МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ

КАФЕДРА ІНФОРМАТИКИ

КУРСОВА РОБОТА

на тему:

РОЗРОБКА РОЗПОДІЛЕНОЇ СИСТЕМИ З АСИНХРОННОЮ КОМУНІКАЦІЄЮ З ВИКОРИСТАННЯМ SPRING WEBFLUX

Виконав: студент 3-го року навчання,

Спеціальності

121 «Інженерія Програмного

Забезпечення»

Буковський Станіслав Валерійович

Керівник

Андрощук Максим Віталійович

«14» травня 2024 р.

Київ 2024

Київ 2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“23” вересня 2024 року

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Буковському Станіслав Валерійовичу

факультету інформатики 3 курсу бакалаврської програми

1. Тема роботи «Розробка розподіленої системи з асинхронною комунікацією з використанням Spring Webflux», керівник роботи Андрощук Максим Віталійович

2. Строк подання студентом роботи 14 травня 2023

3. План роботи

Анотація

Вступ

Розділ 1. Реактивне програмування

Розділ 2. Spring WebFlux

Розділ 3. Огляд застосунку

Висновок

Календарний план виконання курсової роботи

№	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	23.09.2023	
2.	Складання плану курсової роботи	01.10.2023	
3.	Огляд літератури за темою роботи	07.11.2023	
4.	Проєктування архітекури застосунку	14.11.2023	
5.	Написання бекенд-частини застосунку	01.03.2024	
6.	Написання фронтегд-частини застосунку	01.04.2024	
7.	Написання текстової частини	14.05.2024	
8.	Захист курсової роботи	24.05.2024	

Зміст

Анотація

Метою роботи є дослідження та висвітлення світу реактивного програмування за допомогою сучасного фреймворку Spring WebFlux. У роботі показано особливості розробки дистрибутивної системи, яка легко масштабується, а архітектура побудована на основі взірців SQRS та Event Sourcing. Окрім цього комунікація між компонентами відбувається в асинхронній манері, що дозволяє оптимізувати програму.

Дана робота складається з трьох розділів. Перший — теоретичний, який познайомить з концепціями реактивного програмування. У другому, мова буде про фреймворк Spring WebFlux та його використання у застосунку. Третій розповість про архітектуру застосунку, комунікацію між компонентами та реальне використання цього застосунку.

Вступ

У сучасному мінливому світі розробки програмного забезпечення традиційні синхронні та імперативні парадигми вже давно стали нормою. Вони, хоча й ефективні у простих додатках, часто не відповідають вимогам високонавантажених систем, що обробляють велику кількість одночасних запитів. Такі проблеми, як неефективне використання ресурсів, складність обробки великих обсягів паралельних запитів і складність управління станом чи залежностями, є загальними проблемами, з якими стикаються розробники. Ці проблеми можуть призвести до зниження швидкості реагування системи, збільшення затримок і ускладнення масштабування додатків відповідно до потреб користувачів.

Однією з найважливіших проблем у такмх системах є блокування операцій вводу/виводу. Коли потік бере участь в такій операції, він “простоює”, очікуючи на завершення операції та отримання відповіді. Це призводить до неефективного використання системних ресурсів, оскільки потоки зайняті без виконання будь-якої значущої роботи. Крім того, масштабування додатків для роботи з великою кількістю одночасних користувачів стає складним, оскільки кожен запит вимагає виділення окремого потоку, що може швидко вичерпати кількість доступних.

Іншою проблемою є тісний зв'язок між компонентами в синхронних системах: зміни в одній частині системи впливають на іншу, до якої потрібно внести відповідіні зміни, що робить систему менш гнучкою і складнішою в обслуговуванні. Крім того, традиційні синхронні методи не дуже добре справляються зі помилками. Для прикладу якщо компонент виходить з ладу, це може спричинити зупинку всієї системи.

Реактивне програмування, зокрема використання фреймворків, таких як Spring WebFlux, пропонує ефективне розв'язання цих проблем. Реактивне програмування - це парадигма, яка дозволяє розробникам створювати більш стійкі, гнучкі та масштабовані системи. Вона зміщує акцент з імперативного та блокуючого коду на декларативні операції, дозволяючи системі обробляти велику кількість паралельних операцій з меншою кількістю ресурсів.

Розділ 1. Реактивне програмування

Реактивне програмування, у першу чергу, це парадигма програмування, яка визначає у якому руслі розробник має думати. Основна ідея реактивного програмування полягає в тому, що існують певні типи даних, які представляють значення «з плином часу». Окрім цього, реактивна парадигма містить в собі особливості інших парадигм, таких як об'єктноорієнтоване та функціональне. З першої було взято об’єкти які інкапсулюють дані, з другої — функції, які зазвичай використовуються з операторами, про яких мова буде вестися далі.

Ідея створення реактивного програмування зовсім не нова, оскільки потреба створення такої парадигми, з’явилися з появою інтерактивних застосунків, які мають реагувати на зміну стану програми. Поняття змін охоплює широкий спектр подій, від присвоєння нового значення змінної до натискання клавіш. Ще у 1980-их з’явилася перша бібліотека для мови прогумування Haskel, яка позиціювала себе як система для функціональної реактивної анімації [1]. Також, першими спробами імплементації можна вважати синхронні мови програмування ESTEREL, LUSTRE і SIGNAL [2]. Ці мови забезпечують абстрактну основу, де реакція системи на вхідні дані є миттєвою та детермінованою. Така особливість висвітлює їхню синхронну природу. Вони допомагають керувати паралелізмом, притаманним реактивним системам, абстрагуючись від реакцій системи, щоб вони були миттєві. Важливо розуміти що сучасні реактивні системи, зазвичай вважаються асинхронним, оскільки вони у своїй основі мають справу з потоками даних і поширенням змін. Це означає, що модель програмування не передбачає синхронного потоку управління, де виконання чекає на події. Натомість вона базується на асинхронних потоках даних та обробці подій, де компоненти реагують на події в міру їхнього надходження, не зупиняючись, щоб дочекатися завершення попередніх подій.

Реактивне програмування дозволяє розробникам декларувати, що робити з цими подіями, а програма бере відповідальність коли запускати визначені операції. Розглянемо простий приклад обчислення суми двох змінних (рис. 1.1).

```
1   var1 = 1
2   var2 = 2
3   var3 = var1 + var2
```

Рис. 1.1

У звичайному імперативному програмуванні значення змінної *var3* завжди міститиме 3, що є сумою початкових значень змінних *var1* і *var2*, навіть якщо змінним *var1* і *var2* пізніше буде присвоєно нове значення. У реактивному програмуванні значення змінної *var3* завжди підтримується в актуальному стані. Іншими словами, значення змінної *var3* буде автоматично перераховуватися з плином часу щоразу, коли змінюється значення змінних *var1* або *var2*. Це ключове поняття реактивного програмування.

Згідно з роботою [3], реактивне програмування може бути схарактеризувати на основі шести понять: базові абстракції, модель виконання, ліфтинг, багатовекторність, уникнення збоїв та підтримка дистрибутивності.

1.1 Базова абстракції

У кожній парадигмі є свої базові абстракції на яких і побудовані будь-які програми. Такі абстракції полегшують розробку та надають можливість розробляти більш складні абстракції на снові базових. Наприклад для імперативної парадигми такими абстракціями можуть бути змінні та операція присвоєння. Своєю чергою реактивне програмування має такі основні поняття: *Behavious* і *Events* [3].

Behavious — це вищезгадані значення “з плином часу”, які безперервно змінюються під час виконання програми. Вони абстрагуються від поняття стану, який змінюється з часом, дозволяючи розробникам моделювати стан системи як функцію часу, послідовно оновлюючи його відповідно до змін у навколишньому світі. *Events* - це події, які можуть вплинути на стан системи. Це можуть бути дії користувачів, сигнали датчиків або повідомлення від інших систем. Події можуть викликати зміни в *Behavious*, що призводить до каскадних оновлень у всій програмі.

Реактивне програмування зазвичай передбачає побудову графа потоків даних, де вузли представляють обчислювальні одиниці (функції, оператори тощо), а ребра — залежності даних між цими обчисленнями. Коли вхідні дані функції змінюються (через оновлення поведінки або спрацювання події), функція автоматично пере обчислюється, а її результат поширюється через граф.

Для того аби зіставити ці поняття з сучасними імплементаціями реактивної парадигми важливо ввести таку низку понять: *Observables*, *Observers*, *Operators*, *Subscriptions* і *Reactive Streams.*

Observables у сучасних бібліотеках, таких як RxJava, RxJS, Rx.NET тощо, розширюють концепцію *Behavious*, представляючи потік даних у часі, який може набувати різних значень. Як і *Behavious*, *Observables* є основною концепцією реактивного програмування, але вони можуть працювати з послідовністю значень (потоками), а не лише з одним значенням, що змінюється. Це робить його більш загальними, ніж *Behavious*.

Events в традиційному реактивному програмуванні запускають оновлення в системі, впливаючи на *Observables* і змушуючи реактивну систему оновлюватися відповідно. У сучасних фреймворках події, що спостерігаються, такі як *next*, *error* і *complete*, слугують аналогічній меті. next-події відповідають новим доступним даним, подібно до традиційних тригерів подій. *error* використовується для обробки помилок у потоці даних, а *complete* сигналізує про кінець потоку даних. Така структура дозволяє *Observables* повідомляти не лише про оновлення даних, але й про життєвий цикл самого потоку даних.

Observer - це механізм, який реагує на зміни. У класичному розумінні сама система може вважатися спостерігачем подій і поведінки. У сучасних бібліотеках Observer явно підписується на спостережувані об'єкти, щоб реагувати на “випущені” події, дозволяючи більш пряму і контрольовану взаємодію з потоком даних.

Operators є аналогом функціональних перетворень, що застосовуються до поведінки у традиційних реактивних системах. Подібно до того, як можна використовувати функції для перетворення або комбінування *Behavious*, оператори дозволяють трансформувати комбінувати та керувати потоками даних в *Observables*.

Reactive Streams забезпечують стандарт для обробки асинхронних потоків з *backpressure* [4], метод управління ресурсами, що буде розглянути у наступному підрозділі. Ця концепція розширює традиційні моделі реактивного програмування, додаючи надійність і масштабованість шляхом стандартизованої поведінки, яка не була явно охоплена в попередніх абстракціях реактивного програмування, але має вирішальне значення для побудови додатків з великим навантаженням.

1.2 Модель виконання

Модель виконання визначає як саме зміни стану будуть поширюватися по реактивним потокам. З погляду розробника, поширення змін відбувається автоматично. У цьому і полягає суть реактивного програмування: зміна має автоматично поширюватися від *Observable* (далі — виробник) до всіх *Observer'ів* (далі — споживачі). Коли нова подія з'являється, споживачі повинні опрацювати зміни, що, можливо, призведе до переобчислення. На рівні мови, проєктне рішення полягає у вирішенні хто ініціює розповсюдження змін. Тобто, чи виробник має «підштовхувати» нові дані до своїх залежних споживачів, чи залежні споживачі мають «витягувати» дані з виробника подій. В обох випадках послідовність значень рухається від виробника до споживача.

У моделі, заснованій на витягуванні (*pull-based*), споживачі, які потребують змін стану, повинні «витягувати» його з джерела. Тобто, розповсюдження керується попитом на нові дані (demand-driven). Така модель передбачає можливість того, що об'єкти, які потребують значення, мають свободу витягувати нові значення лише тоді, коли це дійсно потрібно або з певною періодичністю. Для легшого розуміння pull-модель можна прирівняти до взірця *Itertors*, який визначає два методи "*hasNext()*" та "*getNext()*". Основна проблема такої моделі полягає в тому, що вона може призвести до значного запізнення між виникненням події та реакцією на неї, оскільки споживач не може одразу дізнатися про нові зміни поки сам цього не перевірить їх наявність.

У моделі, заснованій на підштовхуванні (*push-based*), коли джерело має нові дані, воно передає їх до залежних від нього споживачів. Тобто, розповсюдження визначається наявністю нових даних (*data-driven*), а не попитом на них. Зазвичай це передбачає виклик зареєстрованого зворотного виклику або методу. Можна помітити схожість до взірця *Obsevable,* де зацікавлені суб'єкти підписуються на

об’єкт і коли щось з ним відбувається, він викликає метод суб'єктів, аби повідомити про цю зміну. Оскільки розповсюдження змін визначається наявністю нових даними, реагування на цю зміну відбувається миттєво. Імплементації реактивної парадигми, що реалізовують push-модель, потребують ефективного розв'язання проблеми марнотратних переобчислень, оскільки переобчислення відбуваються кожного разу, коли змінюються джерела вхідних даних.

У реактивній системі на основі підштовхування, джерело може генерувати дані зі швидкістю, більшою за ту, яку може обробити споживач. Без механізму контролю цього потоку у споживача може вичерпатися пам'ять або він може втратити здатність реагувати, що призведе до збоїв у роботі системи. Backpressure (далі — протитиск) дозволяє споживачеві сигналізувати виробнику про те, скільки ще даних він може обробити в цей момент часу, таким чином запобігаючи переповненню і підтримуючи стабільність. Так утворюється третя модель на основі двох попередніх, що об’єднує їхні переваги та запобігає перенавантаженню системи.

1.3 Ліфтинг

Під ліфтингом у реактивному програмуванні розуміють процес перетворення функції, яка оперує значеннями, у функцію, яка оперує потоками значень. Це фундаментальний механізм, який дозволяє розробникам застосовувати традиційні синхронні функції до асинхронних потоків даних, не змінюючи реалізацію функції. Процес ліфтингу можна розуміти як обгортання функції в іншу функцію, яка обробляє асинхронну природу потоків даних. Коли функцію підіймають, це не змінює логіку або поведінку оригінальної функції. Натомість це покращує роботу функції в контексті потоку даних, дозволяючи їй оперувати з кожним елементом, що проходить потоком під час його проходження через систему.

Функція f (рис. 1.3.1) отримує число x і повертає його, збільшивши на одиницю. У синхронному світі ви б застосували цю функцію безпосередньо до числа: *f(2)*, яка повернула б 3.

```
1   f(x) = x + 1
```

Рис. 2.3.1

У реактивному програмуванні дані подаються не як прості значення, а як потоки значень. Щоб застосувати функцію *f* до потоку, розробник повинен підняти *f* в реактивній манері. Процес підйому обгортає *f* так, щоб її можна було застосувати до кожного елемента в потоці, створюючи новий потік, де кожен елемент є результатом застосування *f*.

1.4 Уникнення збоїв

Уникнення збоїв — ще одна властивість, яку має враховувати реактивна система. Збої — це неузгоджені оновлення, які можуть виникати під час поширення змін. Коли обчислення виконується до того, як обчислюються всі залежні від нього вирази, це може призвести до того, що свіжі значення будуть об'єднані із застарілими, що призведе до *збою*. Вони призводять до некоректного стану програми та марнотратного використання ресурсів і тому їх слід уникати. Більшість реактивних програм усувають збої шляхом розташування виразів у топологічно відсортованому графі, таким чином гарантуючи, що вираз завжди обчислюється тільки після того, як обчислені всі його залежні.

1.5 Багатовекторність

Ще однією властивістю реактивних мов програмування є те, чи відбувається поширення змін відбувається в одному напрямку (односпрямовані) чи в обох напрямках (різноспрямовані). При багатовекторності зміни похідних значень поширюються назад до виробників, з яких вони були отримані. Наприклад, запис виразу *F = (C * 1,8) + 32* для перетворення температури між Фаренгейтом і Цельсієм, означає, що кожного разу, коли стає доступним нове значення *F* або *C*, інше значення оновлюється.

1.6 Підтримка дистрибутивності

Ця властивість стосується того, чи надає реактивна мова підтримку для написання розподілених реактивних програм. Підтримка розподілення дозволяє створювати залежності між обчисленнями або даними, які розподілені між декількома вузлами. Наприклад, у виразі *var3 = var1 + var2*, *var1, var2 і var3* можуть бути розташовані на різних вузлах. Потреба у підтримці дистрибутивності у реактивній мові зумовлена тим, що додатки, наприклад, вебдодатки, мобільні додатки тощо, стають все більш розповсюдженими. Однак, у додаванні підтримки

дистрибутивності до реактивної мови є певні підводні камені. Забезпечити узгодженість графа залежностей, який розподілений між декількома сервісами, складніше між кількома вузлами через особливості таких систем, такі як затримки, збої в мережі тощо.

1.7 Порівняння реактивних бібліотек.

У своїй роботі [3], автори порівняли 15 різних бібліотек і окремих мов програмування, які реалізовуй реактивне програмування [3]. Продовжуючи їхній вклад, далі буде йти мова про три найпопулярніші бібліотеки для мови програмування Java. Java не є «реактивною мовою» в тому сенсі, що вона не підтримує це нативно. Існують інші мови на JVM (Scala та Clojure), які підтримують реактивні моделі більш нативно, але сама Java не підтримує їх до версії 9. Проте, ця мова програмування є потужним інструментом розробки, і останнім часом спостерігається велика активність у створенні реактивних шарів поверх JDK. Тому продовжуючи вищезгадану працю, далі буде йти порівняння трьох найпопулярніших реактивних бібліотек: RxJava, Project Reactor і Mutiny.

1.7.1 RxJava

RxJava - це реалізація Reactive Extensions для віртуальних машин Java: бібліотека для створення асинхронних програм. Вона розширює патерн *Obesrvable* для підтримки потоку даних/подій і додає оператори, які дозволяють складати оператори декларативно, абстрагуючись від проблем низькорівневої багато поточності, синхронізації, безпеки потоків і паралельних структур даних. [5]. Прикладом масштабного використання RxJava стало оптимізація Netflix API [6].

RxJava використовує клас *Observable* як основну абстракцію, яка представляє потік даних, що може створювати декілька елементів з плином часу. Бібліотека також підтримує *Single, Maybe, Completable* та *Flowable* для різних типів потоків даних.

- *Observable* (рис. 1.7.1.1) — потік даних, який може видати 0 або більше елементів, за якими слідує або успішне завершення, або помилка.

```
Observable<String> observable = Observable.just("Hello", "World");
observable.subscribe(System.out::println);
```

Рис. 3.7.1.1

- *Single* (рис. 1.7.1.2) — спеціалізований потік, який видає лише один елемент або помилку. Він корисний для операцій, від яких очікується одна відповідь, наприклад, HTTP запитів або запитів до бази даних.

```
Single<String> single = Single.just("Hello World");
single.subscribe(System.out::println);
```

Рис. 1.7.1.2

- *Maybe* (рис. 1.7.1.3) — потік для одного значення, відсутності значення взагалі або помилки. Корисно для операцій, де результат може бути необов'язковим.

```
Maybe<Integer> maybe = Maybe.just(1);
maybe.subscribe(System.out::println, Throwable::printStackTrace, () -> System.out.println("Completed without a value"));
```

Рис. 1.7.1.3

- *Completable* (рис. 1.7.1.4) — потік для операцій, де потрібно лише статус завершення, а не значення. Ідеально підходить для таких завдань, як запис до бази даних або файлу, де цікавить лише успіх чи невдача операції.

```
Completable completable = Completable.complete();
completable.subscribe(() -> System.out.println("Completed Successfully"));
```

Рис. 1.7.1.4

- *Flowable* (рис. 1.7.1.5) — схожий на Observable, але з протитиском, що дозволяє йому обробляти величезну кількість даних. У прикладі використано observeOn(Schedulers.io()), що спрямовує Flowable на підштовхування наступних подій у потоці, яким керує планувальник io(). Своєю чергою він підтримується пулом потоків, який оптимізовано для операцій вводу/виводу, таких як читання і запис файлів, транзакцій з базами даних та мережевих

операцій.

```
Flowable.range(1, 1000)
        .observeOn(Schedulers.io())
        .subscribe(System.out::println);
```

Рис. 1.7.1.5

Оператори в RxJava можуть перетворювати потоки даних за допомогою різноманітних вбудованих методів, таких як *map*, *filter* і *flatMap*, які дозволяють здійснювати складні перетворення і маніпуляції з даними.

RxJava використовує push-модель [7], де дані виштовхуюся виробником, не чекаючи запиту від споживача. Ця модель асинхронна за своєю структурою, використовуючи планувальники для відокремлення потоку виконання від потоку, що викликає. RxJava використовує ліниве виконання у своєму API: застосування перетворень або об'єднання декількох *Observables* не ініціює жодного потоку даних або обчислень. Натомість ці операції лише створюють схему потоку даних, а обчислення та обробка даних починаються лише тоді, коли підписник явно підписується на *Observable*. Такий підхід дозволяє RxJava створювати складні "конвеєри" даних, не витрачаючи жодних ресурсів, поки результат цих конвеєрів не буде дійсно потрібен.

RxJava містить стратегії для обробки надмірної кількості даних і збоїв за допомогою стратегій протитиску, особливо з типом Flowable (рис. 1.7.1.6).

```
Flowable<Integer> source = Flowable.range(1, 1000000).onBackpressureBuffer();
source.observeOn(Schedulers.computation(), false, 100)
      .subscribe(System.out::println);
```

Рис. 1.7.1.6

Оператор *onBackpressureBuffer()* використовується для обробки протитиску шляхом буферизації всіх елементів, які не можуть бути негайно спожиті наступними операторами. Розмір буфера за замовчуванням необмежений і потенційно може зростати до нескінченності. Хоча це запобігає втраті даних, це створює ризик виникнення помилки *OutOfMemoryError*, якщо джерело постійно випереджає споживача й у JVM закінчується пам'ять.

Оператор *observeOn(Schedulers.computation())* змінює потік, в якому оператори, що знаходяться нижче, спостерігають за результатами потоку Flowable, що знаходиться вище. У цьому випадку використовується *Schedulers.computation(),* який призначений для ресурсомісткої роботи й використовує фіксовану кількість потоків, виходячи з кількості доступних процесорів. boolean-параметр визначає, чи затримувати помилки від висхідного потоку. Якщо встановлено значення false, при будь-яких помилках негайно повідомлятиметься наступним операторам і не затримуватиметься, доки не буде оброблено всі події, що не містять помилок. Параметр 100 визначає кількість елементів для попередньої вибірки з висхідного потоку. Цей параметр допомагає оптимізувати потік даних, запитуючи дані по 100 елементів за раз, балансуючи між використанням пам'яті та перемиканням контексту. Занадто велика кількість елементів може призвести до перевитрати пам'яті, якщо елементи обробляються недостатньо швидко, тоді як замала кількість може призвести до частіших запитів до висхідного потоку, що збільшує накладні витрати.

RxJava використовує злиття операторів (*operator fusion*) для підвищення продуктивності. Злиття операторів — це процес, коли сусідні оператори в реактивному потоці об'єднуються в один оператор, який виконує завдання всіх цих окремих операторів. Це робиться для мінімізації кількості проміжних кроків і переходів стану в потоці, що може зменшити накладні витрати на перемикання контексту і розподіл пам'яті, що призводить до підвищення продуктивності. Попри цю оптимізацію, в праці [8] зазначається, що вона не обов'язково дає кращу продуктивність у практичних застосувань, прив'язаних до вводу/виводу. Однак, для завдань, пов'язаних з процесором, RxJava демонструє вищу продуктивність порівняно з двома іншими бібліотеками, що свідчить про те, що її оптимізація більш ефективні у сценаріях зі складними обчисленнями, а не з інтенсивним введенням/виведенням.

RxJava може обробляти розподілені операції, дозволяючи використання реактивних потоків даних через мережі. Вона є високоефективною для сценаріїв, що включають асинхронні операції вводу/виводу між розподіленими сервісами, такими як мікросервіси або при використанні без серверної архітектури. Для прикладу, RxJava можна комбінувати з мережевими бібліотеками для полегшення неблокування мережевих викликів і безперешкодної інтеграції з розподіленими

системами, керованими подіями (*event-driven*). RxJava часто використовується в парі з Retrofit для обробки мережевих запитів у неблокуючий спосіб [9].

1.7.2 Project Reactor

Reactor використовує Mono та Flux як основні абстракції для роботи з одиничними та множинними виробниками даних відповідно. Ці типи інтегруються безпосередньо з екосистемою Spring, розширюючи її можливості.

Mono (рис. 1.7.2.1) — це спеціалізована імплементація інтерфейсу Publisher, яка видає максимум один елемент. Mono являє собою одну асинхронну операцію і є аналогом CompletableFuture в Java, але не блокується і підтримує протитиск реактивних потоків, тому ідеально підходить для зв'язку «запит-відповідь», коли очікується лише один результат, наприклад, HTTP-запит або запит до бази даних.

```
1   Publisher<String> mono = Mono.just("Hello World");
2   mono.subsribe(System.out::println)
```

Рис. 1.7.2.1

Flux (рис. 1.7.2.2) — це ще одна імплементація інтерфейсу Publisher, яка являє собою реактивну послідовність з 0-N елементів. Вона використовується для потоків даних, які можна обробляти індивідуально в міру їх надходження. Це робить Flux ідеальним для операцій, де очікується багато елементів, наприклад, потокова передача результатів запиту до бази даних або обробка рядків з файлу в реальному часі.

Подібно до RxJava, Reactor також дотримується push-моделі, але він глибоко інтегрується з модулем WebFlux у Spring, забезпечуючи неблокуючий веб стек. Також реалізує ліниве виконання у своїх компонентах. Визначення потоків даних за допомогою Flux або Mono не запускають жодної обробки або потоку даних. Такі операції, як *map, filter* або *flatMap*, лише описують операції, що мають бути

виконані, і не виконуються, доки не з'явиться підписка на цей потік.

```
Publisher<String> mono = Flux.just("Hello", "World");
mono.subsribe(System.out::println)
```

Рис. 1.7.2.2

Ця бібліотека підтримує реактивні потоки даних у обидва напрямки (рис. 1.7.2.3) і може ефективно використовуватися для складних сценаріїв потоків даних у реактивних додатках.

```
Mono<String> request = Mono.just("Request");
Mono<String> response = request.map(req -> "Response to " + req);
response.subscribe(System.out::println);
```

Рис. 1.7.2.3

Reactor дозволяє розробникам налаштовувати спосіб застосування протитиску, що особливо помітно у Flux.

- *onBackpressureDrop()*: Скидає вхідні дані, якщо нижній потік не встигає за ними.
- *onBackpressureBuffer()*: Буферизує всі елементи до того часу, поки наступний потік не зможе їх обробити, з необов'язковим обмеженням ємності, щоб запобігти виникненню помилки OutOfMemoryError.
- *onBackpressureError()*: Сигналізує про помилку, якщо наступний потік не встигає.
- *onBackpressureLatest()*: Зберігає лише останній елемент, відкидаючи всі попередні, якщо наступний елемент не готовий.

Для прикладу розглянемо оператор onBackpressureDrop() (рис. 1.7.2.4) Reactor надає такі механізми, як limitRate(int), що дозволяють більш детально контролювати швидкість споживання даних. Цей метод дозволяє встановлювати параметри попередньої вибірки, балансуючи між пропускною здатністю та використанням ресурсів, що особливо корисно у сценаріях зі змінною швидкістю

потоку даних.

```
1  Flux.range(1, 1000)
2      .onBackpressureDrop()
3      .subscribe(System.out::println);
```

Рис. 4. 7.2.4

Подібно до RxJava, Reactor використовує методи злиття операторів та попередньої вибірки. Вони призначені для підвищення ефективності обробки даних шляхом зменшення кількості операцій та проактивної вибірки даних до того, як вони будуть потрібні для наступних операцій. Оптимізації в Reactor також мають тенденцію до збільшення складності розробки та обслуговування. У вищезгаданій праці [] сказано, що оптимізації як і в RxJava, не суттєво підвищують продуктивність у сценаріях, пов'язаних з вводом/виводом. Відзначається, що Reactor працює з такою самою ефективністю як і RxJava в CPU-бенчмарках, але не настільки ефективно, як RxJava в деяких з наведених мікробенчмарків.

При обробці даних, розподілених між різними вузлами, наприклад, в архітектурі мікросервісів або при взаємодії з розподіленою базою даних, Reactor можна використовувати для ефективного управління асинхронними та неблокуючими потоками даних, що буде висвітлено у наступних розділах

1.7.3 Mutiny

Mutiny - це ще одна реактивна бібліотека, яка відзначається простотою використання та інтуїтивно зрозумілим синтаксисом. Mutiny займає важливе місце у фреймворку Quarkus та інструментах Vert.x. Особливо помітним покращенням відносно інших бібліотек було забезпечити інтуїтивний підхід до обробки асинхронних операцій, зосереджуючись на простоті використання та зменшенні шаблонного коду в реактивному програмуванні. Бібліотека дотримується підходу,

Mutiny пропонує *Uni* для одиничних асинхронних подій та *Multi* для множинних подій, зосереджуючись на простоті та ясності реактивного програмування.

Uni (рис. 1.7.3.1) — тип для асинхронних дій, які призводять до єдиного результату або невдачі.

```
Uni.createFrom().item("Hello World")
    .subscribe().with(item -> System.out.println(item));
```

Рис. 5.7.3.1

Multi (рис. 1.7.3.2) — тип для роботи з потоками даних (0..n елементів).

Mutiny пропонує кілька стратегій для обробки сценаріїв, коли споживач не може встигати за виробником. Ці стратегії дозволяють розробникам контролювати потік даних і визначати, як слід обробляти надлишкові елементи.

```
Multi.createFrom().items("Hello", "World")
    .subscribe().with(System.out::println);
```

Рис. 6.7.3.2

1. Стратегія за замовчуванням, за якої, якщо наступний потік не встигає, він сигналізує про помилку переповнення. Це може бути корисно у сценаріях, де втрата даних неприйнятна, і краще відмовитися швидко і помітити проблему.
2. Буферизація всіх елементи, які не може опрацювати, за допомогою *onOverflow().buffer(int)* Ця стратегія корисна, коли виникають тимчасовий і стрімкий потік даних, але тривале перевиробництво може призвести до помилки *OutOfMemoryError*. На рис. 1.7.3.3 показано приклад буферизації 100. елементів.

```
Multi.createFrom().ticks().every(Duration.ofMillis(1))
          .onOverflow().buffer(100)  // Buffer up to 100 items
          .subscribe().with(
              item -> System.out.println("Processed: " + item),
              failure -> System.out.println("Failed with: " + failure.getMessage())
          );
```

Рис. 1.7.3.3

3. Видалення найновіших елементів, коли наступний потік не встигає за ними, за допомогою оператора o*nOverflow().drop()*.

4. Зберігання лише останні елементи, перезаписуючи всі попередні необроблені елементи *onOverflow().latest()*.
5. Опрацювання лише останнього елементу, отриманого за вказаний проміжок часу, за допомогою *onOverflow().debounce(Duration)*.

Підтримуючи базові багатовекторні потоки, Mutiny фокусується на спрощенні моделі реактивного програмування, через що жертвує деякою гнучкістю (рис. 1.7.3.4), яку можна побачити в інших бібліотеках.

```
1  Uni<String> uni = Uni.createFrom().item("Request");
2  Multi<String> responses = uni.onItem()
3    .transformToMulti(request ->Multi.createFrom().items("Response"));
4  responses.subscribe().with(System.out::println);
```

Рис. 1.7.3.4

Операції, оголошені на Uni або Multi, такі як перетворення або підписка на події, не виконуються, доки не буде здійснено підписку. Така лінива поведінка гарантує, що ресурсомісткі завдання відкладаються до тих пір, поки їх результати не будуть дійсно потрібні.

На відміну від RxJava та Reactor, Mutiny не реалізує складних оптимізацій, таких як злиття операторів або попередня вибірка. Основна увага в Mutiny приділяється підтримці простоти та зрозумілості кодової бази, що потенційно знижує витрати на розробку та підтримку. Однак це не ставить Mutiny у невигідне становище у бенчмарк тестах, а навпаки цих бібліотек показує себе найкраще у тестах пов'язаних з вводом/виводом, порівняно з іншими бібліотеками. Це свідчить про те, що простіші кодові бази без значних оптимізацій працюють краще у випадках наближених до реального використання.

Хоча Mutiny підтримує розподілені системи, для створення мікросервісів пріоритетною є інтеграція з Quarkus, що забезпечує безперешкодну інтеграцію з цим фреймворком Java.

1.7.4 Приклад реалізації

У цьому підрозділі розглянемо приклад, як кожна бібліотека реактивного програмування обробляє типовий сценарій. Сценарій, який ми розглянемо, — це

обробка потоку запитів, де кожен запит передбачає отримання даних з бази даних, їх обробку та можливу обробку тайм-аутів або помилок. Цей тип сценаріїв є поширеним у бекенд-сервісах, де дані потрібно отримувати та обробляти асинхронно, щоб підтримувати швидкість виконання запитів.

Уявіть, що у нас є API, який повинен асинхронно отримувати дані про користувачів з бази даних. Для кожного користувача нам потрібно:

1. Отримати дані користувача.
2. Перетворити дані (наприклад, перетворити ім'я користувача у верхній регістр).
3. Обробляти будь-які помилки бази даних або тайм-аути, надаючи запасні дані.

RxJava надає повний набір операторів, які можна використовувати для обробки асинхронних потоків даних зі складними перетвореннями та обробкою помилок (Додаток 1). Project Reactor використовує схожий підхід, надаючи реактивні рішення, які не блокуються і не реагують на протитиск (Додаток 2). Mutiny має на меті спростити концепції реактивного програмування і зосереджується на покращенні читабельності та зменшенні шаблонного коду (Додаток 3).

1. 8 Реактивність Vs Імперативність

Парадигми програмування визначають підхід до написання коду. Кожна з них має свої переваги, недоліки та сфери застосування. При розв'язанні проблеми важливо розуміти декілька парадигм аби визначити яка з них підходить найбільше.

Імперативна парадигма найбільш поширена серед продуктів та розробників. Вона зосереджується на виконанні завдання крок за кроком. Код у цьому випадку описує як саме має вирішуватися завдання команда за командою які змінить стан всієї програми. Для прикладу, визначення змінної, контроль над виконанням за допомогою циклів та умовних операторів. Імперативний підхід має низку плюсів:

1. Імперативне програмування дозволяє точно контролювати хід виконання програми, що є важливим для систем, які вимагають дуже високої ефективності обчислень.

2. Імперативні мови програмування добре підходять для завдань з інтенсивними обчисленнями, оскільки вони можуть бути високо оптимізовані, що дозволяє системі досягти високої продуктивності.
3. Імперативне програмування здебільшого використовується у низькорівневому програмуванні, наприклад, у ядрах або драйверах пристроїв, оскільки воно забезпечує високий рівень контролю над апаратним забезпеченням.

Серед мінусів присутні такі:

1. В імперативному програмуванні важко досягти абстракції, що ускладнює розробку великих складних систем.
2. Імперативне програмування зазвичай вимагає набагато більше рядків коду, ніж інші парадигми, що призводить до дуже великої кількості шаблонного коду, яким може бути важко керувати у великих складних системах.
3. Імперативне програмування важко розпаралелити, що є дуже несприятливим для розподілених систем.

Загалом, імперативне програмування є хорошим вибором, для високого ступеня контролю над ходом виконання програми й коли розв'язок задачі можна легко виразити у вигляді послідовності кроків. На приклад, утилітні скрипти, процедурні алгоритми, системні компоненти нижчого рівня чи реалізація ігрової логіки.

Своєю чергою, реактивне програмування фокусується на потоці даних і як дані змінюється з плином часу. Воно засноване на концепції подієво-керованого програмування, де потоки даних обробляються асинхронно, тобто в міру їх надходження. Згідно з маніфестом “The Reactive Manifesto" [10], реактивні системи мають підпорядковуватися таким вимогам:

1. Адаптивні системи зосереджені на забезпеченні швидкого та стабільного часу відгуку. Така послідовна поведінка зміцнює довіру кінцевого користувача та заохочує його до подальшої взаємодії.
2. Збої локалізуються всередині кожного компонента, ізолюючи компоненти один від одного і тим самим гарантуючи, що частини системи можуть виходити з ладу і відновлюватися без шкоди для системи в цілому. Це забезпечує високу доступність системи.

3. Реактивні системи залишаються доступними ми й можуть відповідати на зміни вхідних даних, збільшуючи або зменшуючи ресурси, виділені для обслуговування цих даних. Ця функція сприяє високій масштабованості.
4. Реактивні системи покладаються на асинхронну передачу повідомлень для встановлення межі між компонентами, що забезпечує вільний зв'язок, ізоляцію та прозорість розташування. Деблокуючи зв'язок дозволяє одержувачам споживати ресурси лише тоді, коли вони активно ініціюються дзвінком, повідомленням чи подією.

Окрім заявлених плюсів, реактивна парадигма має наступні мінуси:

1. Оскільки події обробляються асинхронно, може бути складно відстежити послідовність подій, які призвели до виникнення проблеми. Крім того, складна взаємодія між подіями, підписниками та спостерігачами може ускладнити написання тестів.
2. Реактивне програмування може призвести до певного зниження продуктивності через необхідність керувати й координувати потоки даних, що найбільш помітно в додатках, які передбачають великий обсяг даних або велику кількість підписників.
3. У реактивному програмуванні потоки даних можуть стати досить складними, особливо в міру того, як додатки стають більшими й складнішими. Це може ускладнити розуміння про поведінку системи та її підтримку з плином часу.

Загалом, реактивне програмування є гарним вибором, для створення додатків, які повинні швидко реагувати, масштабуватися та підтримуватися, а також якщо потрібно реагувати на зміни в даних з плином часу.

Розділ 2. Особливості роботи з фреймфорком Spring WebFlux

У світі мови програмування Java є безліч популярних бібліотек та фреймворків, які дозволяють створити неблокуючий, асинхронний та реактивний застосунок. Для прикладу, Vert.x, Akka HTTP, Quarkus, Micronaut та Spring WebFlux. Вибір впав саме на останній, через переваги які наведені нижче.

Spring WebFlux, частина екосистеми Spring Framework, вносить у Spring реактивне програмування, дозволяючи обробляти велику кількість одночасних з'єднань з невеликими обчислювальними ресурсами завдяки неблокуванню вводу/виводу. Цей фреймворк має наступні переваги:

1. Spring WebFlux легко інтегрується з широкою екосистемою Spring, включаючи Spring Data, Spring Security та Spring Cloud. Для розробників, які вже використовують Spring для інших частин свого додатка, це означає, що вони можуть використовувати з мінімальними зусиллями. WebFlux використовує знайому модель конфігурації Spring на основі анотацій, що полегшує розробникам перехід з Spring MVC на WebFlux.
2. WebFlux можна використовувати з Reactor, який використовується за замовчуванням, RxJava або будь-якою іншою реактивною бібліотекою, що підтримує специфікацію Reactive Streams, пропонуючи гнучкість залежно від уподобань розробника або наявного стека.
3. Spring Boot надає широку підтримку для тестування реактивних додатків за допомогою таких інструментів, як *WebTestClient*, що полегшує написання інтеграційних тестів для WebFlux-додатків.
4. Spring має велику й активну спільноту, що забезпечує велику кількість ресурсів, від начальної інформації до сторонніх бібліотек та інструментів.
5. Spring Data надає реактивну підтримку для різних сховищ даних, включаючи MongoDB, Cassandra, Redis та R2DBC (Reactive Relational Database Connectivity) для баз даних SQL. Це дозволяє здійснювати повністю реактивні транзакції з базами даних та потокову обробку.

Як вказано у перевагах, WebFlux підтримує не тільки бібліотеку Project Reactor, але обрано саме її, через вподобання автора, повагу до розробників Spring Framework та ефективність, яка хоч й не найкраща, але заслуговує уваги.

Цей розділ фокусується на особливостях розробки реактивного бекенду порівняно зі звичною моделлю Sping MVC від компонентів пов'язані зі сховищами даних до HTTP та WebSocket контролерів. Приклади коду буде взятий з реактивного застосунку, що репрезентує бекенд-частину системи управління завдань.

2.1 Робота з MongoDB

Поєднання реактивного програмування та баз даних NoSQL стає все більш популярним для створення високомасштабованих та адаптивних додатків. MongoDB, одна з найпопулярніших баз даних NoSQL, надзвичайно добре поєднується з реактивним фреймворком Spring WebFlux, забезпечуючи потужний стек для розробників. MongoDB - це документна NoSQL-база даних, відома своєю високою продуктивністю, доступністю та простотою масштабування. Вона зберігає дані у документах у форматі JSON, поля якою можуть змінюватися від документа до документа. MongoDB добре компонується з реактивним Java-стеком, оскільки має нативний драйвер для такої комунікації [11].

Інтегрування розпочинається з декларування залежностей та прописування конфігурації з’єднання. Оскільки ці кроки не відрізняються вони будуть пропущені, а в Додатку 4 можна знайти весь перелік залежностей.

Для того аби активувати реактивну підтримку для MongoDB, потрібно створити конфігураційний клас, який розширює абстрактний клас AbstractReactiveMongoConfiguration, як показано на для рис. 2.1.1.

```java
package org.tasker.common.config;

import com.mongodb.reactivestreams.client.MongoClient;
import com.mongodb.reactivestreams.client.MongoClients;
import org.springframework.context.annotation.Bean;
import org.springframework.data.mongodb.config.AbstractReactiveMongoConfiguration;
import org.springframework.data.mongodb.repository.config.EnableReactiveMongoRepositories;

@EnableReactiveMongoRepositories
public class ReactiveMongoConfiguration extends AbstractReactiveMongoConfiguration {

    @Bean
    public MongoClient mongoClient() {
        return MongoClients.create();
    }

    @Override
    protected String getDatabaseName() {
        return "reactive";
    }
}
```

Рис. 2.1.1

Анотація EnableReactiveMongoRepositories дозволить запустити автоматичну конфігурацію від Spring Boot. Важливо зазначити що стрічка “reacive” не назва база даних, а назва драйвера який має використовуватися.

Після цих налаштувань програма готова для створення документа та репозиторіїв. Розглянемо приклад на рис. 2.1.2 для документа Board, що відображає дошку з завданнями. Для визначення документа, індексів та назв полів використовуються анотація] з модуля org.springframework.data.mongodb.core, що є універсальним для реактивної та нереактивної MongoDB.

```java
package org.tasker.common.models.domain;

import org.springframework.data.mongodb.core.index.Indexed;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;
import org.springframework.data.mongodb.core.mapping.FieldType;
import org.springframework.data.mongodb.core.mapping.MongoId;

// ...

@Data
@Builder
@Document(collection = "boards")
public class BoardDocument {

    @MongoId(FieldType.OBJECT_ID)
    private String id;

    @Indexed(unique = true)
    @Field("aggregate_id")
    private String aggregateId;


    @Indexed
    @Field("owner_id")
    private String ownerId;

    //...

}
```

Рис. 2.1.2

Spring забезпечує гнучку та зручну специфікацію репозиторіїв, з уже визначеними методами CRUD, а також з підтримкою деяких інших поширених методів. Spring Data надає той самий набір методів і специфікацій, за винятком того, що ми будемо працювати з результатами й параметрами в реактивний спосіб, як показано на рис. 2.1.3. У прикладі наведено використання ReactiveMongoRepository<T,ID> замість звичного MongoRepository<T,ID>. А об'єкти які повертається є Mono-обгортками. За необхідності репозиторії такою підтримують реактивні вхідні параметри, і замість String можна використати Mono<String>.

```java
package org.tasker.task.output.persistance;

import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import org.tasker.common.models.domain.BoardDocument;
import reactor.core.publisher.Mono;

public interface BoardRepository extends ReactiveMongoRepository<BoardDocument, String>,
  CustomBoardRepository {
    Mono<BoardDocument> findByAggregateId(String aggregateId);
    Mono<Void> deleteByAggregateId(String aggregateId);
}
```

Рис. 2.1.3

У деяких випадках, програма вимагає складніших запитів до MongoDB, які зручніше створювати за допомогою об’єкту MongoTemplate. Для таких випадків є реактивний аналог — ReactiveMongoTemplate. У прикладі (рис 2.1.3) інтерфейс BoardRepository також наслідує CustomBoardRepository ReactiveMongoTemplate.

ReactiveMongoTemplate є частиною реактивної інфраструктури Spring Data MongoDB, що забезпечує абстракцію вищого рівня для реактивних операцій MongoDB. Він інкапсулює основні функції та операції MongoDB, пропонуючи реактивний API для використання можливостей MongoDB у неблокуючий спосіб.

Update API використовується для модифікації документів у MongoDB. Criteria API використовується для побудови запитів. Використана API визначає умови, за яких документи запитуються або оновлюються. Розглянемо метод *addInvitedId* з рисунку (рис 2.1.5):

- *update*: Вказує клас сутності для оновлення.
- *matching*: Застосовує критерії, створені за допомогою API критеріїв.
- *apply*: Застосовує визначення оновлення.
- *all*: Вказує, що оновлення має бути застосовано до всіх документів, які відповідають критеріям.
- *doOnError* і *doOnSuccess*: Це методи побічних подій, які реагують на успіх або невдачу операції.
- then: Повертає Mono<Void>, що вказує на завершення операції.

У методі findBoardsByUserId використовуються комбінація критеріїв, щоб знайти документи на основі ідентифікаторів користувачів:

- *query*: Запускає операцію запиту для вказаного класу.
- *matching*: Застосовує екземпляр критерію, який використовує оператор orOperator для об'єднання декількох умов. В результаті будуть знайдені документи, де userId міститься в полі owner_id або joined_ids.
- *all*: Отримує всі документи, що відповідають критеріям.

Таким чином, можна інтегрувати MongoDB у WebFlux-застосунок та отримати повністю реактивний застосунок.

2.2 Робота з PostrgreSQL

Вибір між реляційними базами даних і NoSQL часто залежить від конкретних потреб або обмежень програми. Реляційні бази даних, що характеризуються своєю структурованою схемою та мовою SQL для доступу до даних та управління ними. Реляційні бази даних чудово справляються зі складними запитами. Вони підтримують розширені операції об'єднання для запитів що вимагають декілька таблиць.Реляційні бази даних, як правило, відповідають стандартам ACID, що робить їх ідеальним вибором для додатків, де узгодженість і надійність транзакцій є критично важливими. Якщо дані добре структуровані й навряд чи будуть часто змінюватися, реляційні бази даних є кращим вибором.

Для з'єднання з базою даних, звичайні Java-застосунки використовують JDBC драйвер, а модуль Spring Data забезпечує зручний та потужний функціонал для роботи з реляційними базами даних. Для роботи у контексті неблокуйючого та асинхронного програмування таке рішення є неприйнятим, тому з’явився драйвер R2DBC.

Концепція JDBC є синхронним і блокуючим. Коли здійснюється виклик JDBC, потік, що виконує цей виклик, блокується до завершення операції. Це може призвести до неефективного використання ресурсів, особливо в середовищах з високим рівнем паралелізму, де декілька потоків змушені чекати. JDBC дотримується традиційного імперативного стилю програмування, який є простим, але може бути менш ефективним при обробці операцій, пов'язаних з вводом-виводом або схильних до затримок. Своєю чергою R2DBC розроблений як неблокуючий та асинхронний. Він дотримується парадигми реактивного програмування, яка допомагає створювати масштабовані та ефективні додатки та не втрачати функціональність які забезпечує JDBC.

Через свою блокувальну природу JDBC споживає більше ресурсів програми. У сценаріях з високим навантаженням, може знадобитися більше потоків для обробки паралельних звернень до бази даних, що збільшує споживання пам'яті й накладні витрати на перемикання контексту. R2DBC, як неблокуючий, споживає менше ресурсів під навантаженням, оскільки не вимагає, щоб потоки очікували завершення операцій з базою даних.

JDBC керує транзакціями синхронно, що є простим для реалізації та розуміння. Своєю чергою R2DBC підтримує реактивні транзакції, які не блокуються. Однак реактивне керування транзакціями є складним і вимагає обережності, щоб уникнути Dead Lock або перевантаження.

Оскільки застосунок який розробляється побудованим на взірцях SQRS та Event Sourcing, про які буде йтися у наступному розділі. Поки зупинимося на тому, що події мають десь зберігатися, для цього була використана PosgreSQL, як високоефективна та поширена база даних. Для інтеграції реактивної PostgreSQL, залежності вказані у Додатку 4, а Sptring Boot візьме на себе відповідальність за конфігурацію з'єднання. Тому можна почати одразу з конфігурації яка дозволить застосувати міграції при запуску застосунку (рис 2.2.1). У теці *resources* міститься файл з декларацією усіх таблиць.

```java
package org.tasker.common.config;

import io.r2dbc.spi.ConnectionFactory;
import org.springframework.r2dbc.connection.init.CompositeDatabasePopulator;
import org.springframework.r2dbc.connection.init.ConnectionFactoryInitializer;
import org.springframework.r2dbc.connection.init.ResourceDatabasePopulator;
//...

@Configuration
public class DatabaseConfiguration {
    @Bean
    public ConnectionFactoryInitializer initializer(ConnectionFactory connectionFactory) {
        ConnectionFactoryInitializer initializer = new ConnectionFactoryInitializer();
        initializer.setConnectionFactory(connectionFactory);

        CompositeDatabasePopulator populator = new CompositeDatabasePopulator();
        populator.addPopulators(new ResourceDatabasePopulator(new ClassPathResource("schema.sql")));
        initializer.setDatabasePopulator(populator);

        return initializer;
    }
}
```

Рис. 2.2.1

Наведений приклад створює бін *ConnectionFactoryInitializer*, який містить популятор з файлом *schema.sql*, який розташований у теці *resources* та наведений у Додатку 5. Для моделі Event (рис 2.2.2) можемо не використовувати анотації через те, що будемо ми будемо використовувати *DatabaseClient.*

```java
package org.tasker.common.es;

// ...

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Event {

    private UUID id;
    private String aggregateId;
    private String eventType;
    private String aggregateType;
    private long version;
    private byte[] data;
    private LocalDateTime createdAt;

   // ...
}
```

Рис. 2.2.2

Spring Data R2DBC використовує репозиторії для забезпечення рівня абстракції для реактивного виконання операцій з базами даних. Єдина відмінність між цим репозитаріями та для JDBC драйвера полягає в об'єктах що повертається, аналогічно до ситуації з MongoDB. У реактивному програмуванні, повертається Mono<Event> та Flux<Event> замісь Event та List<Envent> відповідно.

DatabaseClient в Spring Data R2DBC - це універсальний інструмент, який дозволяє виконувати складні SQL-запити в реактивній манері. Він забезпечує гнучкість і контроль над SQL-операціями, що робить його ідеальним для ситуацій, коли стандартних методів репозиторію недостатньо. DatabaseClient пропонує різні методи для побудови та виконання SQL запитів, такі як sql(), bind(), fetch() та map(). У цьому прикладі (рис. 2.2.3) DatabaseClient використовується для виконання запиту з кількома параметрами, що є типовим для додатків, які вимагають динамічної фільтрації.

```java
import org.springframework.data.r2dbc.core.DatabaseClient;
import reactor.core.publisher.Flux;

public class CustomQueryService {

    private final DatabaseClient databaseClient;

    public Flux<Event> searchEventsWithFilters(String eventType, LocalDateTime startDate, LocalDateTime endDate) {
        String sql = "SELECT * FROM events WHERE event_type = :eventType AND created_at BETWEEN :start AND :end";

        return databaseClient
           .sql(sql)
            .bind("eventType", eventType)
            .bind("start", startDate)
            .bind("end", endDate)
            .map((row, metadata) -> new Event(
                row.get("id", UUID.class),
                row.get("aggregate_id", String.class),
                row.get("event_type", String.class),
                row.get("aggregate_type", String.class),
                row.get("version", Long.class),
                row.get("data", byte[].class),
                row.get("created_at", LocalDateTime.class)
            ))
            .all();
    }
}
```

Рис. 2.2.3

Декларування транзакцій не відрізняється від звичного підходу анотацій Spring Data (рис. 2.2.4). Відмінність поглядає у проксі які створяться та будуть керувати транзакціями. Для цього буде використаний ReactiveTransactionManager з рисунку

```java
@Bean
public ReactiveTransactionManager transactionManager(ConnectionFactory connectionFactory) {
  return new R2dbcTransactionManager(connectionFactory);
}
```

Рис. 2.2.4

Після цього можна використовувати анотація Transactional (рис. 2.2.5).

```java
@Transactional
public Mono<Event> createOrUpdateEvent(Event event) {
  return eventRepository.findById(event.getId())
              .defaultIfEmpty(event)
              .flatMap(existingEvent -> {
                  existingEvent.setData(event.getData());
                  return eventRepository.save(existingEvent);
              });
}
```

Рис. 2.2.5

Отже, JDBC-драйвер залишається надійним вибором для традиційних застосунків, тоді як R2DBC пропонує ефективний і сучасний підхід для тих, хто рухається до реактивних архітектур. При цьому єдина зручність може бути у виборі БД, оскільки не так багато їх мають створений реактивний драйвер.

2.3 Черги повідомлень

RabbitMQ є брокером повідомлень: він приймає і пересилає повідомлення. Повідомлення, отримані від виробників, зберігатимуться в черзі повідомлень, і як тільки споживачі будуть готові їх отримати, вони споживатимуть повідомлення. Саме так брокер повідомлень гарантує, що одержувач отримає певне повідомлення.

Як згадувалося раніше, виробник надсилає повідомлення брокеру повідомлень. Точкою входу цієї посередницької програми є *Exchange*, який відповідає за маршрутизацію повідомлень до різних черг. Зв'язок між *Exchange* і *Queue* називається Binding.

Традиційне використання RabbitMQ з Spring зазвичай охоплює *RabbitTemplate* та анотацію *RabbitListener*, які є блокуючими. Вони добре працюють в рамках синхронної моделі програмування, типової для багатьох Spring-додатків. Reactive RabbitMQ, використовуючи такі бібліотеки, як RabbitMQ Java Client і Porject Reactor, дозволяє неблокувати споживання та надсилання повідомлень з використанням типів Flux та Mono. Такий підхід є дуже корисним у середовищах, де потоки є дефіцитними ресурсами, оскільки він допомагає не блокувати ці потоки під час очікування повідомлень.

За допомогою Reactor RabbitMQ можна легко компонувати та трансформувати потоки повідомлень за допомогою широкого набору операторів, наданих Project Reactor. Наприклад, можна вільно об'єднувати, фільтрувати, трансформувати та затримувати повідомлення.

Для інтеграції Reactive RabbitMQ в Spring-застосунок, спершу потрібно задекларувати відповідні залежності (див. Додаток 4).

ConnectionFactory налаштовується за допомогою хосту, порту, імені користувача та паролю для підключення до RabbitMQ (рис. 2.3.1).

```java
@Bean
Mono<Connection> connectionMono(@Value("${spring.rabbitmq.host}") String host,
                                @Value("${spring.rabbitmq.port}") int port,
                                @Value("${spring.rabbitmq.username}") String username,
                                @Value("${spring.rabbitmq.password}") String password) {
    ConnectionFactory connectionFactory = new ConnectionFactory();
    connectionFactory.useNio();
    connectionFactory.setHost(host);
    connectionFactory.setPort(port);
    connectionFactory.setUsername(username);
    connectionFactory.setPassword(password);
    return Mono.fromCallable(() -> connectionFactory.newConnection("reactor-rabbit")).cache();
}
```

Рис. 2.3.1

Метод useNio() викликається для увімкнення неблокуючого вводу/виводу. Mono.fromCallable() використовується для відкладення створення з'єднання до моменту фактичної підписки на нього. Метод *cache()* кешує результат, гарантуючи повторне використання того самого з'єднання.

При створенні біна *SenderOptions* (рис. 2.3.2),

```java
@Bean
public SenderOptions senderOptions(Mono<Connection> connectionMono) {
    return new SenderOptions()
        .connectionMono(connectionMono)
        .resourceManagementScheduler(Schedulers.boundedElastic());
}

@Bean
public Sender sender(SenderOptions senderOptions) {
    return RabbitFlux.createSender(senderOptions);
}
```

Рис. 2.3.2

r*esourceManagementScheduler(Schedulers.boundedElastic())* встановлюється для управління ресурсами, такими як створення *Exhange*, *Queue* тощо, в окремому пулі потоків, що забезпечує ефективність. Бін *Receiver* та *ReceiverOptions*, для надсилання повідомлень, створюються аналогічно.

Для надсилання повідомлень використовується вищезгаданий бін *Sender*, який надає методи для реактивного надсилання повідомлень (рис. 2.3.3).

Спершу потрібно створити об'єкт *OutboundMessage*, який містить *Exchange*, ключ маршрутизації та тіло повідомлення. Для надсилання використовується метод sender.send(), передаючи Mono, який містить *OutboundMessage*. Цей метод не блокується і завершить роботу, коли повідомлення буде успішно надіслано.

```
private final Sender sender;

//...

public Mono<Void> sendMessage(String exchange, String routingKey, String message) {
    byte[] messageBody = message.getBytes(StandardCharsets.UTF_8);
    OutboundMessage outboundMessage = new OutboundMessage(exchange, routingKey, messageBody);
    return sender.send(Mono.just(outboundMessage));
}
```

Рис. 2.3.3

Метод *consumeAutoAck()* біна *Receiver* (рис. 2.3.4) використовується для споживання повідомлень з вказаної черги. Він автоматично підтверджує повідомлення після того, як вони спожиті.

```
public Flux<String> receiveMessages(String queueName) {
    return receiver.consumeAutoAck(queueName)
                   .map(delivery -> new String(delivery.getBody(), StandardCharsets.UTF_8));
}
```

Рис. 2.3.4

Використання Reactive RabbitMQ у Spring Boot-додатку дозволяє реактивно обробляти операції обміну повідомленнями, використовуючи реактивне програмування. Таке налаштування є особливо вигідним для високонавантажених, масштабованих додатків, де управління ефективністю використання ресурсів та швидкістю відгуку є критично важливим.

2.4 Взірець Request/Reply

Взірець *Request/Reply* - це спосіб комунікації, що використовується в розподілених системах, де клієнт надсилає запит на сервер і чекає на відповідь. Цей патерн є фундаментальним у сценаріях, де необхідна негайна відповідь після запиту, оскільки він дозволяє здійснювати прямий і синхронний зв'язок між розподіленими компонентами. Його можна порівняти з іншими шаблонами обміну

повідомленнями, такими як *Pub/Sub*, де повідомлення транслюються декільком підписникам без очікування відповіді.

В контексті RabbitMQ та реактивного програмування з Spring Boot, взірець Request/Reply може бути реалізований для ефективної обробки асинхронного, неблокуючого зв'язку. Цього можна досягти використовуючи приклад на рис. 2.4.1.

```
public Mono<byte[]> publishAndReceive(String actionName, Object messageBody) {
    final var correlationID = UUID.randomUUID().toString();
    final var responseQueue = responseQueueName + "." + correlationID;
    AMQP.BasicProperties properties = new AMQP.BasicProperties.Builder()
            .correlationId(correlationID)
            .replyTo(correlationID)
            .build();

    return sender.declareQueue(QueueSpecification.queue(responseQueue).exclusive(true).autoDelete(true))
            .then(sender.bind(BindingSpecification.binding(responseExchangeName, correlationID, responseQueue)))
            .then(sender.send(Mono.just(
                  new OutboundMessage(requestExchangeName, actionName, properties, serializeToJsonBytes(messageBody)))
                ).doOnSuccess(v -> log.info("Sent message to {}: {}", requestExchangeName, messageBody)))
            .thenMany(receiver.consumeAutoAck(responseQueue))
            .next()
            .map(delivery -> {
                log.info("Received response on {}: {}", correlationID, delivery);
                return delivery.getBody();
            });
}
```

Рис. 2.4.1

Кожному запиту присвоюється унікальний ідентифікатор, який використовується для зіставлення відповідей з запитами. Це дуже важливо в системі, де багато запитів і відповідей обробляються одночасно. Для кожного запиту створюється унікальна тимчасова черга, часто ексклюзивна і така, що автоматично видаляється, для отримання відповіді. Ексклюзивні черги видаляються, коли з'єднання, що їх декларує, закривається або розривається, наприклад, через втрату основного TCP-з'єднання.

Запит надсилається на вказаний *Exchange* з ключем маршрутизації та полями, які включають чергу відповідей та ідентифікатор кореляції. Відповідь споживається з унікально створеної черги. Функція consumeAutoAck() прослуховує повідомлення у цій черзі. Як тільки відповідь надходить, вона обробляється або трансформується за необхідності, а потім повертається.

Тепер розглянемо приклад зі сторони, яка отримує запит та надсилає відповідь (рис. 2.4.2).

```
private void handle(GetUserQuery query, Delivery delivery) {
    authQueryService.handle(query)
            .map(users -> // створення DTO-об'єкта)
            .subscribe(response -> {
                final String exchange = authMessagingSpecs.getResponseExchangeName();
                final byte[] serializedResponse = SerializerUtils.serializeToJsonBytes(response);
                final String routingKey = delivery.getProperties().getReplyTo();
                OutboundMessage message = new OutboundMessage(exchange, routingKey, serializedResponse);
                sender.send(Mono.just(message)).subscribe();
            });
}
```

Рис. 2.4.2

Наведений метод має справу з запитом *GetUserQuery*, на який має повернути користувача або повідомити про помилку. Вся бізнес-логіка відбувається в *authQueryService#handle(query)*, а потім тільки трансформується в DTO об’єкт. Об'єкт відповіді серіалізується в байти JSON. Цей крок серіалізації є дуже важливим, оскільки RabbitMQ має справу з двійковими повідомленнями. Метод отримує поле *replyTo* з властивостей оригінального запиту, яке вказує, куди слід надіслати відповідь.Створює *OutboundMessage* з відповідним *Exchannge’ом*, ключем маршрутизації та серіалізованою відповіддю і надсилає його назад запитувачу через RabbitMQ.

2.5 HTTP контролер

Створення Sping WebFlux, частково пов'язано з потребою у неблокувальному вебстеку, який може працювати паралельно з невеликою кількістю потоків і масштабуватися з меншою кількістю програмних ресурсів. Це стало мотивацією для створення нового загального API, який слугуватиме основою для будь-якого неблокуючого середовища виконання. Це важливо через наявність серверів, таких як Netty, які добре зарекомендували себе в асинхронному та неблокуючому світі. Інша частина відповіді — функціональне програмування. Подібно до того, як додавання анотацій в Java 5 створило нові можливості, наприклад, анотовані REST-контролери або модульні тести, додавання лямбда-виразів в Java 8 створило можливості для функціональних API на Java. Це є перевагою для неблокуючих додатків та API у реактивному стилі, які дозволяють декларативно компонувати асинхронну логіку.

На цій основі Spring WebFlux надає вибір між двома моделями програмування:

- Анотовані контролери: Узгоджується з Spring MVC і базується на тих самих анотаціях з модуля spring-web. І Spring MVC, і WebFlux контролери підтримують реактивні (Reactor і RxJava) типи повернення, і, як наслідок, їх нелегко відрізнити. Однією з помітних відмінностей є те, що WebFlux також підтримує реактивні аргументи @RequestBody.
- Функціональні контролери: Легка і функціональна модель програмування на основі лямбда-функцій. Різниця з анотованими контролерами полягає в тому, що додаток відповідає за обробку запитів від початку до кінця, а не декларує функцію за допомогою анотацій і чекає на зворотний виклик. Для цього використовуються *HandlerFunction і RouterFunctions*. *HandlerFunction* (рис. 2.5.1) є функціональним інтерфейсом, який генерує відповіді на запити, що надходять до неї.

Цей інтерфейс також являє собою функцію *Function<Request, Response<T>>*, яка поводиться дуже схоже на *Servlet*. Хоча, у порівнянні зі стандартним *Servlet#service(ServletRequest req, ServletResponse res)*, HandlerFunction не приймає відповідь як вхідний параметр.

```
@FunctionalInterface
public interface HandlerFunction<T extends ServerResponse> {
    Mono<T> handle(ServerRequest request);
}
```

Рис. 2.5.1)

RouterFunction слугує альтернативою анотації *RequestMapping* і використовується її для маршрутизації запитів до функцій-обробників. Зазвичай можна імпортувати допоміжну функцію RouterFunctions.route() (рис. 2.5.2) для створення маршрутів, замість того, щоб писати повну функцію маршрутизації.

```
@Bean
RouterFunction<ServerResponse> routerFunction(AuthHandler handler) {
    return route()
            .POST("api/v1/auth/sign-up", accept(APPLICATION_JSON), handler::registerNewUser)
            .POST("api/v1/auth/sign-in", accept(APPLICATION_JSON), handler::loginUser)
            .POST("api/v1/auth/verify", accept(APPLICATION_JSON), handler::verifyToken)
            .build();
}
```

Рис. 2.5.2

Вона дозволяє маршрутизувати запити, застосовуючи предикат RequestPredicate. Коли предикат збігається, повертається другий аргумент -- функція-обробник. Після декларування ендпойнтів, потрібно визначити хендлери, які будуть обробляти запити (рис. 2.5.3). З прикладу можна прослідкувати певну схожість з використанням анотацій.

```java
@Component
public class AuthHandler {

    private final ValidationService validator;
    private final AuthService authService;

    public Mono<ServerResponse> registerNewUser(ServerRequest request) {
        Mono<RegisterRequest> registerRequestMono = request.bodyToMono(RegisterRequest.class);
        return registerRequestMono
                .doOnNext(r -> validator.validate(r, "register_request"))
                .flatMap(authService::registerNewUser)
                .flatMap(response -> ServerResponse.status(response.getStatus()).build())
                .onErrorResume(this::sendErrorResponse);
    }

  // ...
}
```

Рис. 2.5.3

Метод *registerNewUser* призначений для обробки HTTP POST запитів на реєстрацію користувачів

- *request.bodyToMono(RegisterRequest.class)*: Витягує тіло JSON з вхідного запиту і перетворює його в Mono<RegisterRequest>, який є реактивним типом, що зрештою видасть один об'єкт RegisterRequest або жодного.
- *.doOnNext(r -> validator.validate(r, «register_request»)):* Перевіряє десеріалізований *RegisterRequest*. Метод *doOnNext* дозволяє застосовувати побічні оператори, такі як валідація, до елементу, що виводиться. Він не змінює потік, але може запустити логіку перевірки, яка може викликати помилку, якщо запит є недійсним.
- *.flatMap(authService::registerNewUser)*: Перетворює *RegisterRequest* в *Mono<Response>* шляхом виклику методу з *AuthService*. На цьому кроці виконується бізнес-логіка реєстрації нового користувача, яка включає операції з базою даних та зовнішні виклики API.

- *.flatMap(response -> ...)*: Перетворює результат реєстрації в HTTP-відповідь, статус якої встановлюється динамічно на основі результату процесу реєстрації.
- *.onErrorResume(this::sendErrorResponse):* Обробляє будь-які помилки, що виникають під час обробки запиту. Якщо виникає помилка, він перенаправляється до методу *sendErrorResponse* для генерації відповідної HTTP-відповіді.

2.6 WebSocket з'єднання

WebSocket - це протокол, що забезпечує повнодуплексні канали зв'язку через одне TCP-з'єднання. Цей протокол забезпечує взаємодію між веббраузером або іншим клієнтським додатком і вебсервером з меншими накладними витратами, що дозволяє створювати більш інтерактивні додатки в режимі реального часу. Spring WebFlux підтримує WebSockets і надає спосіб реактивної обробки з'єднань.

HandlerMapping (рис. 2.6.1) спрямовує запити на з'єднання WebSocket до певних обробників на основі шляху до URL-адреси. Шлях */api/v1/updates* асоціюється з *WSHandler*, себто будь-які з'єднання WebSocket, зроблені до цього ресурсу, обробляються *WSHandler*. Порядок дорівнює -1, щоб надати цьому зіставленню пріоритет над іншими потенційними зіставленнями.

```
@Bean
public HandlerMapping handlerMapping(WSHandler WSHandler) {
    Map<String, WebSocketHandler> handlerByPathMap = new HashMap<>();
    handlerByPathMap.put("/api/v1/updates", WSHandler);

    SimpleUrlHandlerMapping handlerMapping = new SimpleUrlHandlerMapping();
    handlerMapping.setUrlMap(handlerByPathMap);
    handlerMapping.setOrder(-1);

    return handlerMapping;
}
```

Рис. 2.6.1

AuthRequestUpgradeStrategy (рис. 2.6.2) - це спеціальний компонент покращення HTTP-запитів до з'єднань WebSocket.

```java
@Slf4j
@Component
public class AuthRequestUpgradeStrategy implements RequestUpgradeStrategy {

    @Override
    public Mono<Void> upgrade(ServerWebExchange exchange, WebSocketHandler handler, @Nullable
                              String subProtocol, Supplier<HandshakeInfo> handshakeInfoFactory) {
        ServerHttpResponse response = exchange.getResponse();
        HttpServerResponse reactorResponse = getNativeResponse(response);
        HandshakeInfo handshakeInfo = handshakeInfoFactory.get();
        NettyDataBufferFactory bufferFactory = (NettyDataBufferFactory) response.bufferFactory();

        HttpCookie accessToken = handshakeInfo.getCookies().getFirst("access_token");
        return Mono.justOrEmpty(accessToken)
                // обробка токену
                .flatMap(authRes -> {
                    if (authRes.getStatus() != HttpStatus.OK.value()) {
                        response.setRawStatusCode(authRes.getStatus());
                        return response.setComplete();
                    }

                    return reactorResponse.sendWebsocket((in, out) -> {
                        final var session = new ReactorNettyWebSocketSession(in, out,
                                                                             handshakeInfo,
                                                                             bufferFactory,
                                                                             this.maxFramePayloadLength);
                        session.getAttributes().put("aggregate_id", authRes.getData());
                        return handler.handle(session)
                                .contextWrite(ctx -> ctx.put("aggregate_id", authRes.getData()));
                    });
                })
                .onErrorResume((ex) -> {
                    // ...
                })
                .then();
    }
}
```

Рис. 2.6.2

Вона включає логіку автентифікації, гарантуючи, що лише автентифіковані користувачі можуть встановлювати з'єднання WebSocket. Компонент перевіряє наявність cookie-файлу access_token, значення якого використовується для автентифікації користувача за допомогою AuthService. Якщо автентифікація успішна, то цей компонент записує в реактивний контекст унікальний ідентифікатор користувача.

WSHandler обробляє вхідні повідомлення, застосовує бізнес-логіку та надсилає відповіді (рис. 2.6.3).

```java
@Component
public class WSHandler implements WebSocketHandler {
    // Services and active sessions management
    private final ConcurrentMap<String, Set<WebSocketSession>> activeSessions;

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        return Flux.merge(
                        Flux.deferContextual(ctx -> {
                                    final String currentUserId = ctx.get("aggregate_id");
                                    activeSessions.putIfAbsent(currentUserId, ConcurrentHashMap.newKeySet());
                                    activeSessions.get(currentUserId).add(session);
                                    return Mono.just(currentUserId);
                                }).flatMap(currentUserId -> userStatusService.updateUserStatus(currentUserId, true)
                                        .then(Mono.just(currentUserId)))
                        session.receive()
                                .<WSRequest>handle((requestRaw, sink) -> {
                                    // handling of incoming messages
                                })
                                .thenMany(Flux.empty())
                ).doFinally(ignored -> {
                    activeSessions.getOrDefault(userId, Set.of()).remove(session);
                    userStatusService.updateUserStatus(userId, false)
                            .subscribeOn(Schedulers.boundedElastic())
                            .subscribe();
                })
                .then();
    }
}
```

Рис. 2.6.3

Він має мапу активних сесій, що дозволяє надсилати оновлення певним користувачам на основі подій, що відбуваються у системі. Повідомлення десеріалізуються та обробляються. Обробник також включає обробку помилок для повідомлень і закриття сеансів.

Реалізація WebSockets у Spring WebFlux за допомогою функціонального API дозволяє створювати вебдодатки в режимом реального часу. Наведені приклади ілюструють налаштування обробників WebSocket, аутентифікацію з'єднань та ефективне керування сесіями WebSocket.

Розділ 3. Огляд застосунку

Цей розділ досліджує шлях створення системи управління завданнями “Tasker”, з використанням інноваційної, реактивної моделі з використанням Spring WebFlux особливих практик. Кінцевий результат буде складатися з бекенду та фронтенду, які мають комунікати у реальному часі та висвітлювати для користувачів найновіші дані.

3.1 Архітектура застосунку

Робота починається з мікромоноліту, який являє собою проєкт, поділений на модулі. Кожен модуль може мати власні залежності, конфігурацію та архітектуру. Основою вимогою такого архітектури є те, що ці модулі не мають напряму використовувати інші модулі, які є частиною проєкту. Себто, все спілкування між модулями, відбувається за допомогою черги повідомлень, або інтерфейсів, що після розділення на окремі мікросервіси, будуть реалізовані через мережеве API, для прикладу, gRPC. Таке архітектурне рішення значно полегшить розробку системи, та збереже всі переваги моноліту та мікросервісу. Для прикладу, мікромоноліт легше запускати, підтримувати та розробляти однією невеликою командою. Після закінчення роботи над проєктом можна без труднощів перейти на мікосервісну архітектуру, де система вже буде повністю дистрибутивною та легко масштабуватися відповідно до навантаження.

На першій сходинці, проєкт складається з трьох модулів: *application*, *common* і *config* (рис. 3.1.1).

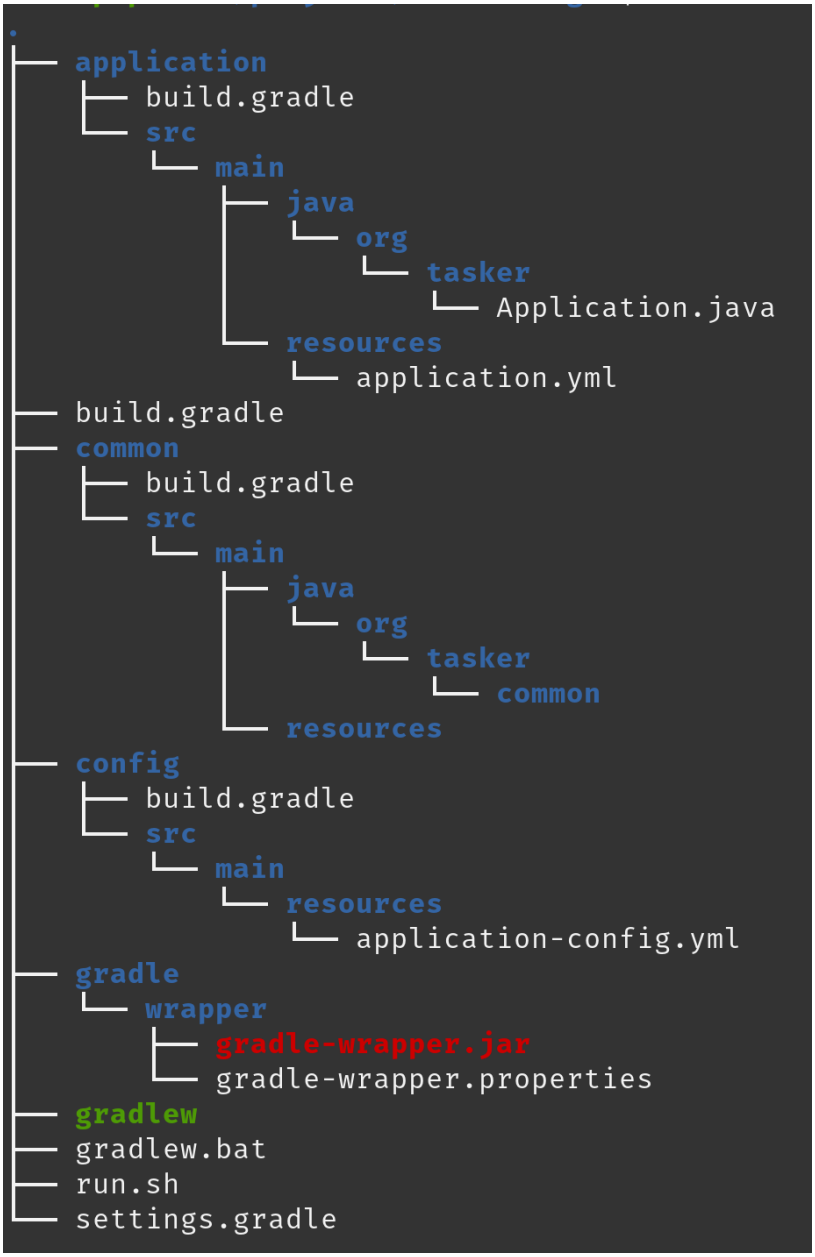

Рис. 3.1.1

Модуль *application* - це точка входу для всього застосунку. У ньому імпортуються всі інші модулі, які аналізуються Spring'ом та запускається через *SpringBootApplication*. Заради того, аби підтримувати чистоту коду, існує *common* модуль, який відповідальний за розміщення будь-яких елементів, які потрібні для декількох інших модулів. Модуль *config* зберігає глобальну конфігурацію проєкту в *application-config.yaml* файлі. Таким чином цей проєкт може легко розростатися з появою новими модулями, які у майбутньому стануть окремими мікросервісами.

На рис. 3.1.2 показано кінцеву архітектуру проєкту.

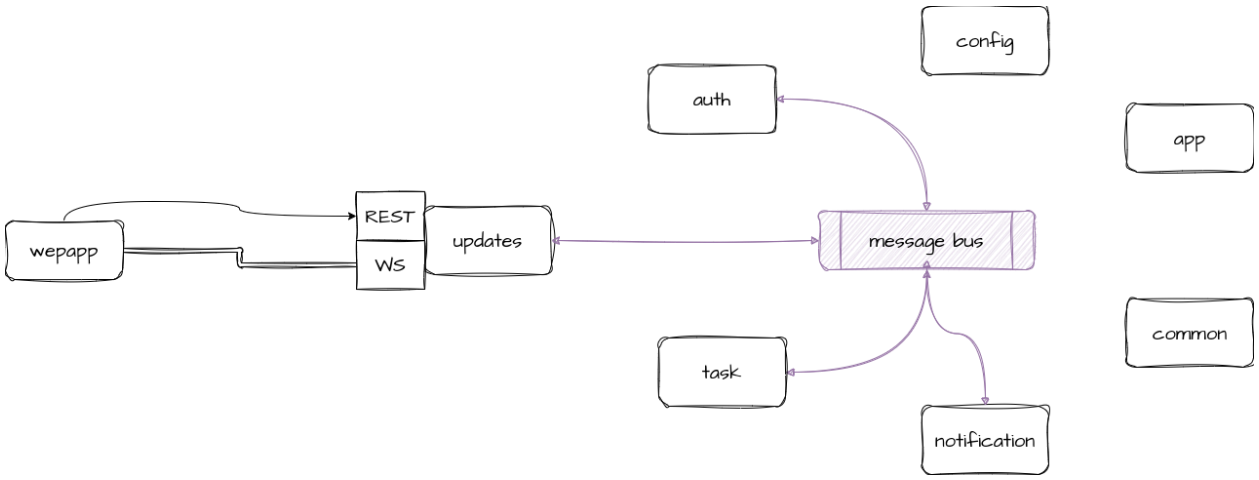

Рис. 3.1.2

У ньому присутні такі модулі, окрім тих що були початково згадані та не несуть бізнес-логіки:

- *auth* — модуль, який відповідальний за авторизацію та автентифікацію користувачів у застосунку. Також цей модуль несе відповідальність за реєстрацію користувачів.
- *notification* - модуль, що відповідальний за створення, надсилання та інвалідизації сповіщень для користувачів про зміни стану дошок та їхніх завдань.
- *task* — модуль що відповідальний за основну бізнес-логіку застосунку. А саме цей сервіс забезпечує CRUD-функцінал для дошок, завдань та запрошень до дошки.
- *updates* — модуль що відповідальний за всю комунікацію зі зовнішнім світом через REST та WebSocket з’єднання.
- *wepapp* — окремий модуль що репрезентує вебзастосунок з графічним інтерфейсом для користувачів. Він з’єднується з модулем *updates* для комунікації через WebSocekt-з’єднання.

Вся комунікація між сервісами, окрім *wepapp-м*одуля, відбувається через чергу повідомлень, асинхронно або за допомогою взірця Request/Reply. Вибір між синхронним та асинхронним методом комунікації відіграє чи не найважливішу роль у проєктуванні системи. У нашому випадку, система має сповіщати про нові зміни щойно вони зареєструвалися в системі. З такою бізнес-вимогою, вибір впав на

асинхронну комунікацію. У такому способі спілкування, сервіс А надсилає повідомлення на сервіс Б і не чекаючи відповідь, продовжує виконувати свої справи. Своєю чергою сервіс Б отримує на опрацювання повідомлення та може надіслати повідомлення про зміни. У випадках коли все ж потрібна синхронна комунікація, використовується взірець Request/Reply, який показано у минулому розділі.

3.2 Взірець Event Sourcing, як єдине джерело правди

Event Sourcing — це підхід, за яким, застосунок зберігає всю історію змін, які сталися з доменним об’єктом або агрегатом. Події, або коміти, це не просто тимчасові логи, вони розцінюються як послідовна історія змін агрегата, що є єдиним джерелом правди у застосунку. Така концепція дуже схожа на систему версіювання *git*, де замість кінцевого стану, зберігаються зміни які були внесені у файл. Окрім попереднього прикладу, *Event Sourcing*, використовується у більшості серйозних застосунках. Як сказав Грег Янг в одному з виступів про даний підхід [12], баланс який ви бачите у себе на картці, це не просто значення з рядка бази даних, це сума всіх транзакцій, що були здійснені.

Усі події що сталися у застосунку є незмінними, тобто ми не можемо оновити подію, яка вже сталася. Натомість ми можемо створити нову подію яка внесе потрібні зміни. Події можна тільки додавати у сховище подій, але видаляти категорично не рекомендується. Ще одним важливим моментом є те що події можуть підтримувати версіювання, це схоже до того як в API є декілька версій, які різняться структурою запитів чи відповідей. Так само й події можуть мати кілька версій, у який змінюються структура відповідно до нових вимог.

На рис. 3.2.1 показано подію оновлення статусу завдання.

```java
package org.tasker.common.models.event;

// ...

@Data
@EqualsAndHashCode(callSuper = false)
public class TaskStatusUpdatedEvent extends BaseEvent {

    public static final String TASK_STATUS_UPDATED_V1 = "TASK_STATUS_UPDATED_V1";
    public static final String AGGREGATE_TYPE = TaskAggregate.AGGREGATE_TYPE;

    private String boardId;
    private TaskStatus status;

    @Builder
    public TaskStatusUpdatedEvent(String aggregateId, String boardId, TaskStatus status) {
        super(aggregateId);
        this.status = status;
        this.boardId = boardId;
    }
}
```

Рис. 3.2.1

Подія TaskStatusUpdatedEvent містить лише ті поля, які потрібні у її контексті. У прикладі — це сам статус та ідентифікатор дошки. Таких подій для одного завдання може бути безліч. Щоб оновити поточний статус, застосунок повинен витягнути всі події які здійснилися з завданнями та застосувати їх до агрегату, а тільки після цього, він може провести валідування та оновити статус. Така процедура може займати досить багато часу. З метою оптимізації вводиться поняття *snapshot*, проміжний зріз, який містить стан після N кількості подій. Щоб отримати фінальний стан, застосунку вже не потрібні всі події, достатньо найновішого снепшота та подій що сталися після нього.

У Додатку 6 показно клас Event, абстрактний клас *AggregateRoot*, та його реалізацію на прикладі агрегату завдання. Для того, щоб застосувати подію до агрегату потрібно викликати метод *when(final Event event)*, який є різним для конкретної імплементації *AggregateRoot*. Щоб створити нову подію, викликається відповідний метод агрегату. Щоб оновити статус завдання викликається метод *updateStatus(TaskStatus status)*, який створює відповідну подію та додає до списку. Останнім кроком є збереження цієї події до Event Store. У застосунку для цього використовується PostgreSQL, що чудово підходить для цієї ролі, через ефективність запису.

У Додатку 7 показано інтерфейс *EventStore* та його конкретну реалізацію *EventStoreImpl.* Вона дозволяє гнучко працювати з агрегатами, не прив’язуючись до конкретного типу. Спершу розглянемо приклад збереження, що відповідає методу save(T aggregate). Даний метод використовує, вже згаданий у минулому розділі, метод *saveEvents.* Після чого зберігає снепшот, якщо версія кратна певному значенню. Також цікава реалізація витягування агрегати з бази даних — load(String aggregateId, Class<T> aggregateType). Метод намагається отримати снепшот, якщо такого немає, то створює “пустий” агрегат. Після цього витягує всі події які сталися після версії снепшоту та застосовує їх до агрегату. Важливо також зазначити, що після збереження подій вони надсилаються до черги повідомлень, це буде використано згодом.

Разом з *ES,* зазвичай, використовуються взірець *SQRS,* що не є випадковістю. Адже використовувати ES без SQRS вигідно при малій кількості даних, але у більш серйозних застосунках це є необхідністю. SQRS є логічним продовженням взірця CQS, який заснований на принципі “розділяй та пануй” і вперше описаний Бертраном Мейєром. CQS полягає в тому, щоб розділити методи на модифікацію (команди) та читання (запити) на окремі потоки. Таким чином після виконання команд умовно нічого не повертається, але в окремих випадах може повертатися ідентифікатор об’єкта. Такий підхід дає змогу відокремити частини коді які потрібні для специфічних дій.

3.3 Взірець CQRS

CQRS є еволюцією CQS описаний Грегом Янгом і Джонатаном Олівером. Однією з причин появи цього взірця це нерівномірний розподіл навантаження на підсистеми читання і запису. Цей нерівномірний розподіл своєю чергою від того, що бізнес-правила або бізнес-логіка зосереджені на етапі запису інформації й одночасно з цим запити на багато частіше використовуються, ніж команди. Впровадження CQRS збільшує складність архітектури системи, що потенційно може призвести до збільшення витрат на розробку та обслуговування. CQRS часто використовує різні моделі для запису та читання, своєю чергою це підвищує гнучкість, оскільки ми можемо використовувати тільки ті поля які дійсно потрібні.

Якщо з якоїсь причини щось пішло не так, сховище подій можна використати, щоб повернутися назад у часі, відтворити події і з'ясувати проблему. Будь-яку

подію можна відтворити у будь-який час, щоб змінити або виправити щось у системі, або відновити минулий стан за допомогою повторного відтворення. З такими можливостями система стає дуже зручною для розв'язання проблем.

Розглянемо приклад використання CQRS на прикладі оновленні статусу завдання починаючи від дій користувача. Приклад схематично зображено на рис. 3.2.2.

1. Першим кроком, *webapp*-клієнт відправляє запит *UpdateTaskStatusRequest* на оновлення статусу до updates-подуля через WebSocket з’єднання.

2. Updates-модуль перетворює запит на команду *UpdateTaskStatusCommand*, яка містить окрім всіх тих полів що й запит, але ще й ідентифікатор користувача для авторизації, та надсилає її до черги повідомлення. Іншими словами, сервіс який буде опрацьовувати цю команду має перевірити чи справді цей користувач має доступ до завдання.

3. З черги повідомлення, команда потрапляє до модуля *task,* де вона опрацюється на кілька рівнів, починаючи від споживача до *EventStore*. Результатом опрацювання є подія *TaskStatusUpdatedEvent*.

4. Створена подія *TaskStatusUpdatedEvent,* серіалізується у список байтів та зберігається у PostgreSQL.

5. Збережена подія *TaskStatusUpdatedEvent* відправляється у чергу повідомлення, де може бути отримана будь-яким компонентом, який у цьому зацікавлений.

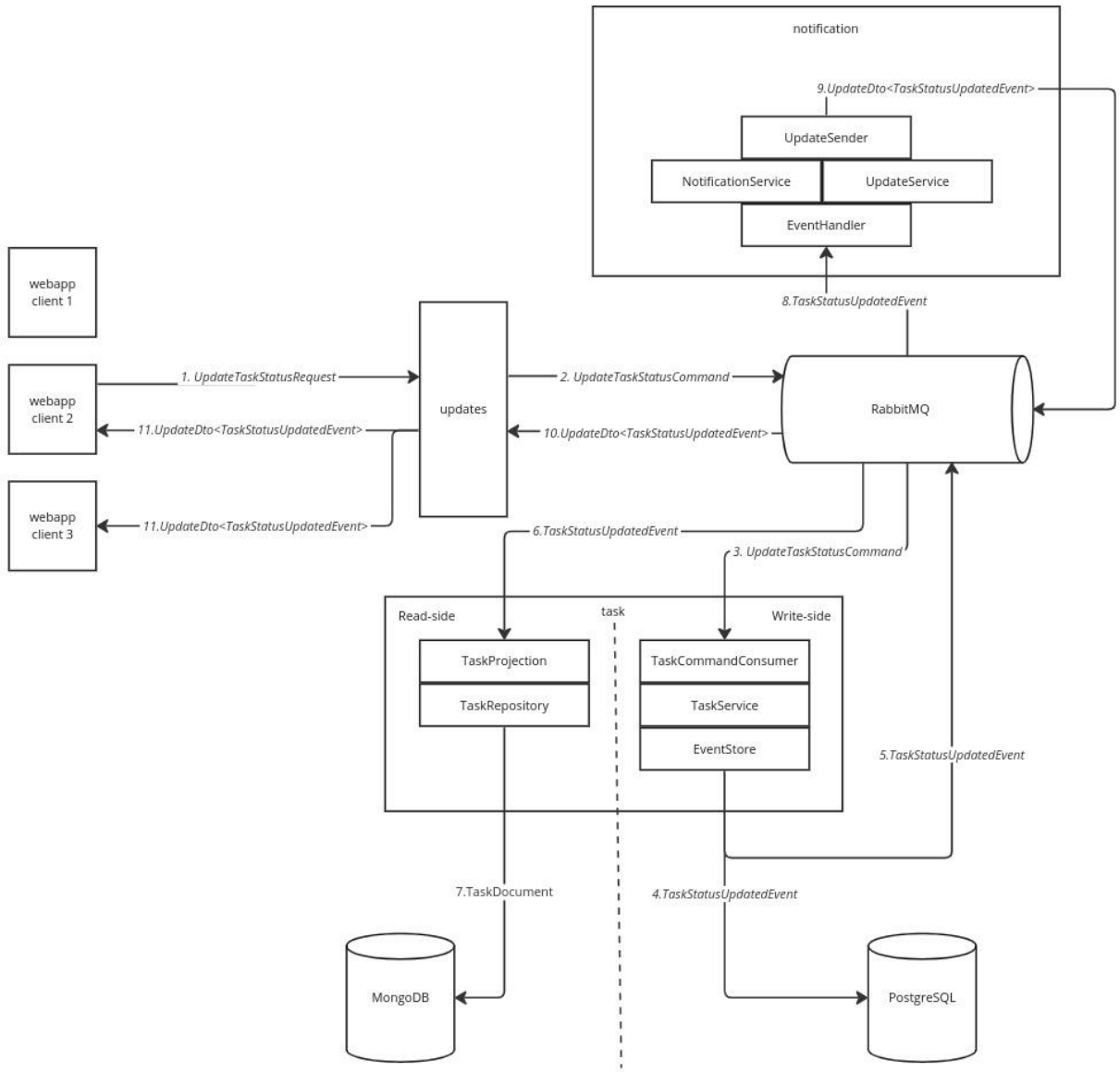

Рис. 3.2.2

6. Одним з зацікавленим компонентом є TaskProjection. Концепція проєкцій є чимось схожим до терміну в контексті геометрії, де тривимірна фігура проєктується на площину як двовимірна. У контексті ES, проєкція обробляє модель для читання, тобто оновлює дані. Особливістю проєкція є те, що їх може бути скільки завгодно і вони можуть обробляти які завгодно події, таким чином можна створити різні моделі даних під різні потреби. Для прикладу TaskProjection може отримувати події які не зв’язані з самими завданням, але потрібна для того аби створити зручну модель для читання.

7. TaskRepository оновлює документ завдання.

8. Окрім проєкції, *TaskStatusUpdatedEvent* потрапляє до модуля *notification*. Цей крок не обов'язково йде за 6-м, оскільки вся комунікація асинхронна, то повідомлення може потрапити у модулі у будь-якому порядку.

9. Після потрапляння у цей сервіс, подія обробляється та перетворюється в *UpdateDto* (рис. 3.3.3). Цей модуль відповідальний за те яким користувачам надсилати сповіщення та оновлення про подію, що відбулася у системі. Далі цей DTO-об'єкт потрапляє до черги повідомлення.

10. *UpdateDto* потрапляє до модуля *updates.*

11. Модуль *updates* надсилає сповіщення та оновлення про змінений статус, всім користувачам, що були вказані попереднім сервісом.

Після цього прикладу, зрозуміло як працює поєднання *CQRS* та *Event Sourcing.* Окрім них, ці модулі використовують реактивне програмування, розам з яким, це схоже на суцільний трубопровід, по якому перетікають події, що є дуже зручним для розуміння та розростання застосунку. Також можна помітити що всі сервіси працюють незалежно один від одного і можуть легко масштабуватися відповідно до навантаження на окремий сервіс.

Важливо розуміти, будь-яка система не застрахована від помилок і дуже важливо правильно їх опрацьовувати. Для прикладу, коли у модулі *task*, виникає помилка або користувач не має прав виконати певну дію, сервіс надсилає повідомлення до RabbitMQ з тим самим ідентифікатором кореляції, аби webapp-клієнт зміг зіставити початковий запит та повідомлення про помилку.

Всі інші компоненти застосунку, створені аналогічно до вищенаведеного прикладу, з однією відмінністю, що вони виконують різну бізнес-логіку. Для прикладу авторизація користувача відбувається за аналогічним процесом за допомогою взірця Request/Reply. Для прикладу, команда яка буде опрацьовуватися auth-модулем, потрапить до нього, порівняє дані з read-моделі та поверне результат до черги про успішність чи помилку процесу.

3.4 Огляд застосунку

У застосунку “Tasker”, функціонал не закінчується на оновленні статусу завдання. У ньому присутні безліч функцій які будуть розглянуті. Почати варто з сервісу webapp. Цей сервіс створений за допомогою Javascript-фраймфорку — Vue.

Для початку у застосунку можна зареєструватися (рис. 3.4.1) та увійти в наявний акаунт (рис. 3.4.2)

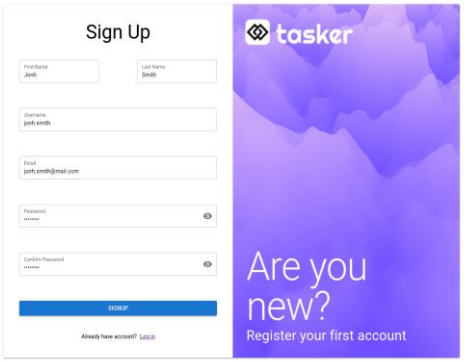

Рис. 3.4.1

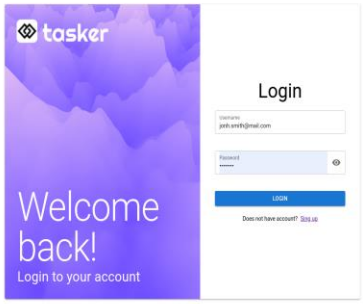

Рис. 3.4.2

.

Після успішної авторизації, користувач потрапляє на головну сторінку (рис. 3.4.3), де розміщена статистика завдань, зібрана зі всіх дошок, які він створив або до яких приєднався.

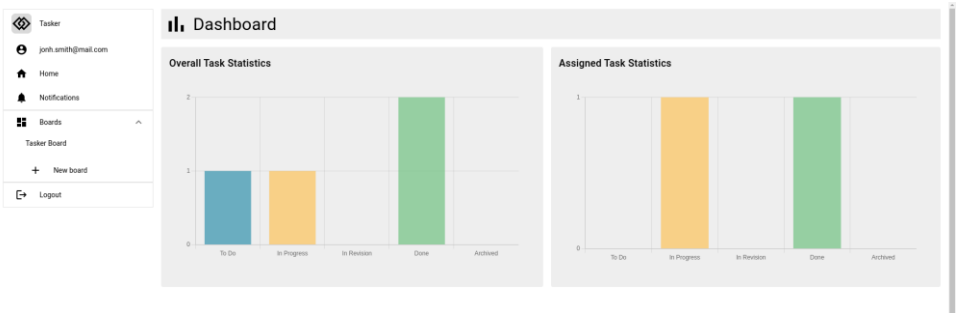

Рис. 3.4.3

У застосунку можна створювати, редагувати або видаляти дошки. Дошка — це місце де зібрано всі завдання для окремого проєкту чи команди. До неї можна запрошувати інших користувачів та працювати у реальному часі. На рис. 3.4.4

зображено створення нової дошки, до якої запрошені користувачі user_1 та user_2.

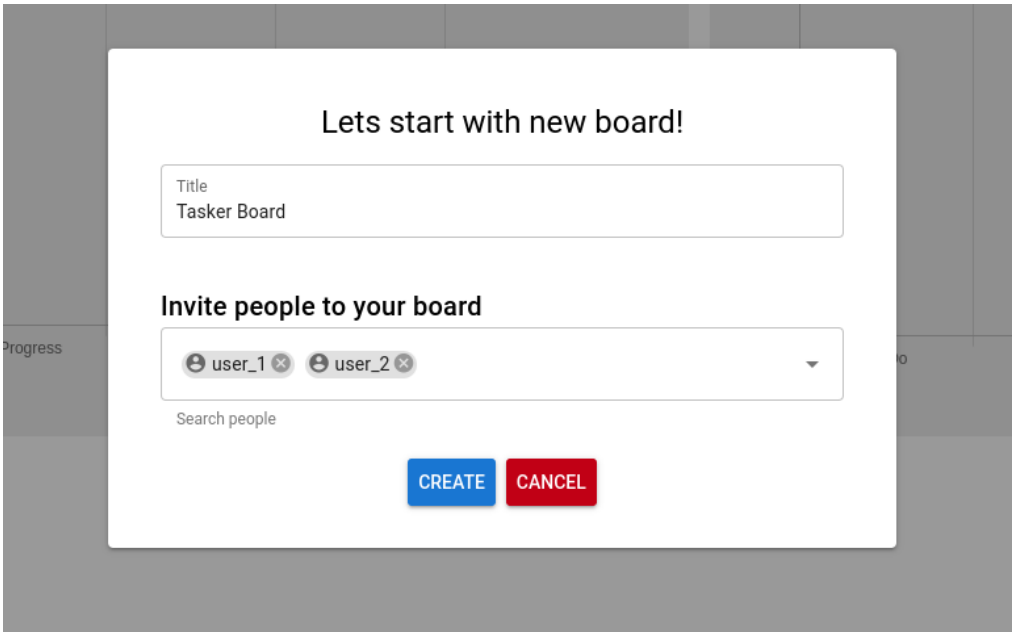

Рис. 3.4.4

Після створення першої дошки, користувачі вже можуть почати працювати зі своїм проєктом. Для того аби іншим користувачам приєднатися до дошки, потрібно прийняти запрошення, що знаходиться на сторінці зі сповіщеннями, як показано на рис. 3.4.5.

Рис. 3.4.5

Дошка містить всі завдання згруповані за статусом (рис 3.4.6).

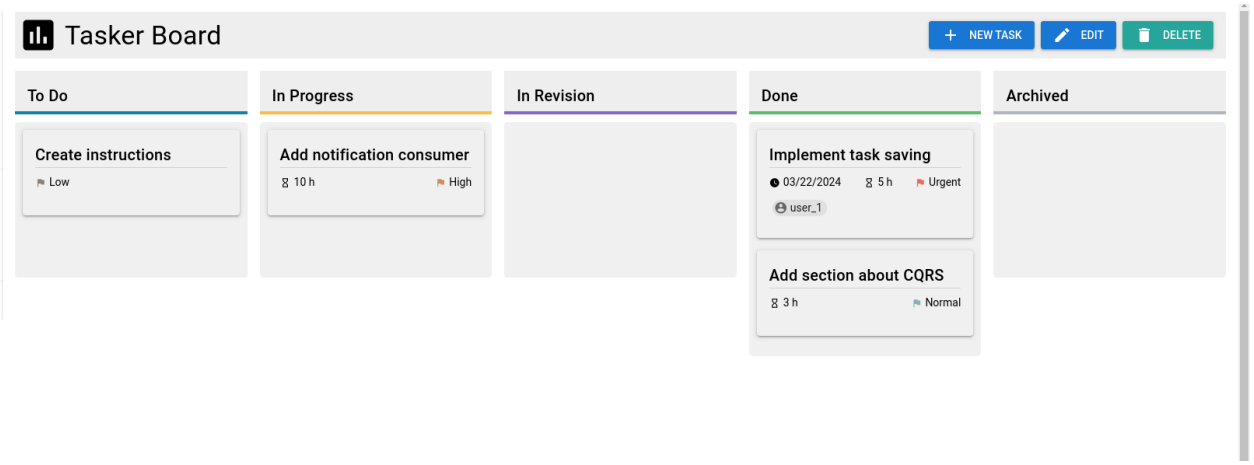

Рис. 3.4.6

Користувачі можуть редагувати, видаляти та додавати завдання. Також кожне завдання може мати такі поля: назва, опис, час на виконання, дата початку та закінчення, пріоритетність та статус. Щоб змінити статус завдання варто лише перенести його з одної колонки в інше. Перевагою цього застосунку є те, що всі дані оновлюється у реальному часі, тому коли один користувач змінити опис завдання чи його статус, інший одразу може побачити зміни. Ще однією перевагою є відсутність будь-яких обмежень на кількість, користувач може створити необмежену кількість дошок та завдань. Для безпечності дошки, тільки власник може редагувати чи видаляти власну дошку, а також запрошувати чи видаляти користувачів з неї.

На сторінці сповіщень, користувач може побачити всю діяльність пов’язану з її дошками або завданнями на які він був поставлений (рис. 3.4.7).

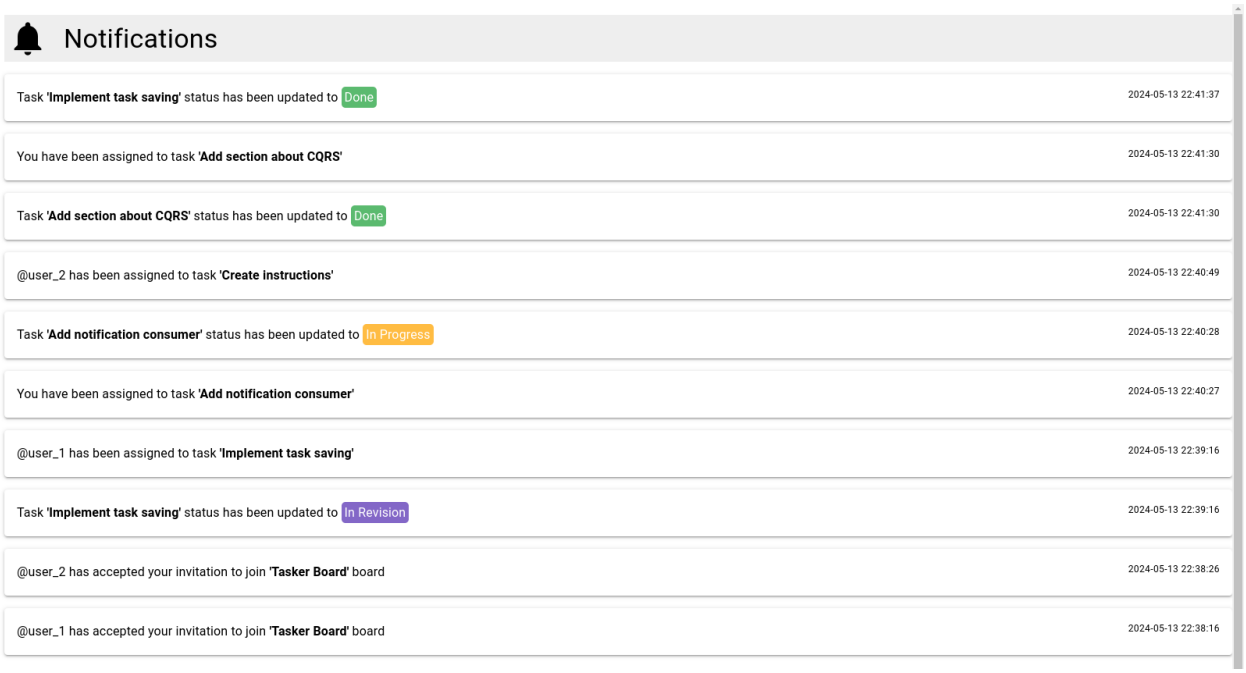
Notifications
Task 'Implement task saving' status has been updated to Done
2024-05-13 22:41:37
You have been assigned to task 'Add section about CQRS'
2024-05-13 22:41:30
Task 'Add section about CQRS' status has been updated to Done
2024-05-13 22:41:30
@user_2 has been assigned to task 'Create instructions'
2024-05-13 22:40:49
Task 'Add notification consumer' status has been updated to In Progress
2024-05-13 22:40:28
You have been assigned to task 'Add notification consumer'
2024-05-13 22:40:27
@user_1 has been assigned to task 'Implement task saving'
2024-05-13 22:39:16
Task 'Implement task saving' status has been updated to In Revision
2024-05-13 22:39:16
@user_2 has accepted your invitation to join 'Tasker Board' board
2024-05-13 22:38:26
@user_1 has accepted your invitation to join 'Tasker Board' board
2024-05-13 22:38:16

Рис. 3.4.7

Висновки

Отже, у цій роботі, ми ознайомилися з теоретичною базою про реактивне програмування, а саме про базові абстракції, модель виконання, ліфтинг, уникнення збоїв, багатовекторність та підтримка дистрибутивності. Також на основі цих параметрів порівняли три найпопулярніших бібліотеки для реактивного програмування на Java.

Після теоретичного екскурсу, ми обрали одну бібліотек — з Project Reactor, для огляду найпопулярніших компонентів, такі як різні бази даних чи черги повідомлень. Окрім цього, робота показала, що автоконфігурація мало чим відрізняється від тої, де використовуються імперативний підхід. Це своєю чергою доводить легкість використання реактивних інструментів.

Протягом цієї роботи, було створено застосунок управління завданнями, що показав переваги асинхронної комунікації для легкої масштабованості та розростання проєкту. Для цього було використано два потужні взірці - SQRS та Event Sourcing.

Використані джерела

1. Thompson S. Applications of Fran [Електронний ресурс] / Simon Thompson. – 1999. – Режим доступу до ресурсу: https://www.cs.kent.ac.uk/people/staff/sjt/Fran/.

2. Synchronous Languages and Reactive System Design [Електронний ресурс] // IFAC Infonnation Control in Manufacturing, Nancy - Metz, France – 1999 – Режим доступу до ресурсу: https://www.sciencedirect.com/science/article/pii/S147466701740526X

3. A Survey on Reactive Programming [Електронний ресурс] / Andoni Lombide Carreton, Tom Van Cutsem // ACM Computing Surveys – 2012 – Режим доступу до ресурсу: https://www.researchgate.net/publication/233755674_A_Survey_on_Reactive_Programming

4. Reactive Streams [Електронний ресурс] / Ben Christensen, Roland Kuhn – Режим доступу до ресурсу: https://www.reactive-streams.org/

5. RxJava: Reactive Extensions for the JVM [Електронний ресурс] // ReactiveX – Режим доступу до ресурсу: https://github.com/ReactiveX/RxJava

6. Optimizing the Netflix API [Електронний ресурс] ./ Netflix Technology Blog – 2013 Режим доступу до ресурсу: https://netflixtechblog.com/optimizing-the-netflix-api-5c9ac715cf19

7. Introduction to Data Streaming with RxJava [Електронний ресурс] / Dharesh Vadalia – 2020 – Режим доступу до ресурсу: https://dhareshvadalia.medium.com/introduction-to-data-streaming-with-rxjava-c2c13a2c1f0d

8. Analysing the performance and costs of reactive programming libraries in Java [Електронний ресурс] / Julien Ponge, Arthur Navarro, Clément Escoffier, Frédéric Le Mouël – 2021 – Режим доступу до ресурсу: https://www.researchgate.net/publication/355349305_Analysing_the_performance_and_costs_of_reactive_programming_libraries_in_Java

9. Integrating Retrofit with RxJava [Електронний ресурс] / Predrag Marić // baeldung – 2024 – Режим доступу до ресурсу: https://www.baeldung.com/retrofit-rxjava

10. The Reactive Manifesto [Електронний ресурс] / Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson – 2014 – Режим доступу до ресурсу: https://www.reactivemanifesto.org/

11. Primer [Електронний ресурс] / MongoDB, Inc. – 2024 – Режим доступу до ресурсу: https://www.mongodb.com/docs/languages/java/reactive-streams-driver/current/get-started/primer/

12. Young G. CQRS and Event Sourcing [Електронний ресурс] / Greg Young // Code on the Beach. – 2014. – Режим доступу до ресурсу: https://www.youtube.com/watch?v=JHGkaShoyNs&t=39s&ab_channel=CodeontheBeach.

Додаток 1

```
import io.reactivex.rxjava3.core.Observable;
import io.reactivex.rxjava3.schedulers.Schedulers;

public class RxJavaExample {
    public static void main(String[] args) {
        Observable.fromCallable(() -> fetchUserFromDatabase("userId"))
            .subscribeOn(Schedulers.io()) // Asynchronously fetch from database
            .map(user -> user.toUpperCase()) // Transform user details
            .onErrorReturnItem("DEFAULT USER") // Fallback if error occurs
            .timeout(3, TimeUnit.SECONDS, Observable.just("TIMED OUT USER")) // Fallback if timeout
            .subscribe(System.out::println, Throwable::printStackTrace);
    }

    private static String fetchUserFromDatabase(String userId) {
        // Simulate database access
        return "John Doe";
    }
}
```

Додаток 2

```
import reactor.core.publisher.Mono;
import reactor.core.scheduler.Schedulers;

public class ReactorExample {
  public static void main(String[] args) {
    Mono.fromCallable(() -> fetchUserFromDatabase("userId"))
      .subscribeOn(Schedulers.boundedElastic()) // Asynchronously fetch from database
      .map(String::toUpperCase) // Transform user details
      .onErrorReturn("DEFAULT USER") // Fallback if error occurs
      .timeout(Duration.ofSeconds(3), Mono.just("TIMED OUT USER")) // Fallback if timeout
      .subscribe(System.out::println, Throwable::printStackTrace);
  }

  private static String fetchUserFromDatabase(String userId) {
    // Simulate database access
    return "John Doe";
  }
}
```

Додаток 3

```
import io.smallrye.mutiny.Uni;


public class MutinyExample {
   public static void main(String[] args) {
      Uni.createFrom().item(() -> fetchUserFromDatabase("userId"))
         .onItem().transform(String::toUpperCase) // Transform user details
         .onFailure().recoverWithItem("DEFAULT USER") // Fallback if error occurs
         .ifNoItem().after(Duration.ofSeconds(3)).recoverWithItem("TIMED OUT USER") // Fallback if timeout
         .subscribe().with(System.out::println, Throwable::printStackTrace);
   }

   private static String fetchUserFromDatabase(String userId) {
      // Simulate database access
      return "John Doe";
   }
}
```

Додаток 4

```
implementation 'org.springframework.boot:spring-boot-starter'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
compileOnly "org.projectlombok:lombok:${rootProject.lombokVersion}"
testCompileOnly "org.projectlombok:lombok:${rootProject.lombokVersion}"
annotationProcessor "org.projectlombok:lombok:${rootProject.lombokVersion}"
testAnnotationProcessor "org.projectlombok:lombok:${rootProject.lombokVersion}"
implementation 'org.springframework.boot:spring-boot-starter-amqp'
implementation "io.projectreactor.rabbitmq:reactor-rabbitmq:${rootProject.reactiveRabbitmqVersion}"
implementation "com.rabbitmq:amqp-client:${rootProject.rabbitmqVersion}"
implementation 'org.springframework.boot:spring-boot-starter-webflux'
implementation 'org.springframework.boot:spring-boot-starter-data-r2dbc'
implementation "org.postgresql:r2dbc-postgresql:${rootProject.r2dbcPostgresVersion}"
implementation "org.postgresql:postgresql:${rootProject.postgresVersion}"
implementation "io.r2dbc:r2dbc-pool:${rootProject.r2dbcPostgresVersion}"
implementation 'org.springframework.boot:spring-boot-starter-data-mongodb-reactive'
testImplementation 'io.projectreactor:reactor-test'
implementation 'org.springframework.boot:spring-boot-starter-amqp'
implementation 'org.springframework.boot:spring-boot-starter-webflux'
implementation "io.projectreactor.rabbitmq:reactor-rabbitmq:${rootProject.reactiveRabbitmqVersion}"
implementation "com.rabbitmq:amqp-client:${rootProject.rabbitmqVersion}"
```

Додаток 5

```
CREATE EXTENSION IF NOT EXISTS citext;

CREATE EXTENSION IF NOT EXISTS "uuid-ossp";


CREATE TABLE IF NOT EXISTS events

(

   event_id      UUID                  DEFAULT uuid_generate_v4(),

   aggregate_id  VARCHAR(250) NOT NULL CHECK ( aggregate_id <> '' ),

   aggregate_type VARCHAR(250) NOT NULL CHECK ( aggregate_type <> '' ),

   event_type    VARCHAR(250) NOT NULL CHECK ( event_type <> '' ),

   data          BYTEA,

   metadata      BYTEA,

   version       SERIAL      NOT NULL,

   created_at    TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,

   UNIQUE (aggregate_id, version)

) PARTITION BY HASH (aggregate_id);


CREATE INDEX IF NOT EXISTS aggregate_id_aggregate_version_idx ON events USING btree (aggregate_id, version ASC);


CREATE TABLE IF NOT EXISTS events_partition_hash_1 PARTITION OF events

   FOR VALUES WITH (MODULUS 3, REMAINDER 0);


CREATE TABLE IF NOT EXISTS events_partition_hash_2 PARTITION OF events

   FOR VALUES WITH (MODULUS 3, REMAINDER 1);


CREATE TABLE IF NOT EXISTS events_partition_hash_3 PARTITION OF events

   FOR VALUES WITH (MODULUS 3, REMAINDER 2);


CREATE TABLE IF NOT EXISTS snapshots

(
```

```
    snapshot_id    UUID PRIMARY KEY        DEFAULT uuid_generate_v4(),
    aggregate_id   VARCHAR(250) UNIQUE NOT NULL CHECK ( aggregate_id <> '' ),
    aggregate_type VARCHAR(250)       NOT NULL CHECK ( aggregate_type <> '' ),
    data           BYTEA,
    metadata       BYTEA,
    version        SERIAL             NOT NULL,
    created_at     TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (aggregate_id)
);

CREATE INDEX IF NOT EXISTS aggregate_id_aggregate_version_idx ON snapshots USING btree (aggregate_id, version);

-- create reservation tables
CREATE TABLE IF NOT EXISTS username_email_reservation
(
    aggregate_id VARCHAR(255) PRIMARY KEY,
    email       VARCHAR(255) UNIQUE,
    username     VARCHAR(255) UNIQUE
);
```

Додаток 6

Event.java:

```
package org.tasker.common.es;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.Arrays;
import java.util.UUID;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Event {

    private UUID id;
    private String aggregateId;
    private String eventType;
    private String aggregateType;
    private long version;
    private byte[] data;
    private LocalDateTime createdAt;

    public Event(String eventType, String aggregateType) {
        this.id = UUID.randomUUID();
        this.eventType = eventType;
```

```
        this.aggregateType = aggregateType;
        this.createdAt = LocalDateTime.now();
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Event event)) return false;

        if (version != event.version) return false;
        if (!id.equals(event.id)) return false;
        if (!aggregateId.equals(event.aggregateId)) return false;
        if (!eventType.equals(event.eventType)) return false;
        if (!aggregateType.equals(event.aggregateType)) return false;
        if (!Arrays.equals(data, event.data)) return false;
        return createdAt.truncatedTo(ChronoUnit.SECONDS)
                .isEqual(event.getCreatedAt().truncatedTo(ChronoUnit.SECONDS));
    }

    @Override
    public int hashCode() {
        int result = id.hashCode();
        result = 31 * result + aggregateId.hashCode();
        result = 31 * result + eventType.hashCode();
        result = 31 * result + aggregateType.hashCode();
        result = 31 * result + (int) (version ^ (version >>> 32));
        result = 31 * result + Arrays.hashCode(data);
        result = 31 * result + createdAt.hashCode();
        return result;
    }
}
```

AggregateRoot.java:

```
package org.tasker.common.es;

import lombok.Data;
import lombok.Getter;
import lombok.NoArgsConstructor;
import org.tasker.common.exceptions.InvalidEventException;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
@Data
@Getter
@NoArgsConstructor
public abstract class AggregateRoot {

   protected final List<Event> changes = new ArrayList<>();
   protected String id;
   protected String type;
   protected long version;

   public AggregateRoot(final String id, final String aggregateType) {
      this.id = id;
      this.type = aggregateType;
   }

   public abstract void when(final Event event);

   public void load(final List<Event> events) {
```

```
    events.forEach(event -> {
        this.validateEvent(event);
        this.raiseEvent(event);
        this.version++;
    });
}

public void apply(final Event event) {
    this.validateEvent(event);
    event.setAggregateType(this.type);

    when(event);
    changes.add(event);

    this.version++;
    event.setVersion(this.version);
}

public void raiseEvent(final Event event) {
    this.validateEvent(event);

    event.setAggregateType(this.type);
    when(event);

    this.version++;
}

public void clearChanges() {
    this.changes.clear();
}
```

```
    public void toSnapshot() {
        this.clearChanges();
    }

    private void validateEvent(final Event event) {
        if (Objects.isNull(event) || !event.getAggregateId().equals(this.id))
            throw new InvalidEventException(event.toString());
    }

    protected Event createEvent(String eventType, byte[] data) {
        return Event.builder()
                .aggregateId(this.getId())
                .version(this.getVersion())
                .aggregateType(this.getType())
                .eventType(eventType)
                .data(Objects.isNull(data) ? new byte[]{} : data)
                .createdAt(LocalDateTime.now())
                .build();
    }
}
```

TaskAggregate.java:

```
package org.tasker.common.models.domain;

import lombok.*;
import org.tasker.common.es.AggregateRoot;
import org.tasker.common.es.Event;
import org.tasker.common.es.SerializerUtils;
import org.tasker.common.models.enums.TaskPriority;
import org.tasker.common.models.enums.TaskStatus;
import org.tasker.common.models.event.*;
```

```
import java.util.Collections;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

@Getter
@Setter
@ToString
@EqualsAndHashCode(callSuper = false)
@NoArgsConstructor
public class TaskAggregate extends AggregateRoot {

  public static final String AGGREGATE_TYPE = "task_aggregate";

  private String boardId;
  private String title;
  private String description;
  private Date startDate;
  private Date dueDate;
  private int estimatedTime;
  private TaskPriority priority;
  private TaskStatus status;
  private Set<String> assigneeIds;
  private boolean isDeleted;

  public TaskAggregate(String id) {
    super(id, AGGREGATE_TYPE);
  }

  @Override
  public void when(Event event) {
```

```
    switch (event.getEventType()) {
        case TaskCreatedEvent.TASK_CREATED_V1 ->
            handle(SerializerUtils.deserializeFromJsonBytes(event.getData(), TaskCreatedEvent.class));
        case TaskStatusUpdatedEvent.TASK_STATUS_UPDATED_V1 ->
            handle(SerializerUtils.deserializeFromJsonBytes(event.getData(), TaskStatusUpdatedEvent.class));
        case TaskAssigneeAdded.TASK_ASSIGNEE_ADDED_V1 ->
            handle(SerializerUtils.deserializeFromJsonBytes(event.getData(), TaskAssigneeAdded.class));
        case TaskInfoUpdatedEvent.TASK_INFO_UPDATED_V1 ->
            handle(SerializerUtils.deserializeFromJsonBytes(event.getData(), TaskInfoUpdatedEvent.class));
        case TaskAssigneeDeleted.TASK_ASSIGNEE_DELETED_V1 ->
            handle(SerializerUtils.deserializeFromJsonBytes(event.getData(), TaskAssigneeDeleted.class));
        case TaskDeletedEvent.TASK_DELETED_V1 ->
            handle(SerializerUtils.deserializeFromJsonBytes(event.getData(), TaskDeletedEvent.class));
    }
}

private void handle(TaskCreatedEvent taskCreatedEvent) {
    this.boardId = taskCreatedEvent.getBoardId();
    this.title = taskCreatedEvent.getTitle();
    this.description = taskCreatedEvent.getDescription();
    this.startDate = taskCreatedEvent.getStartDate();
    this.dueDate = taskCreatedEvent.getDueDate();
    this.estimatedTime = taskCreatedEvent.getEstimatedTime();
    this.priority = taskCreatedEvent.getPriority();
    this.status = TaskStatus.TODO;
    this.assigneeIds = Collections.synchronizedSet(new HashSet<>());
}

private void handle(TaskStatusUpdatedEvent taskCreatedEvent) {
    this.status = taskCreatedEvent.getStatus();
}
```

```
  private void handle(TaskAssigneeAdded taskAssigneeAdded) {
    if (this.assigneeIds == null) {
      this.assigneeIds = Collections.synchronizedSet(new HashSet<>());
    }
    this.assigneeIds.add(taskAssigneeAdded.getAssigneeId());
  }

  private void handle(TaskInfoUpdatedEvent taskInfoUpdatedEvent) {
    this.title = taskInfoUpdatedEvent.getTitle();
    this.description = taskInfoUpdatedEvent.getDescription();
    this.startDate = taskInfoUpdatedEvent.getStartDate();
    this.dueDate = taskInfoUpdatedEvent.getDueDate();
    this.estimatedTime = taskInfoUpdatedEvent.getEstimatedTime();
    this.priority = taskInfoUpdatedEvent.getPriority();
  }

  public void createTask(String boardId, String title, String description, Date startDate, Date dueDate, int estimatedTime, TaskPriority priority, String createdBy) {
    final var data = TaskCreatedEvent.builder()
        .aggregateId(id)
        .boardId(boardId)
        .title(title)
        .description(description)
        .startDate(startDate)
        .dueDate(dueDate)
        .estimatedTime(estimatedTime)
        .priority(priority)
        .createdBy(createdBy)
        .build();
```

```
    final byte[] dataBytes = SerializerUtils.serializeToJsonBytes(data);
    final var event = this.createEvent(TaskCreatedEvent.TASK_CREATED_V1, dataBytes);
    this.apply(event);
}

public void updateStatus(TaskStatus status) {
    final var data = TaskStatusUpdatedEvent.builder()
            .aggregateId(id)
            .boardId(this.boardId)
            .status(status)
            .build();

    final byte[] dataBytes = SerializerUtils.serializeToJsonBytes(data);
    final var event = this.createEvent(TaskStatusUpdatedEvent.TASK_STATUS_UPDATED_V1, dataBytes);
    this.apply(event);
}

public void addAssignee(String assigneeId) {
    final var data = TaskAssigneeAdded.builder()
            .aggregateId(id)
            .boardId(boardId)
            .assigneeId(assigneeId)
            .build();

    final byte[] dataBytes = SerializerUtils.serializeToJsonBytes(data);
    final var event = this.createEvent(TaskAssigneeAdded.TASK_ASSIGNEE_ADDED_V1, dataBytes);
    this.apply(event);
}

private void handle(TaskAssigneeDeleted taskAssigneeDeleted) {
    this.assigneeIds.remove(taskAssigneeDeleted.getAssigneeId());
```

```
    }

    private void handle(TaskDeletedEvent ignored) {
        this.isDeleted = true;
    }

    public void updateTaskInfo(String title, String description, Date startDate, Date dueDate, int estimatedTime, TaskPriority priority) {
        final var data = TaskInfoUpdatedEvent.builder()
                .aggregateId(id)
                .boardId(boardId)
                .title(title)
                .description(description)
                .startDate(startDate)
                .dueDate(dueDate)
                .estimatedTime(estimatedTime)
                .priority(priority)
                .build();

        final byte[] dataBytes = SerializerUtils.serializeToJsonBytes(data);
        final var event = this.createEvent(TaskInfoUpdatedEvent.TASK_INFO_UPDATED_V1, dataBytes);
        this.apply(event);
    }

    public void deleteAssignee(String assigneeId) {
        final var data = TaskAssigneeDeleted.builder()
                .aggregateId(id)
                .boardId(boardId)
                .assigneeId(assigneeId)
                .build();
```

```
        final byte[] dataBytes = SerializerUtils.serializeToJsonBytes(data);
        final var event = this.createEvent(TaskAssigneeDeleted.TASK_ASSIGNEE_DELETED_V1, dataBytes);
        this.apply(event);
    }

    public void deleteTask() {
        final var data = TaskDeletedEvent.builder()
                .aggregateId(id)
                .boardId(boardId)
                .build();

        final byte[] dataBytes = SerializerUtils.serializeToJsonBytes(data);
        final var event = this.createEvent(TaskDeletedEvent.TASK_DELETED_V1, dataBytes);
        this.apply(event);
    }
}
```

Додаток 7

EventStore.java:

```
package org.tasker.common.es;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.List;

public interface EventStore {

    Mono<Void> saveEvents(final List<Event> events);

    Flux<Event> loadEvents(final String aggregateId, long version);

    <T extends AggregateRoot> Mono<Void> save(final T aggregate);

    <T extends AggregateRoot> Mono<T> load(final String aggregateId, final Class<T> aggregateType);

    Mono<Boolean> exists(final String aggregateId);

    Mono<Boolean> exists(final String aggregateId, final String aggregateType);
}
```

EvestStoreImpl.java:

```
package org.tasker.common.es;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.r2dbc.core.DatabaseClient;
```

```
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.UUID;

@RequiredArgsConstructor
@Slf4j
@Repository
public class EventStoreImpl implements EventStore {

   private final DatabaseClient databaseClient;
   private final EventBus eventBus;

   @Value("${es.snapshot.frequency}")
   private long snapshotFrequency;



   @Override
   @Transactional
   public <T extends AggregateRoot> Mono<Void> save(T aggregate) {
      final List<Event> aggregateEvents = new ArrayList<>(aggregate.getChanges());

      return Mono.fromSupplier(() -> {
               if (aggregate.getVersion() > 1) {
                  return this.handleConcurrency(aggregate.getId());
```

```
            }
            return Mono.empty();
          })
          .then(this.saveEvents(aggregate.getChanges()))
          .then(Mono.defer(() -> {
            if (aggregate.getVersion() % snapshotFrequency == 0) {
              return this.saveSnapshot(aggregate);
            }
            return Mono.empty();
          })).then(eventBus.publish(aggregateEvents));
  }

  @Override
  @Transactional(readOnly = true)
  public <T extends AggregateRoot> Mono<T> load(String aggregateId, Class<T> aggregateType) {
    return loadSnapshot(aggregateId)
        .switchIfEmpty(Mono.just(EventSourcingUtils.snapshotFromAggregate(getAggregate(aggregateId,
aggregateType))))
        .map(snapshot -> EventSourcingUtils.aggregateromSnapshot(snapshot, aggregateType))
        .flatMap(aggregate -> loadEvents(aggregateId, aggregate.getVersion())
            .collectList()
            .doOnNext(events -> events.forEach(aggregate::raiseEvent))
            .thenReturn(aggregate));
  }

  @Override
  public Mono<Void> saveEvents(List<Event> events) {
    return Flux.fromIterable(events)
        .flatMap(event -> {
          event.setId(UUID.randomUUID());
```

```
            return databaseClient.sql("SELECT insert_event(:event_id, :aggregate_id, :event_type, :aggregate_type, :data, :version, :created_at)")
                    .bind("event_id", event.getId())
                    .bind("aggregate_id", event.getAggregateId())
                    .bind("aggregate_type", event.getAggregateType())
                    .bind("event_type", event.getEventType())
                    .bind("data", event.getData())
                    .bind("version", event.getVersion())
                    .bind("created_at", event.getCreatedAt())
                    .fetch()
                    .rowsUpdated();
            })
            .reduce(0L, Long::sum)
            .doOnError(throwable -> log.error("(saveEvents) error saving events", throwable))
            .doOnSuccess(result -> log.debug("(saveEvents) saved events: {}", result))
            .then();
    }

    @Override
    public Flux<Event> loadEvents(String aggregateId, long version) {
        return databaseClient.sql("SELECT event_id ,aggregate_id, aggregate_type, event_type, data, metadata, version, created_at FROM events e WHERE e.aggregate_id = :aggregate_id AND e.version > :version ORDER BY e.version")
            .bind("aggregate_id", aggregateId)
            .bind("version", version)
            .map(row -> Event.builder()
                .id(row.get("event_id", UUID.class))
                .aggregateId(row.get("aggregate_id", String.class))
                .aggregateType(row.get("aggregate_type", String.class))
                .eventType(row.get("event_type", String.class))
                .data(row.get("data", byte[].class))
                .version(row.get("version", Long.class))
```

```
                .createdAt(row.get("created_at", LocalDateTime.class))
                .build())
            .all();
    }

    private <T extends AggregateRoot> Mono<Void> saveSnapshot(T aggregate) {
        aggregate.toSnapshot();
        final var snapshot = EventSourcingUtils.snapshotFromAggregate(aggregate);

        return databaseClient.sql("INSERT INTO snapshots (aggregate_id, aggregate_type, data, version, created_at) VALUES (:aggregate_id, :aggregate_type, :data, :version, now()) ON CONFLICT (aggregate_id) DO UPDATE SET data = :data, version = :version, created_at = now()")
                .bind("aggregate_id", snapshot.getAggregateId())
                .bind("aggregate_type", snapshot.getAggregateType())
                .bind("data", Objects.isNull(snapshot.getData()) ? new byte[]{} : snapshot.getData())
                .bind("version", snapshot.getVersion())
                .fetch()
                .rowsUpdated()
                .doOnError(ex -> log.error("(saveSnapshot) error saving snapshot <{}>", snapshot, ex))
                .doOnSubscribe(result -> log.debug("(saveSnapshot) saved snapshot <{}> with row updated: {}", snapshot, result))
                .then();
    }

    private Mono<Void> handleConcurrency(String aggregateId) {
        return databaseClient.sql("SELECT aggregate_id FROM events e WHERE e.aggregate_id = :aggregate_id LIMIT 1 FOR UPDATE")
                .bind("aggregate_id", aggregateId)
                .map(row -> row.get("aggregate_id", String.class))
                .first()
```

```
            .doOnError(throwable -> log.error("(handleConcurrency) error handling concurrency with aggregate ID <{}>", aggregateId, throwable))
            .doOnSubscribe(result -> log.debug("(handleConcurrency) successfully handled concurrency with aggregate ID <{}>: {}", aggregateId, result))
            .then();
    }

    private Mono<Snapshot> loadSnapshot(String aggregateId) {
        return databaseClient.sql("SELECT aggregate_id, aggregate_type, data, metadata, version, created_at FROM snapshots s WHERE s.aggregate_id = :aggregate_id")
            .bind("aggregate_id", aggregateId)
            .map(row -> Snapshot.builder()
                .aggregateId(row.get("aggregate_id", String.class))
                .aggregateType(row.get("aggregate_type", String.class))
                .data(row.get("data", byte[].class))
                .metaData(row.get("metadata", byte[].class))
                .version(row.get("version", Long.class) == null ? 0 : row.get("version", Long.class))
                .timeStamp(row.get("created_at", LocalDateTime.class))
                .build())
            .first()
            .doOnError(throwable -> log.error("(loadSnapshot) error loading snapshot with aggregate ID <{}>", aggregateId, throwable))
            .doOnSubscribe(result -> log.debug("(loadSnapshot) successfully loaded snapshot with aggregate ID <{}>: {}", aggregateId, result));
    }

    private <T extends AggregateRoot> T getAggregate(final String aggregateId, final Class<T> aggregateType) {
        try {
            return aggregateType.getConstructor(String.class).newInstance(aggregateId);
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
```

```
  }


  @Override
  public Mono<Boolean> exists(String aggregateId) {
    return databaseClient.sql("SELECT count(*) > 0 FROM events WHERE aggregate_id = :aggregate_id")
        .bind("aggregate_id", aggregateId)
        .map(row -> row.get(0, Boolean.class))
        .first()
        .defaultIfEmpty(false)
        .doOnError(throwable -> log.error("(exists) error checking on existence with aggregate ID <{}>", aggregateId, throwable))
        .doOnSubscribe(result -> log.debug("(exists) successfully check on existence with aggregate ID <{}>: {}", aggregateId, result));
  }


  @Override
  public Mono<Boolean> exists(String aggregateId, String aggregateType) {
    return databaseClient.sql("SELECT count(*) > 0 FROM events WHERE aggregate_id = :aggregate_id AND aggregate_type = :aggregate_type")
        .bind("aggregate_id", aggregateId)
        .bind("aggregate_type", aggregateType)
        .map(row -> row.get(0, Boolean.class))
        .first()
        .defaultIfEmpty(false)
        .doOnError(throwable -> log.error("(exists) error checking on existence with aggregate ID <{}>", aggregateId, throwable))
        .doOnSubscribe(result -> log.debug("(exists) successfully check on existence with aggregate ID <{}>: {}", aggregateId, result));


  }
}
```