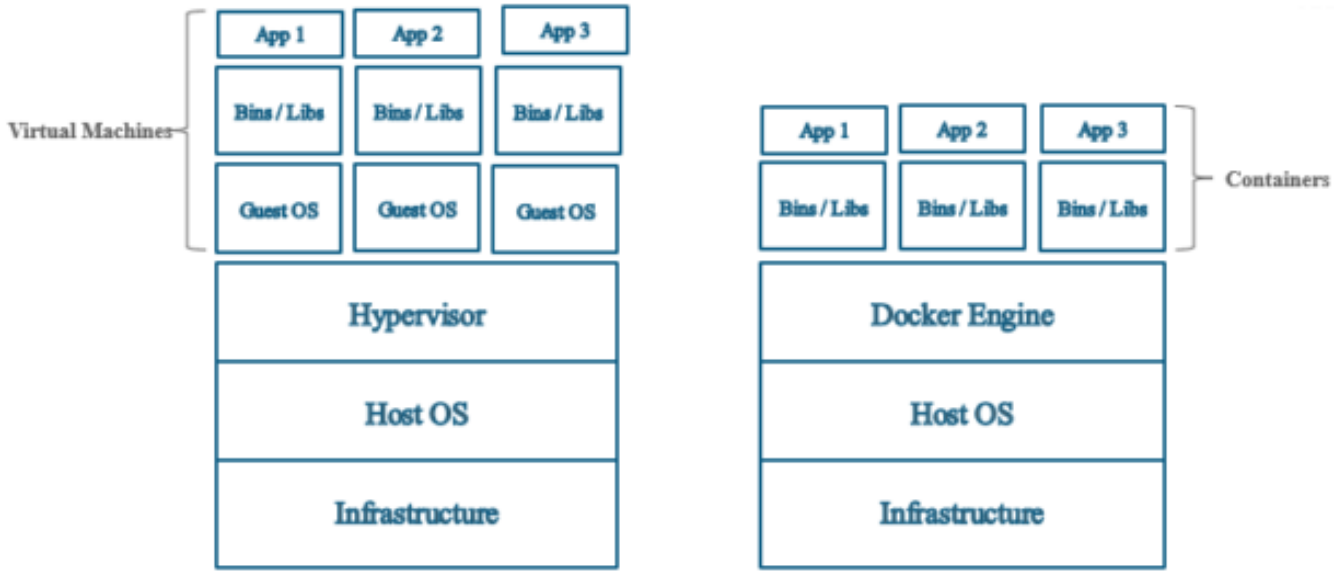

Рис. 1.5. Порівняння віртуальної машини і контейнерів

В порівнянні з віртуальними машинами, які використовувались раніше контейнери не мають потреби у власній операційній системі (рис. 1.5), а значить потребують менше ресурсів і можуть швидше запускатись.

1.5. Оркестрація контейнерів

Маючи можливість відтворювано запускати наш програмний продукт, у нас залишилось ще декілька невирішених проблем, а саме:

- Кожний контейнер запускається вручну, а параметри передаються як аргументи у командному рядку
- Мережа між контейнерами повинна налаштовуватись для кожного окремо, і налаштування не зберігаються під час перезапуску
- Ми не можемо автоматично розгортати додаткові контейнери відповідно до навантаження
- Процес розгортання застосунку все ще мануальний і потребує багато часу
- Система виявлення сервісів

Для рішення цих проблем існують системи оркестрації контейнерів. На данний момент однією з найросповсюджених є Kubernetes. Це платформа з відкритим кодом, що початково була розроблена в компанії Google. Вона має підтримку більшості хмарних провайдерів (Google Cloud, Amazon Web Services, Microsoft Azure) і легко розгортається на них.

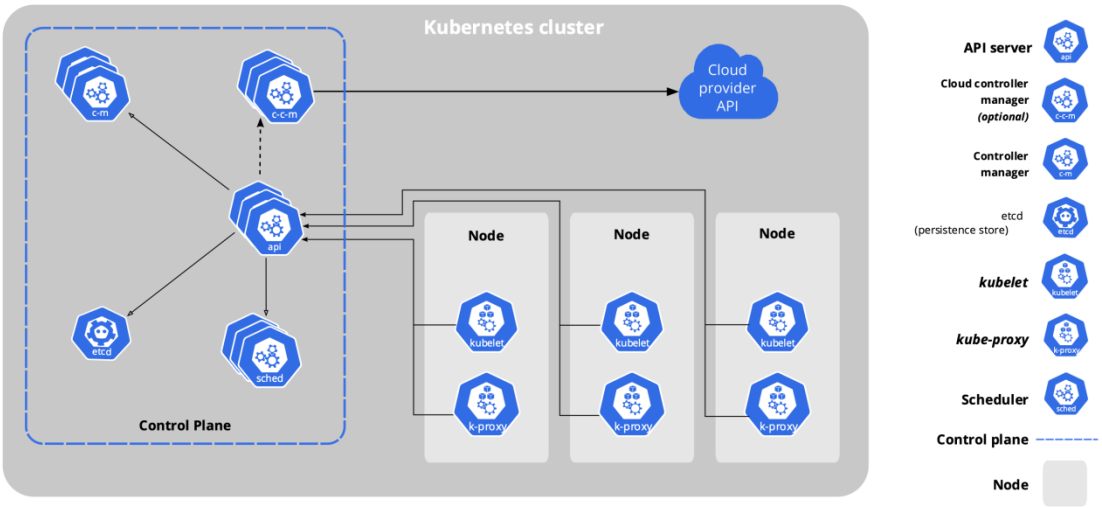

Рис. 1.6. Структура Kubernetes кластеру

На рис. 1.6 представлений типовий кластер на основі Kubernetes. Node на даному рисунку представлені віртуальні машини, які ми беремо в оренду у хмарної платформи. Навантаження між ними буде розподілено автоматично, тобто при розгортанні контейнера система самостійно вирішить відповідно до потрібних ресурсів, де можна розгорнути його. Kubelet і K-proxy, що є на кожній ноді, це системні сервіси, які відповідають за її експлуатацію та мережу відповідно. За розгортання нових контейнерів і виділення ресурсів відповідає сервіс sched, а за взаємодію з платформою – Cloud Controller Manager.

Kubernetes має чимало готових сценаріїв: від періодичних задач до постійно працюючих сервісів, від безперебійного розгортання до автоматичного масштабування. Якщо ж для застосунку не вистачає сценаріїв, його завжди можна дописати самостійно завдяки розгорнутому програмному інтерфесу.

Висновки до розділу 1

Розробка програмного забезпечення в сучасному світі досить складне завдання. Вже недостатньо створити та випустити ПЗ, його ще необхідно розвивати та підтримувати, запускати на різних оточеннях.

Для вирішення всіх цих викликів є багато альтернатив, готових архітектур і патернів проектування та програмування. Модульні підходи до розробки роблять ітеративну побудову застосунку простіше та абстрагують бізнес логіку від конкретних реалізацій, що в свою чергу дозволяє концентруватися на одній справі одночасно.

З поширенням контейнерних технологій зникла проблема відтворюваності і налаштувань на кожному окремому оточенні. Це також пришвидшує як розробку, так і впровадження, але і кидає виклик управлінням, або оркестрацією окремих одиниць.

Для керування контейнерами існує Kubernetes та подібні йому рішення. Вони дозволяють побудувати цілі кластери, налаштувати в них мережу, правила доступу, автоматичне розгортання, оновлення та інше. І це все лише завдяки текстовим файлам з конфігурацією, замість купи команд в терміналі.

2 ВИБРАНІ ІНСТРУМЕНТИ ДЛЯ РОЗРОБКИ

2.1. Мова програмування Clojure

Не зважаючи на те, що майже всі існуючі мови програмування можуть вирішувати будь-які задачі, особливо, якщо ми говоримо про популярні мови, деякі класи задач вирішувати не на всіх з них однаково комфортно. Наприклад, якщо наша задача розробити модель для машинного навчання, буде легше зробити на Python чи R завдяки великій кількості бібліотек та фреймворків. Якщо нам треба створити інструменти для цієї ж задачі, вище зазначені мови не підходять, оскільки не мають потужних засобів для низкорівневих оптимізації, тому для цієї задачі краще підійде C++.

Для чого ж використовується Clojure? Які переваги в нього і які класи задач вирішує? Ця мова програмування була створена програмістом на ім’я Річ Хіккі. Це інженер з багатим досвідом програмування на різних мовах: C++, C#, Java тощо. Метою створення було полегшити розробку складних веб-застосунків. Однією з ключових подій було знайомство з сім’єю мов програмування Lisp, а особливо з Common Lisp та Scheme. Ці мови були досить поширені в академічних колах, але не доволі слабко представлені у виробництві. Необхідно зауважити, що ці мови через свою непопулярность не мали достатнього набору бібліотек, а за рахунок таких вбудованих можливостей як макроси ще й робили програмування на них досить складним, адже завдяки їм кожен автор інструментів робив свою мову зі своїм синтаксисом. Це явище також відоме як «Прокляття Lisp». З іншого боку, з допомогою тих же макросів і функціонального підходу код був досить компактний і легким для розуміння.

Іншим фактором стало популяризація на початку 2000-х мов на віртуальних машинах, зокрема Java та C#. На той момент на платформі clr крім C# вже було досить багато мов таких як F#, VisualBasic, навіть J#, що був клоном Java. Всі ці мови за рахунок спільної середи виконання і компіляції в єдиний байткод могли використовувати всі бібліотеки для цієї платформи. Тепер створюючи свою мову

не треба було створювати багато інструментів для того, щоб нею можна було користуватися.

Отже, Clojure – це Lisp подібна мова, що має всі переваги та інструменти своєї платформи, а саме JVM, з спрямованістю на функціональне програмування. Також треба зазначити, що ця мова може бути скомпільована в декілька мов та байткодів інших платформ:

- ClojureCLR компілюється в байткод Common Language Runtime
- ClojureScript транспілюється в JavaScript (EcmaScript)
- ClojureErl компілюється в байткод для Beam, віртуальної машини Erlang

З допомогою вбудованих інструментів, таких як Reader Conditionals, код буде спільний для всіх платформ. Тобто можна використовувати один і той же код в користувацькому додатку і в серверній частині з усіма перевагами, які може дати JVM на сервері і JavaScript у браузері.

2.1.1 Незмінювані структури даних

Однією з найбільших складностей в програмуванні є взаємодія зі станом системи. Ця проблема пов’язана з однією з особливостей більшості популярних мов – змінними. Ще гіршою вона стала в конкурентних або багатопоточних застосунках, де багато паралельно запущених частин програми можуть змінити певну змінну. Це може спричиняти багато проблем:

- Стан перегонів (Race Condition)
- Псування даних
- Витік пам’яті
- Взаємне блокування (Deadlock)
- Та інші

Для вище описаних проблем на сьогодні існує багато готових рішень та підходів: семафори, м’ютекси, сигнали, планувальники виконання, тредпули. Були також створені об’єктно орієнтоване програмування, система акторів та інші. Всі ці засоби були націлені на локалізацію стану або на покращення контролю його зміни.

З іншого боку, існує функціональне програмування. У ньому з самого початку була концепція чистих функцій, які виконують обчислення тільки на основі даних, що були передані зовні і результатом їх виконання є теж дані. Ніяких зовнішніх ефектів на кшталт запиту ресурсів чи зміни стану програми, у них немає і бути не може. Замість зміни значення змінної використовується композиція функцій, і програма розбивається на маленькі частини, які легко зрозуміти, протестувати і для роботи яких не треба якийсь специфічний стан оточення. Такі функції можна запускати паралельно, оскільки ми можемо бути впевнені, що вони нічого не змінять, і це відкриває нові можливості в написанні конкурентного та багатопотокового коду.

Існує декілька зауважень до цього підходу:

1. Якщо функції не змінюють змінні та не мутують зовнішній стан (не чистять пам’ять), пам’ять може закінчитись.
2. Якщо ми передаємо досить велику структуру даних у функцію, і вона змінює тільки кілька елементів, чи не спричинить це копіювання всієї структури?

По-перше, саме в функціональній мові Lisp було створено перший збирач сміття
(GC), мета якого є збирання непотрібних змінних і видалення їх. У випадку Clojure роль такого GC на себе бере платформа JVM. Java, мабуть, найвдаліший вибір як платформи для вирішення цієї задачі. У цієї віртуальної машини існує система, яка дозволяє змінювати збирачі сміття в залежності від потреб, а також є досить багато готових рішень, як від Oracle, так і від сторонніх розробників.

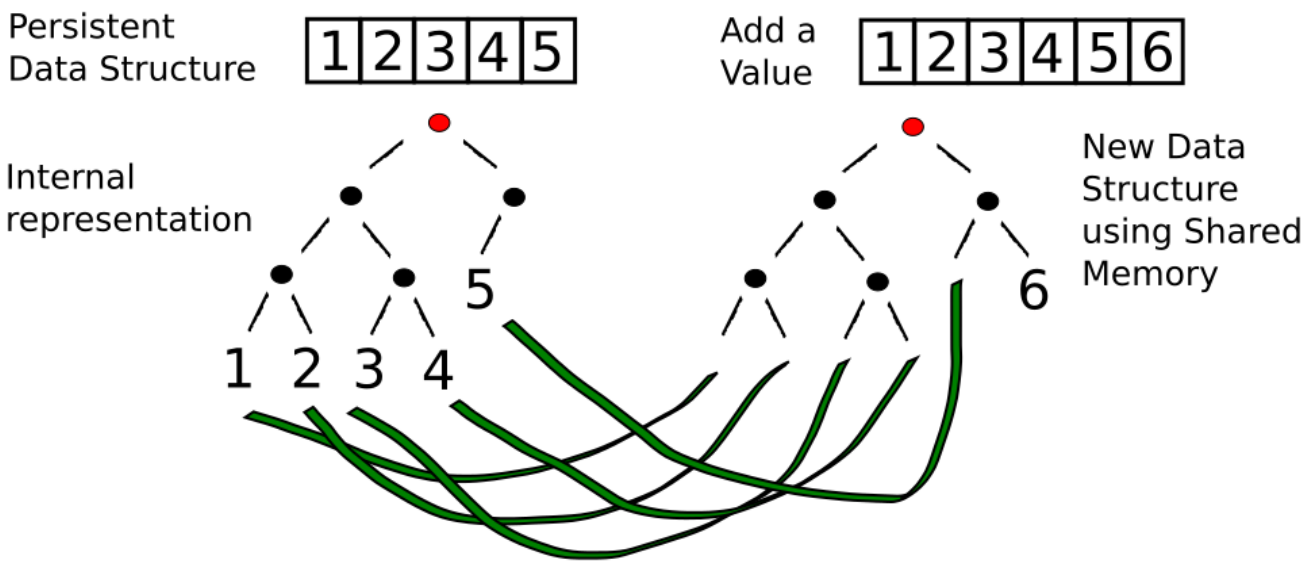

Рис. 2.1. Візуалізація внутрішньої реалізації структури даних.

По-друге, ми не повинні копіювати всі дані. Ми можемо зберегти посилання на вже існуючі частини. Це є безпечною операцією, оскільки можемо бути впевнені, що початкова структура не зміниться. В прикладі, показаному на рис. 2.1, буде повністю скопійована гілка, що містить значення 1, 2, 3, 4 і тільки корньовий вузол і контейнер для 5 будуть збудовані заново відповідно до доданих даних.

Одним з перших у питанні ефективних структур даних був Кріс Окасакі, який описав багато ефективних підходів і алгоритмів для зберігання і модифікування персистентних даних в своїй книзі «Purely Functional Data Structures». У свою чергу при створенні Clojure Річ Хіккі пішов ще далі – для всіх структур даних, що є в мові, використовується майже однакова реалізація, заснована на високоефективній структурі Hash Array Mapped Trie. У результаті ми маємо ефективний, як з точки зору пам’яті, так і з точки зору використання ресурсів центрального процесора, підхід для зберігання, доступу та обробки даних, що вбудований прямо в мову програмування.

2.1.2 Інтерактивна розробка

Останньою, але не за значенням, являється інтерактивна розробка, що є особливістю мов сімейства Lisp, ще з моменту їх створення. Досягається це

завдяки циклу читання з інтерактивного інтерфейсу, виконання і виведення користувачу відомому як REPL. Багато сучасних мов програмування мають сьогодні цей механізм, але в Lisp мовах він досяг найпотужнішого втілення. Завдяки йому можна приєднатися до запущеної програми і редагувати її без повного перезапуску.

Clojure, як представник цього сімейства, має повноцінний REPL і ,мабуть , єдиною мовою на платформі JVM, що підтримує інтерактивну розробку, як на етапі написання, так і при експлуатації. Для цього в мові існують досить різноманітні інструменти:

- Вбудований в мову інтерактивний термінал, що запускає сесію в терміналі;
- Бібліотека, яка відкриває мережевий порт і слухає його, очікуючи нові команди;
- Бібліотека, що відкриває HTTP з’єднання і підтримує авторизацію;
- Бібліотека для ClojureScript, яка відкриває websocket з’єднання в браузер і запускає транспільований в JavaScript код прямо в браузері.

Даний інструмент відкриває нові можливості при написанні коду. Цей процес перетворюється з того, що ми взаємодіємо з компілятором, тобто пишемо певний код, намагаємось його скомпілювати, виправляємо помилки і нарешті запускаємо в безперервний цикл написання і запуску, експериментів з функціями, їх реалізацією та поступове вирощення програми.

2.1.3 Стабільна базова реалізація мови

Іншою особливістю Clojure є стабільна базова бібліотека і синтаксис. Завдяки потужній системі макросів і можливості писати кожному свій синтаксис, ядро мови є досить компактним і не мало змінюється з часом. Це є суттєвою перевагою перед конкурентами, в яких з часом може змінюватися синтаксис, або назви стандартних інструментів. Код написаний 10 років тому буде працювати і зараз у більшості випадків. Як приклад протилежного підходу можна привести Python чи Scala, в яких при переході з версії 2 на версію 3 треба змінювати кодову

базу, що є досить проблематично. Наприклад, для повної відмови від підтримки Python 2 розробникам знадобилося більше 11 років після випуску 3 версії, з 3 грудня 2008 по 1 січня 2020 року.

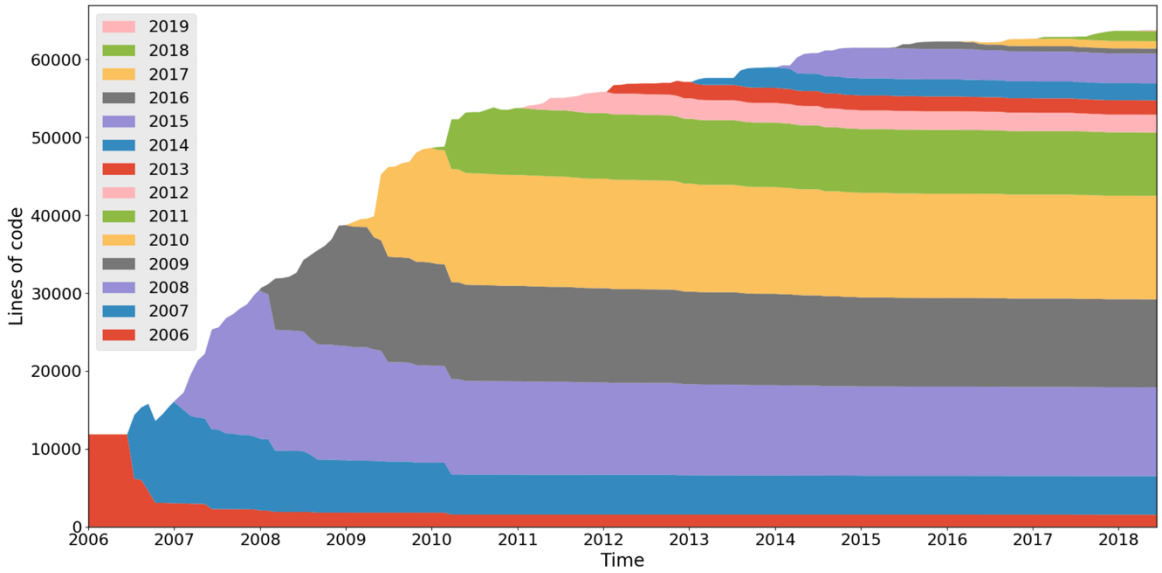

Рис. 2.2. Зміна кодової бази мови Clojure від часу

З рисунку 2.2 ми можемо бачити, що навіть в останньому релізі 1.10.3 можна знайти рядки коду з оригінальної версії, тут кольорами позначені рядки коду, що були написані у відповідні роки. Це дає неймовірну стабільність і можливість працювати з бібліотеками майже не зважаючи на їх вік. Трьохрічна бібліотека, яка не оновлювалась, але активно використовується, тут досить поширене явище. В порівняння можна привести подібний графік розвитку мови Scala на рис. 2.3. Як можна побачити код частіше змінювався і скоріш за все поставало питання сумісності версій, але і розвиток мови більш стрімкий.

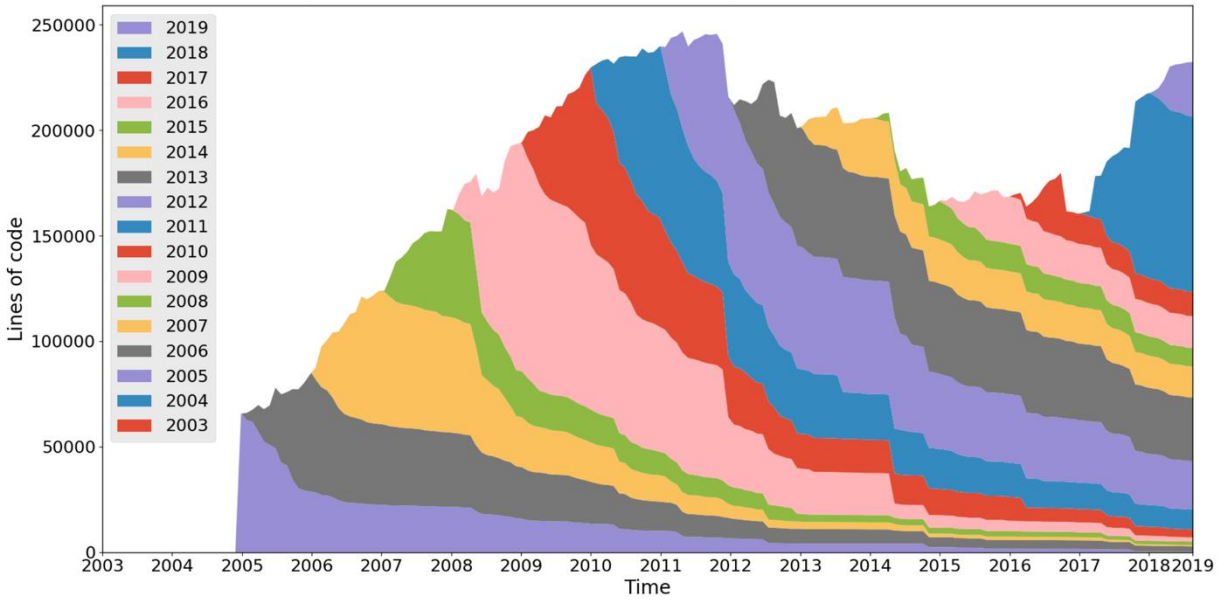

Рис. 2.3. Зміна кодової бази мови Scala від часу

Іншою перевагою такої стабільності і гнучкої системи макросів є те, що вона сама себе підтримує. Наприклад в Python, щоб створити підтримку асинхронного вводу-виводу прийшлося змінювати мову і вводити новий синтаксис. До цього існували бібліотеки для таких задач, але вирішували проблему в досить громіздкий спосіб. Це створило несумісність коду, який був написаний для нової і старої версій, та чітку залежність розвитку мови і бібліотеки асинхронного коду. У порівнянні, у Clojure є декілька конкуруючих бібліотек для цієї ж задачі, всі вони приносять свій синтаксис завдяки макросам, але розвиваються паралельно з мовою. Бібліотека не залежить від версії мови і не має чекати коли представлять новий синтаксис з оптимізаціями. Замість цього автори можуть незалежно розвивати її самостійно.

2.2. Модульний фреймворк для керованої даними архітектури Duct

Другою проблемою при створенні веб-застосунку є вибір фреймворку. Безперечно, можна обійтися і без нього, використовуючи примітивні бібліотеки і створюючи архітектуру під себе, але це досить непроста задача, і, як правило, вирішуючи її, людина приходить до створення свого власного фреймфорку. Зазвичай в ньому зібрані якісь зручні підходи і кращі практики програмування, але є і недоліки у вигляді більшої кількості коду для обслуговування системи.

Зазвичай кращим рішенням є вибір готового фреймворку, що втілює в життя потрібні практики й архітектуру і не є громіздким у використанні. Саме таким для мене і став Duct. Це модульне рішення, а значить його базовий функціонал досить мінімальний, і я можу додавати потрібні модулі за мірою необхідності. Окрім того для його використання вистачає одного файла з конфігурацією, що також вирішує проблему зчитування змінних оточення, а ще його також можна розбити на декілька, з'єднавши за допомогою спеціальних посилань. У нього також вбудований функціонал ін'єкції залежностей, без якого майже не можливо створити модульну програму, а ще він відповідає вимогам 12 факторів, які ми описали в розділі 1.3.

2.2.1. Імплементація кордонів застосунку

Як було сказано в розділі 1.2, сучасний застосунок повинен підтримувати модульну архітектуру. В Duct це винесено на рівень фреймворку та є в базовій документації. Тут такі модулі називаються кордони, або баундарі. В Clean Architecture це зовнішне коло, яке відповідає за взаємодію з навколишнім світом та полегшує процес абстрагування внутрішньої логіки від імплементації конкретних сервісів, баз, протоколів і так далі. Внутрішня логіка не повинна бути складнішою ніж наприклад:

- Створи новий пост
- Онови пост
- Додай коментар
- Редагуй коментар
- Користувач вподобав пост

Тобто, реалізація цих інтерфейсів може приховувати один або декілька запитів в базу, інтеракцію з зовнішнім сервісом чи довільну комбінацію цих дій. Головне, щоб інтерфейс працював. Так розробник, що пише логіку на основі цих даних, може не думати як їх отримати, синхронно або асинхронно, чи треба тут використати синхронізацію ниток, або в якій таблиці все це лежить, який її розмір, та як оптимізувати запит. Все що потрібно від програміста в цьому

випадку це вирішити конкретну задачу, будь то пошук посту по тексту, що ввів користувач, чи пагінацію для виводу у відповіді додатку. Такий код можна легше зрозуміти і легше писати, він менш схильний до низкорівневих помилок, адже ви винесли їх у кордони, та його можна без наслідків передати молодшим розробникам, а їх код легше перевірити на код-рев'ю.

Для того, щоб створити такий баундарі в Duct треба зробити кілька кроків:

1. Створити протокол

```
(defprotocol BlogPost
  (create [this new-post])
  (update-text [this post-id new-text])
  (add-tag [this post-id tags])
  (add-comment [this post-id comment]))
```

2. Створити конфігурацію

```
[:platform.sql/boundary :boundary/blog-post] {:queries    #ig/ref :queries/blog-post
                                              :datasource $ig/ref :duct.database/sql}
[:platform.sql/query :query/blog-post]       {:sql-file "queries/blog_post.sql"}
```

3. Реалізувати для базового типу

```
(extend-protocol BlogPost
  SQLBoundary
  (create [{ds                    :datasource
            {:keys [insert-post]} :queries}
           post]
    (try+
      (insert-post ds {:post post})
      (catch Object _
        (throw+ {:type ::cannot-create-post
                 :ctx  {:post post}}
                (:throwable &throw-context)
                "Cannot create a new post."))))
  (update-text [{ds                         :datasource
                 {:keys [update-post-body]} :queries}
                post-id
                text]
    (try+
      (update-post-body ds {:post-id post-id
                            :text    text})
      (catch Object _
        (throw+ {:type ::cannot-update-post
                 :ctx  {:post-id post-id}}
                (:throwable &throw-context)
                "Cannot update post `%s`." post-id))))
  (add-tag [{ds                         :datasource
             {:keys [add-tag-to-post]} :queries}
            post-id
            tags]
    (try+
      (add-tag-to-post ds {:post-id post-id
                           :tags    tags})
      (catch Object _
        (throw+ {:type ::cannot-add-tag-to-post
                 :ctx  {:post-id post-id
                        :tags    tags}}
                (:throwable &throw-context)
                "Cannot add tag `%s` to the post `%s`." tags post-id))))
  (add-comment [{ds                         :datasource
                 {:keys [add-comment-to-post]} :queries}
                post-id
                comment]
    (try+
      (add-comment-to-post ds {:post-id post-id
                               :comment comment})
      (catch Object _
        (throw+ {:type ::cannot-add-tag-to-post
                 :ctx  {:post-id post-id
                        :comment comment}}
                (:throwable &throw-context)
                "Cannot add comment `%s` to the post `%s`." comment post-id)))))
```

4. Реалізувати ініціалізацію базового типу, а за потреби і весь життєвий цикл

```
(defrecord SQLBoundary [])

(defmethod ig/init-key :platform.sql/boundary
  [_ data]
  (map→SQLBoundary data))
```

5. Використати в коді

2.2.2. Інтерактивна розробка в Duct

Окрім самої мови інтерактивну розробку ще забезпечує й фреймворк. Вбудований REPL може тільки під'єднатися до існуючого процесу, редагувати перезаписувати функції, змінні, протоколи та інші примітиви мови, але не може керувати життєвим циклом застосунку. Duct може додати відсутній шар і керувати життєвим циклом компонентів. Допомагає цьому і модульна система, і побудовані кордони, і слабка зв'язаність. У базовій поставці фреймворку є модуль dev, що відповідає за інструменти розробки. В нього вбудовано декілька функцій, зокрема для мануального запуску, перезапуску та зупинки застосунку, для запуску тестів та функція зо запускає процес стеження на файлами вихідного коду. При зміні коду, конфігурації, або файлів с запитами до бази всі модулі описані в конфігурації будуть перезапущені і дані оновлені.

Також деякі моду можуть додавати свої аспекти до інтерактивної розробки. Так, наприклад, модуль для написання коду на ClojureScript має код, що запустить інструмент для гарячого перезапуску коду Figwheel, що також, як і Duct може відслідковувати зміну файлів, і оновлювати код, що працює в браузері. Для функціонування йому потрібна ще й встановлений компілятор ClojureScript в JavaScript, що є частиною інструменту і браузер, який підтримує веб-сокети і сучасні інструменти розробки.

Висновки до розділу 2

Вибір технологій при розробці застосунку є досить важливим етапом, від якого залежить не тільки комфорт розробників, але й швидкість розробки, складність у підтримці, а значить і кошти, що будуть потрібні для життєвого циклу програмного продукту.

Clojure є доволі практичною мовою, що дозволяє швидко розробляти додатки, використовуючи всі переваги платформ, на яких запускається, а також повторне використання коду на різних платформах. Іншою перевагою є використання функціонального підходу і незмінювані структури даних, що після

опанування знімають когнітивне навантаження з розробників, дозволяючи швидше писати робочій код. REPL та стабільність базової реалізації мови також полегшують процес розробки та підтримки, адже нові версії не руйнують сумісність коду. Відлагодити проблему, подивитися як працює якась функція чи модуль, або ж написати новий код легше в інтерактивному режимі.

Готові практики програмування й архітектури зібрані в готовому фреймворку полегшують розробку. Duct – гарний приклад такого підходу, який до того ж не потребує багато додаткового коду. Окрім того, цей інструмент ще й покращує процес розробки, додаючи в нього ще один шар інтерактивності.

3 ВПРОВАДЖЕННЯ ГОТОВОГО ЗАСТОСУНКУ

3.1. Проблема доставки готового застосунку на сервери

Розробити додаток – це тільки половина роботи, що потрібна для запуску проекту для користувачів. Ще є проблема доставки його на оточення, або сервери, на яких воно буде використовуватись і надавати сервіс користувачам. Існує багато можливостей запуску. Найпростіший – запакувати додаток у виконуваний файл та запустити на операційній системі серверу, але це не надійний варіант. По-перше, потрібно налаштувати систему, а параметри запуску буде відтворювати щоразу складніше. По-друге, буде розбіжність умов виконання на комп'ютері розробника та на кінцевій машині.

Усі ці недоліки повинна вирішувати система контейнеризації. Кожен контейнер є список необхідних бібліотек, програм та налаштувань, що потрібні для роботи додатку або його компоненту. Такий елемент є добре відтворюваним на будь-якій машині, що підтримує роботу з ними.

Для зручного запуску таких контейнерів використовуються системи менеджменту кластерів. Однією з найпопулярніших є Kubernetes. Ця система підтримується всіма найпопулярнішими хмарними платформами.

В Kubernetes для запуску додатків існує цілий набор ресурсів. Найменшим ресурсом є Pod. Він може містити декілька контейнерів, і це мінімальна одиниця для запуску на кластері. Для ідентифікації є система мета-даних, лейбів та просторів імен.

Є декілька стратегій використання подів:

1. Один контейнер на Pod
2. Декілька контейнерів на Pod, що працюють в синергії

Перша стратегія підходить майже всім застосункам. Слід брати до уваги, що Pod це мінімальний компонент, а це ощначає, що такий компонент може бути створений, запущений, зупинений у будь який момент, і якщо ми використовуємо контейнери для модуляризації проекту, використовуючи мікро-сервісний підхід в

проектуванні і розробці, то перезапуск Pod, що містить декілька незалежних модулів у різних контейнерах, призведе до припинення роботи всієї системи. Саме тому один контейнер на такий ресурс є оптимальною стратегією. Такий спосіб дозволяє досягнути безперервної роботи додатку.

Друга стратегія підходить для випадків, коли в нас є основний додаток, або модуль, та додаткові модулі. Такий патерн називається sidecar і використовується, наприклад, для адаптації логування основного контейнеру.

Також треба зазначити, що в реальному застосунку навряд чи треба бути створювати такий ресурс вручну, але треба знати як його створити, томущо урізаний варіант такого ресурсу є складовою частиною інших у якості шаблону для створення подів.

3.2. Проблема неперервної роботи застосунку

Рішення потреби неперервної роботи треба розділити на 2 питання:

1. Забезпечити неперервну роботи, якщо один з екземплярів вийде зі строю
2. Забезпечити неперервну роботу в момент оновлення версії застосунку

Для рішення першої проблеми достатньо запустити декілька копій застосунку і пересвідчуватись, що в кожен момент часу працюють всі екземпляри, а якщо один або декілька перестали – запустити відповідну кількість екземплярів або реплік. Це питання вирішується завдяки ReplicaSet **(TODO LINK).**

Друге питання вирішується за допомогою стратегій перезапуску. Одна з найпопулярніших – Rolling Update, яка оновлює екземпляри один за одним. Таким чином забезпечується безперервність, бо завжди є контейнер зі старою або новою версією, що працює.

Також треба вміти відповідати на навантаження. Тобто, якщо в даний момент часу майже ніхто не користується додатком, ми не хочемо платити за додаткові ресурси, але якщо користувачів стане більше вони не повинні довго очікувати, а значить, ми маємо своєчасно скорегувати кількість одночасно запущених екземплярів, а відповідно і виділити додаткові ресурси. Як результат

ми зможемо економити на ресурсах, що ми орендуємо у хмарного провайдеру та одночасно зберегти максимально швидкодіючу систему.

В Kubernetes вирішуються за допомогою Deployment **(TODO LINK)**. Цей ресурс, відповідає за життєвий цикл застосунку, його створення, оновлення і розподілення ресурсів. Як і для створення ReplicaSet йому треба шаблон Pod, щоб вміти його створювати. Між цими ресурсами також є і суттєва відміна: Deployment в своїй внутрішній реалізації використовує ReplicaSet, щоб створювати потрібну кількість контейнерів, та додає автоматичну зміну налаштувань останнього, щоб відповідати на друге питання, та на питання зміни навантаження. Цей ресурс є першим, що ми маємо описати повністю для запуску додатку з розглянутих.

3.3. Доступність до додатку

Оскільки Pod створюються динамічно, ми не можемо гарантувати, що їм буде виділений одна і та ж мережева адреса. Це означає, що навіть у кластері ми повинні мати якусь систему, яка б дозволила різним сервісам спілкуватися між собою та надати стабільний шлях для зовнішнього доступу до додатку. Для цього в Kubernetes існує така сутність як Service **(TODO LINK)**. Цей тип ресурсів є логічним і працює завдяки знаходження Pod за лейблами і перенаправлення даних від порта, який надає сервіс до порту, що слухає додаток.

3.4. Збереження даних, та управління змінними оточення

Так як контейнер і Pod, який його запускає, існує тільки обмежений період часу, а на ресурсах, що були виділені одному модулю застосунку може через деякий час бути запущений інший, нам треба вирішити проблему збереження даних. Більшість даних зараз зберігаються в базах даних, які в свою чергу надаються хмарним провайдером, але ми все ж маємо частину, яка зберігається локально, наприклад, файли журналювання подій. Ми можемо їх обробляти за допомогою паттерну SideCar **(TODO LINK)** запускаючи в кожному Pod додатковий контейнер, що буде якимось чином обробляти і зберігати їх в базу даних, але це додаткові ресурси.

Для рішення цього питання існує система постійних сховищ, що підключаються. Тобто ми маємо якийсь умовний накопичувач, який буде підключений до контейнерів, і після перезапуску всі данні будуть збережені, а при повторному запуску контейнера він буде підключений до такого ж сховища. Саме рішенням цього питання займаються ресурси PersistentVolume **(TODO Link)** та PersistentVolumeClaim **(TODO LINK)**. Перший описує сховище, його розмір політику збереження даних, тип, та створює відповідний запит на ресурс у хмарну платформу. Другий ресурс слугує для описання запиту контейнера на ресурс, щоб система могла динамічно розподілити ресурси між споживачами.

Іншою невирішеною проблемою є зберігання налаштувань, спільних для декількох компонентів застосунку, та зберігання секретних даних. Для першого існують ConfigMap **(TODO LINK)**, а для іншого Secret **(TODO Link)**. Обидва ці ресурси сберігають конфігурацію окремо від її користувачів, а Secret ще й робить це в зашифрованому вигляді.

Висновки до розділу 3

Kubernetes – дуже потужна система для запуску застосунків. Він підтримується великою кількістю провайдерів, що знімає необхідність ручного налаштування кластеру. Також інтерфейс, що ним надається, є досить гнучким і водночас зрозумілим. Для роботи є всі необхідні інструменти: від запуску окремо взятого контейнеру до розгортання надійної системи зі зберіганням даних і системою оновлення програмного продукту без його зупинки.

ВИСНОВКИ

В ході виконання бакалаврської роботи були отримані наступні висновки:

Для розробки веб-застосунку, що буде легкий в підтримці необхідна чітка модульна система організації коду, та архітектура, що їх забезпечує. Однією з найпопулярніших архітектур є The Clean Architecture.

Для запуску програмного продукту найкращий спосіб – використовувати контейнери, це дозволить досягнути стабільної відтворюваності і полегшіть розробку, відлагодження та супровід програмного продукту.

Для організації взаємної роботи різних контейнерів треба використовувати системи оркестрації та управління кластерами такі як Kubernetes. Це рішення має достатньо вбудованих функції для організації роботи веб-додатку, та може бути гнучко налаштоване.

В ході дослідження інструментів розробки опрацьовані матеріали щодо мови програмування Clojure, її бібліотек та фреймворків. Мова є гарною альтернативою існуючим рішенням, пропонуючи гнучкість, стабільність базової реалізації й інтерактивну розробку присутні сімейству Lisp, простоту функціональних мов, та потужність і ефективність незмінюваних структур даних заснованих на алгоритмі Hash Array Mapped Trie.

Розроблено застосунок для електронних публікацій на фрейморку Duct, що реалізує принципи модульної архітектури, розширює можливості інтерактивної розробки, та не потребує великої кількості обслуговуючого коду.

Створено конфігурацію для запуску веб-додатку на базі Kubernetes.