МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
Факультет кібербезпеки, комп'ютерної та програмної інженерії (ЗФН)
Кафедра комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
_____ Аліна САВЧЕНКО
"___"_____2021 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ

"МАГІСТРА"

ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ

"ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ"

Тема: «Централізована система багато потокової обробки даних на базі технології Thread Pool Executor»

Виконавець: Бут Сергій Миколайович

Керівник: к.т.н., доцент Холявкіна Тетяна Володимирівна

Нормоконтролер: _____ Ігор РАЙЧЕВ

Київ 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії (ЗФН)

Кафедра Комп'ютерних інформаційних технологій

Галузь знань, спеціальність, освітньо-професійна програма: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології"

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Аліна САВЧЕНКО

«____»_____2021р.

ЗАВДАННЯ

на виконання дипломної роботи студента

Бут Сергія Миколайовича

(прізвище, ім'я, по батькові випускника в родинному відмінку)

1. **Тема проекту (роботи):** «Централізована система багато потокової обробки даних на базі технології Thread Pool Executor» затверджена наказом ректора від 12.10.2021р. №2229/ст.

2. **Термін виконання проекту (роботи):** з 11.10.2021 р. по 31.12.2021 р.

3. **Вихідні дані до роботи:** Проаналізувати структури та роботу Thread Pool Executor. Розробити модель об'єктів, що забезпечують роботу та необхідну архітектуру керування багато потоковою обробкою даних Thread Pool Executor. Підготувати проект багато потокової обробки даних за допомогою Thread Pool Executor та деталізувати його структури.

4. **Зміст пояснювальної записки:** Загальна методика та основні методи багатопотокової обробки даних в Java; концепція пулу потоків на прикладу структури Executor; проект багатопотокової обробки даних запитів громадян; компоненти та порівняльні характеристики проекту багатопотокової обробки даних.

5. **Перелік обов'язкового графічного (ілюстративного) матеріалу:** графіки порівняльних характеристик результатів дослідження, діаграми класів проекту, схема структури бази даних.

6. **Календарний план-графік**

№ п/п	Завдання	Термін виконання	Підпис керівника
1	Проаналізувати існуючу нормативну базу. Накопичити матеріали з теми.	11.10.2021– 14.10.2021	
2	Дослідити підходи до засобів багато потокової обробки даних.	15.10.2021– 19.10.2021	
3	Розробити формальну модель проекту багато потокової обробки даних	20.10.2021– 24.10.2021	
4	Виконати проект багато потокової обробки даних.	25.10.2021– 30.10.2021	
5	Визначити структуру проекту та розробити базове алгоритмічне забезпечення.	31.10.2021– 10.11.2021	
6	Підготувати графічні матеріали	12.11.2021– 20.11.2021	
7	Завершити оформлення пояснювальної записки	21.11.2021– 10.12.2021	
8	Підготувати доповідь до захисту дипломної роботи	11.12.2021– 17.12.2021	
9	Підготуватись до захисту дипломного проекту	18.12.2021– 20.12.2021	

7. **Дата видачі завдання:** 11 жовтня 2021 р.

Керівник дипломної роботи _____ Тетяна ХОЛЯВКІНА
(підпис керівника) (ПІБ)

Завдання прийняв до виконання _____ Сергій БУТ
(підпис випускника) (ПІБ)

РЕФЕРАТ

Пояснювальна записка до магістерської роботи “Централізована система багато потокової обробки даних на базі технології Thread Pool Executor” викладена на 126 сторінках і містить 18 рисунків, 5 таблиць, 35 літературних джерел, 1 додаток.

Ключові слова: THREAD POOL EXECUTOR, БАГАТО ПОТОКОВА ОБРОБКА ДАНИХ, ОДНОПОТОКОВА ОБРОБКА ДАНИХ, ПЕРСОНАЛЬНІ ДАНІ, СТАТИСТИКА.

Об'єкт дослідження: процес багато потокової обробки даних в Java.

Предмет дослідження: система багато потокової обробки даних за допомогою структури Thread Pool Executor.

Мета дипломної роботи: розробка проекту застосування технології Thread Pool Executor для багато потокової обробки даних запитів громадян до державних органів та дослідження доцільності впровадження процесу багатопотокової обробки даних до закладів державної влади.

Методи дослідження: аналіз структури Thread Pool Executor, створення моделі керування елементами в структурі Thread Pool Executor, розробка проекту багато потокової обробки запитів громадян, що звертаються до органів державної влади та деталізація його компонентів і алгоритмічного забезпечення.

Отримані результати: розроблено проект багато потокової обробки запитів громадян, що звертаються до органів державної влади, за допомогою Thread Pool Executor та запропоновані практичні рішення з його програмного застосування.

Результати магістерської роботи можуть бути використані: при проектуванні та реалізації засобів багато потокової обробки даних для пришвидшення опрацювання звернень громадян у закладах державної влади.

Прогнозні припущення про розвиток об’єкту та предмету дослідження: дослідження можливості включення до систем опрацювання запитів громадян у всіх без виключення державних органах.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ 8
ВСТУП…….. 9
РОЗДІЛ 1. ЗАГАЛЬНА МЕТОДИКА ТА ОСНОВНІ МЕТОДИ БАГАТОПОТОКОВОЇ ОБРОБКИ ДАНИХ В JAVA 16
1.1. Основні визначення і поняття про потоки 16
1.1.1. Запуск потоків..... 17
1.1.2. Завершення процесу і процеси-демони 22
1.1.3. Завершення потоків..... 22
1.1.4. Переривання..... 25
1.1.5. Метод Thread.sleep ()..... 28
1.1.6. Метод Thread.yield ()..... 28
1.1.7. Метод Thread.join ()..... 29
1.1.8. Пріоритети потоків..... 30
1.2. Основні визначення та поняття про процеси 30
1.2.1. Переваги потоків 322
1.2.2. Використання кількох процесорів 32
1.2.3. Простота моделювання..... 333
1.2.4. Спрощена обробка асинхронних подій..... 344
1.2.5. Більш відповідальні користувацькі інтерфейси..... 355
1.2.6. Ризики потоків..... 366
1.3. Використання пулів потоків 366
ВИСНОВОК ДО РОЗДІЛУ 1…… 388
РОЗДІЛ 2. КОНЦЕПЦІЯ ПУЛУ ПОТОКІВ НА ПРИКЛАДІ СТРУКТУРИ EXECUTOR..... 40
2.1. Основи структури Executor..... 40

2.1.1. Пули потоків 40
2.1.2. Реалізація багатопоточності 422
2.1.3. Чому пул потоків? 433
2.1.4. Альтернативи пулів потоків 444
2.1.5. Налаштування ThreadPoolExecutor 455
2.1.6. Черги завдань 466
2.2. Можливий ризик при використанні пулів потоків 488
2.2.1. Взаємне блокування 488
2.2.2. «Пробуксовка» ресурсів 499
2.2.3. Проблема витоку потоків 50
2.2.4. Перевантаження запитами 50
2.3. Правила ефективного використання пулів потоків 50
2.3.1. Налаштування розміру пулу 511
ВИСНОВОК ДО РОЗДІЛУ 2…… 533
РОЗДІЛ 3. ПРОЕКТ БАГАТОПОТОКОВОЇ ОБРОБКИ ДАНИХ ЗАПИТІВ ГРОМАДЯН 555
3.1. Обґрунтування розробки проекту багатопотокової обробки даних 555
3.2. Модель проекту багато потокової обробки даних 566
3.2.1. Технології, що були використані при створенні проекту 566
3.2.2. Опис моделі 577
3.2.3. Опис схеми формування структури проекту 599
3.2.4. Структура проекту 60
3.2.5. Залежності 60
3.2.6. Типи пакування системи збірки Apache Maven 633
3.2.7. Опис робочого процесу проекту 644
3.2.8. Модель класів 688

ВИСНОВОК ДО РОЗДІЛУ 3……. 711
РОЗДІЛ 4. КОМПОНЕНТИ ТА ПОРІВНЯЛЬНІ ХАРАКТЕРИСТИКИ ПРОЕКТУ БАГАТОПОТОКОВОЇ ОБРОБКИ ДАНИХ 733
4.1. Опис основних компонентів проекту 733
4.2. Опис важливих компонентів («ядра» проекту) 777
4.2.1. WizardFlowComponent: опис методів 788
4.2.2. WizardFormController: опис методів 799
4.3. Порівняння результатів засобів обробки даних 799
4.3.1 Результати виміру швидкості обробки 100 об’єктів 811
4.3.2. Результати виміру швидкості обробки 1000 об’єктів 822
4.3.3. Результати виміру швидкості обробки 3000 об’єктів 844
ВИСНОВОК ДО РОЗДІЛУ 4……. 866
ВИСНОВКИ 888
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 933
ДОДАТОК А 966

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

БД – база даних

БП – багато поточність (багато поточний)

ОП – одно поточність (одно поточний)

ОС – операційна система

ТРЕ – Thread Pool Executor

ВСТУП

Написання правильних програм – важка справа; правильне написання БП програм – ще важче. Існує багато простих речей, які можуть піти не так в БП програмі, ніж у ОП.

Потоки - особливість мови Java, що може спростити розробку складних систем, повернувши складнощі асинхронного коду в більш простіший прямолінійний код. Крім того, потоки є найпростішим способом застосування обчислювальної потужності багатопроцесорних системах. І при збільшенні кількості процесорів, питання використання БП стає більш важливим.

Раніше комп'ютери не мали ОС, вони виконували одну програму від початку до кінця. І ця програма отримувала прямий доступ до всіх ресурсів машини. Так як в один момент працювала лише одна програма, це було неефективне використання дорогих і дефіцитних ресурсів комп'ютера.

ОС еволюціонувала для підтримки запуску декількох програм одночасно. Запуск окремих програм в процесах є ізольованим, незалежно від виконання програм, для яких операційна система розподіляє такі ресурси, як пам'ять, файли або облікові дані безпеки. Якщо потрібно, процеси можуть взаємодіяти один з одним через різні комунікаційні механізми. Деякі чинники привели до розробки операційних систем, що дозволило програмам працювати одночасно.

Всі сучасні операційні системи, такі як Windows або UNIX, здатні працювати в багатопотоковому режимі, підвищуючи загальну продуктивність системи за рахунок ефективного розпаралелювання виконуваних потоків. Поки один потік знаходиться в стані очікування, наприклад, завершення операції обміну даними з повільним периферійним пристроєм, інший може продовжувати виконувати свою роботу. [1]

Користувачі вже давно звикли запускати паралельно кілька додатків, для того щоб робити кілька справ відразу. Поки один з них займається, наприклад, друком документа на принтері або прийомом електронної пошти з мережі Internet, інший - може перераховувати електронну таблицю або виконувати іншу корисну роботу. при цьому самі по собі додатки, що запускаються, можуть пра-

цювати в рамках одного потоку - ОС сама піклується про розподіл часу між усіма запущеними додатками.

Створюючи програми для операційної системи Windows на мовах програмування С або С + +, можливо вирішувати багато завдань, такі як анімація або робота в мережі, і без використання БП. Наприклад, для анімації можна було обробляти повідомлення відповідним чином налаштованого таймера.

Для додатків Java така методика недоступна, так як в цьому середовищі не передбачено способів періодичного виклику будь-яких процедур. Тому для вирішення багатьох завдань просто не обійтися без БП.

Основна складність, з якою стикаються програмісти,котрі ніколи не створювали раніше БП програми, це синхронізація одночасно працюючих потоків.

ОП програма, при запуску отримує в монопольне розпорядження всі ресурси комп'ютера. Так як в ОП системі існує тільки один процес, він використовує ці ресурси в тій послідовності, яка відповідає логіці роботи програми.

Процеси і потоки, що працюють одночасно в багато потокової системі, можуть намагатися звертатися одночасно до одних і тих же ресурсів, що може призвести до неправильної роботи додатків.

Найпростіший приклад з банківським рахунком - критичні операції зняття суми з рахунку. Припустимо, що створена програма, котра виконує операції з банківським рахунком. Операція зняття деякої суми грошей з рахунку може відбуватися в такій послідовності:

- на першому кроці перевіряється загальна сума грошей, яка зберігається на рахунку;
- якщо загальна сума дорівнює або перевищує розмір суми грошей, що знімається, загальна сума зменшується на необхідну величину;
- значення залишку записується на поточний рахунок.

Якщо операція зменшення поточного рахунку виконується в ОП системі, то ніяких проблем не виникне. Однак уявімо собі, що два процеси намагаються одночасно виконати тільки що описану операцію з одним і тим же рахунком.

Нехай при цьому на рахунку знаходиться 5 млн. доларів, а обидва процеси намагаються зняти з нього по 3 млн. доларів.

Припустимо, події розгортаються таким чином:

•перший процес перевіряє стан поточного рахунку і переконується, що на ньому зберігається 5 млн. доларів;

•другий процес перевіряє стан поточного рахунку і також переконується, що на ньому зберігається 5 млн.доларів;

•перший процес зменшує рахунок на 3 млн. доларів і записує залишок (2 млн. доларів) на поточний рахунок;

•другий процес виконує ту ж саму операцію, так як після перевірки вважає, що на рахунку раніше зберігається 5 млн. доларів.

У результаті вийшло, що з рахунку, на якому знаходилося 5 млн. доларів, було знято 6 млн.доларів, і при цьому там залишилося ще 2 млн. доларів! Разом - банку завдано збитків в 3 млн. доларів. [2]

Дуже просто - на час виконання операцій над рахунком одним процесом необхідно заборонити доступ до цього рахунку з боку інших процесів. У цьому випадку сценарій роботи програми повинен бути наступним:

•процес блокує рахунок для виконання операцій іншими процесами, отримуючи його в монопольне володіння;

•процес проводить процедуру зменшення рахунку і записує на поточний рахунок нове значення залишку;

•процес розблоковує рахунок, дозволяючи іншим процесам виконання операцій.

Коли перший процес блокує рахунок, він стає недоступний іншим процесам. Якщо другий процес також спробує заблокувати цей же рахунок, він буде переведений у стан очікування.

Коли перший процес зменшить рахунок і на ньому залишиться 2 млн. доларів, другий процес буде розблокований. Він перевірить залишок, переконається, що сума недостатня і не проводитиме операцію.

Таким чином, в БП середовищі необхідна синхронізація потоків при зверненні до критичних ресурсів. Якщо над такими ресурсами виконуватимуться операції в неправильній послідовності, це призведе до помилок, котрі дуже важко виявити.

Використання ресурсів. Програмам іноді потрібно чекати допоки зовнішні операції, такі як вхід або вихід, завершаться, так як в цей час очікування програми не можуть зробити жодної корисної роботи. Логічно більш ефективно використати цей час очікування для запуску іншої програми.

Справедливість. Кілька користувачів і програм можуть мати однакові потреби в ресурсах комп'ютера. Найкращий вибір – розподілити ресурси комп'ютера між програмами, виділити певний час для кожної із програм на використання ресурсів. Даний варіант краще, ніж дозволити цикл одного запуску і завершення програми, а потім розпочати цикл для іншої програми.

Зручність. Часто легше або більш бажано написати кілька програм, кожна з яких виконує одну задачу і взаємодіють одна з одною в міру необхідності, ніж написати одну програму, яка виконує всі завдання. [3]

Послідовна модель програмування є інтуїтивно зрозумілою та закономірною, оскільки вона моделює, як люди працюють: виконують одну річ за один раз, в основному послідовно. Встати з ліжка, одягнути халат, спуститися вниз і випити чай. Як і в мові програмування, кожна з цих реальних дій - це абстракція, що складається з послідовності більш детальних дій: відкрити шафу, вибрати смак чаю, перевірити, чи в чайнику достатня кількість води, якщо ні, то налити ще трохи води, поставити його на плиту, увімкнути плиту, почекати, коли закипить вода, і так далі.

Цей останній крок очікування, коли вода закипить також припускає певний ступінь асинхронності. У той час як вода нагрівається, у вас є вибір, що робити: просто чекати, або виконувати щось інше у цей час, таких як запуск тостера або витягнути газету з поштового ящика, знаючи, що незабаром потрібно повернутись до чайника.

Виробники чайників та тостерів знають. що їх продукція часто використовується в асинхронному режимі, тому вони видають звуковий сигнал, коли завершують своє завдання. Знайти правильний баланс послідовності та асинхронності часто є характеристикою ефективних людей, а також ефективних програм.

Ті ж самі принципи (використання ресурсів, справедливість і зручність), що мотивували розвиток процесів також стосується і розвиток потоків. Процеси дозволяють програмі управляти в кілька потоків, щоб вони могли співіснувати в межах одного процесу. Потоки поділяють відкриті для користування ресурси процесу, такі як пам'ять і дескриптори файлів (детальна інформація про кожний з файлів), але кожен потік має свій власний програмний лічильник, стек і локальні змінні. Потоки також забезпечують природне розподілення для використання апаратного паралелізму в багатопроцесорних системах; кілька потоків у рамках однієї програми можуть бути заплановані одночасно на декількох процесорах.

Потоки іноді називають полегшеними процесами, і більшість сучасних операційних систем використовують потоки, а не процеси, в якості основних одиниць планування. У відсутності явної координації, потоки виконуються одночасно і асинхронно по відношенню один до одного. Оскільки потоки спільно використовують адресний простір пам'яті свого процесу, всі потоки у процесі мають доступ до спільних змінних і розміщення об'єктів у загальній памяті процесу, яка дозволяє виконувати кращий обмін даними, ніж між механізмами процесу. Але без явної синхронізації доступу до загальних даних, один потік може змінювати дані, які інший потік використовує, що може призвести до непередбачуваних результатів.

Актуальність теми. Згідно вищеописаних причин і розбіжностей робимо висновок, що, не дивлячись на складний процес реалізації, БП обробка даних значно переважає над ОП обробкою даних. Так як в органах державної влади ще досі для обробки даних громадян використовуються людський ресурс, а використання автоматичної обробки даних ще досі не введено, тема дипломної роботи є актуальною. Впровадження БП обробки даних в органах державної влади підвищить швидкість опрацювання запитів громадян та дозволить зменшити людський

ресурс, що затрачається на цей процес, а також час очікування громадян на офіційну відповідь державного закладу щодо запиту.

Метою дипломної роботи є розробка проекту застосування технології Thread Pool Executor для багато потокової обробки даних запитів громадян до державних органів та дослідження доцільності впровадження процесу багато потокової обробки даних до закладів державної влади.

Об'єктом даного дослідження є процес багато потокової обробки даних в Java. Виділяючи з даного об’єкту предмет, окреслимо предмет даного дослідження як система багато потокової обробки даних за допомогою структури Thread Pool Executor.

Методами дослідження дипломної роботи є аналіз структури Thread Pool Executor, створення моделі керування елементами в структурі Thread Pool Executor, розробка проекту багато потокової обробки запитів громадян, що звертаються до органів державної влади та деталізація його компонентів і алгоритмічного забезпечення.

Наукова новизна отриманих результатів. Вперше розроблено прототип централізованої системи багато потокової обробки даних для декількох органів державної влади, що дозволяє з його використанням мінімізувати кількість можливих систем обробки даних та пришвидшити опрацювання запитів, централізовано зберігати дані громадян, результати їхніх запитів до установ різного роду. Вперше розроблена система статистики для системи багато потокової обробки даних з автоматичним моніторингом оброблених запитів згідно фіксованого періоду часу та можливістю очищення статистики шляхом приховування з неї всіх оброблених запитів.

TPE – це засіб БП обробки даних в Java, введений з релізу Java 1.5 та призначений для виконання кожної задачі, що міститься в черзі TPE, в кожному новому потоці, зменшуючи при цьому обсяг ресурсів системи, що використовуються, та ймовірність взаємного блокування потоків. TPE працює безупинно, постійно перевіряючи чергу на надходження нових об’єктів для обробки. [4]

Розробка системи виконана на основі результатів аналізу особливостей

структури ТРЕ. Висновки. сформульовані в процесі аналізу, покладені в основу створення об’єктної моделі проекту. Модель дозволяє окреслити всі основні елементи процесу БП обробки даних запитів громадян та концепцію створення об’єктів, додання їх в чергу ТРЕ для обробки та відображення на сторінці статистики після закінчення виконання всіх дій, зв’язаних з даним об’єктом.

Подальша розробка ґрунтується на створенні проекту БП обробки даних запитів громадян, що звернулися до органів державної влади. Даний проект деталізує основи процесу БП обробки даних, окреслює перелік основних дій, що дозволяють запустити ТРЕ, а також дозволяє адміністратору системи автоматично збирати дані щодо кількості, часового періоду та успішності обробки запитів громадян. Шляхом використання статистики по запитам громадян буде досліджено час обробки тестових запитів із заздалегідь фіксованою кількістю: 100, 1000 та 3000 запитів відповідно.

Отримана система БП обробки даних орієнтована на заклади державної влади (податкові інспекції, державні, обласні, міські адміністрації і т. д.), що дозволяє пришвидшити та автоматизувати процес обробки запитів громадян з будь-яких причин, а по результаті – повідомляти офіційну відповідь запитувачу.

РОЗДІЛ 1

ЗАГАЛЬНА МЕТОДИКА ТА ОСНОВНІ МЕТОДИ БАГАТОПОТОКОВОЇ ОБРОБКИ ДАНИХ В JAVA

1.1. Основні визначення і поняття про потоки

Один потік - це одна одиниця виконання коду. Кожен потік послідовно виконує інструкції процесу, якому він належить, паралельно з іншими потоками цього процесу.

Слід окремо обговорити фразу «паралельно з іншими потоками». Відомо, що на одне ядро процесора, в кожен момент часу, припадає одна одиниця виконання. Тобто одноядерний процесор може обробляти команди тільки послідовно, по одній за раз (у спрощеному випадку). Однак запуск декількох паралельних потоків можливий і в системах з одноядерними процесорами. У цьому випадку система буде періодично перемикатися між потоками, по черзі даючи виконуватися то одному, то іншому потоку. Така схема називається псевдо-паралелізмом. Система запам'ятовує стан (контекст) кожного потоку, перед тим як переключитися на інший потік, і відновлює його після повернення до виконання потоку. У контекст потоку входять такі параметри, як стек, набір значень регістрів процесора, адреса виконуваної команди і т.д.

Простіше кажучи, при псевдопаралельному виконанні потоків процесор метається між виконанням декількох потоків, виконуючи по черзі частини кожного з них. Ілюстрація даного процесу зображена на рисунку нижче.

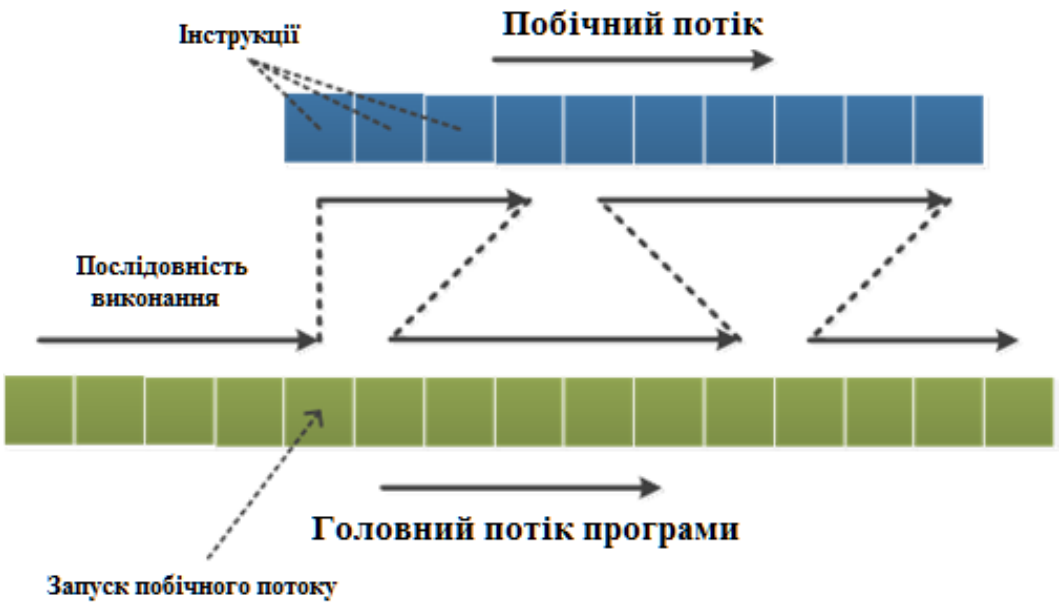

Рис. 1.1. Запуск побічного потоку

Квадрати на малюнку - це інструкції процесора (нижні - інструкції головного потоку, верхні - побічного). Виконання йде зліва направо. Після запуску побічного потоку його інструкції починають виконуватися упереміш з інструкціями головного потоку. Кількість виконуваних інструкцій за кожен підхід не визначено.

Те, що інструкції паралельних потоків виконуються упереміш, в деяких випадках може призвести до конфліктів доступу до даних.

1.1.1. Запуск потоків

Кожен процес має хоча б один потік, що виконується. Той потік, з якого починається виконання програми, називається головним. У мові Java, після створення процесу, виконання головного потоку починається з методу main (). Потім, у міру необхідності, в заданих програмістом місцях, і при виконанні заданих ним же умов, запускаються інші, побічні потоки.

У мові Java потік представляється у вигляді об'єкта - нащадка класу Thread. Цей клас містить в собі стандартні механізми роботи з потоком. Запустити новий потік можна двома способами. [5]

Спосіб 1

Створити об'єкт класу Thread, передавши йому в конструкторі щось, що реалізує інтерфейс Runnable. Цей інтерфейс містить метод run (), який буде виконуватися в новому потоці. Потік закінчить виконання, коли завершиться його метод run (). Приклад реалізації потоку за допомогою передачі в конструктор об’єкта Runnable зображений в лістингу нижче.

```
class SomeThing                  //Дещо, що реалізує інтерфейс Runnable
implements Runnable {
   public void run () {          //Цей метод виконуэться в побічному потоці
         System.out.println ("Виклик побічного потоку");
   }
}

public class Program {                  //Класс с методом main ()
   static SomeThing mThing;  //об’єкт класу, що реалізує інтерфейс Runnable
      public static void main (String[] args) {
      mThing = new SomeThing ();
      Thread myThready = new Thread (mThing);        //Створення потоку
      myThready.start ();                                    //Запуск потоку
      System.out.println ("Головний потік завершився..");
      }
}
```

Спосіб 2

Створити нащадка класу Thread і перевизначити його метод run (). Приклад створення потоку шляхом перевизначення методу run() зображений в лістингу нижче.

```
class AffableThread extends Thread {
      @ Override
```

```
        public void run () { / / Цей метод буде виконаний в побічному потоці
            System.out.println ("Привіт з побічного потоку!");
        }
    }
    public class Program {
        static AffableThread mSecondThread;

        public static void main (String [] args) {
            mSecondThread = new AffableThread (); / / Створення потоку
            mSecondThread.start (); / / Запуск потоку
            System.out.println ("Головний потік завершено..");
        }
    }
```

У наведеному вище прикладі у методі main () створюється і запускається ще один потік. Важливо відзначити, що після виклику методу mSecondThread.start () головний потік продовжує своє виконання, не чекаючи поки породжений ним потік завершиться. І ті інструкції, які йдуть після виклику методу start (), будуть виконані паралельно з інструкціями потоку mSecondThread.

Для демонстрації паралельної роботи потоків в лістингу нижче зображено програму, в якій два потоки сперечаються на предмет філософського питання «що було раніше, яйце чи курка? ». Головний потік впевнений, що першою була курка, про що він і буде повідомляти кожну секунду. Другий потік раз на секунду буде спростовувати свого опонента. Всього суперечка триватиме 5 секунд. Переможе той потік, який останнім прорече свою відповідь на це філософське питання. У прикладі використовуються методи, про які поки не було сказано (isAlive (), sleep () і join ()). До них дані коментарі, а більш детально вони будуть розібрані далі.

```
    class EggVoice extends Thread{
        @ Override
```

```
        public void run (){
                for  (int i = 0; i < 5; i + +){
                        try {
                        Thread.sleep (1000); / / Припиняє потік на 1 секунду
                        } catch  (InterruptedException e) { }
                        System.out.println  (" яйце !");
                        }/ / Слово « яйце » сказано 5 разів
        }
  }
  public class ChickenVoice {
        static EggVoice mAnotherOpinion; / / Побічний потік
        public static void main (String [ ] args) {
        mAnotherOpinion = new EggVoice (); / / Створення потоку
        System.out.println  (" Суперечка розпочато..");
        mAnotherOpinion.start (); / / Запуск потоку
                for  (int i = 0; i < 5; i + +) {
                        try {
                        Thread.sleep (1000); / / Припиняє потік на 1 секунду
                        } catch  (InterruptedException e) { }
                        System.out.println  (" курка !");
                        }/ / Слово « курка » сказано 5 разів
        if  (mAnotherOpinion.isAlive ()) / / Якщо опонент ще не сказав останнє
слово
                {
            try {
          mAnotherOpinion.join (); / / Почекати поки опонент закінчить висловлюватися.
            } catch  (InterruptedException e) { }
            System.out.println  (" Першим з'явилося яйце !");
        }else / / якщо опонент вже закінчив висловлюватися
```

```
        {
            System.out.println  (" Першою з'явилася курка !");
        }
        System.out.println  (" Суперечка закінчено! ");
    }
}
```

```
консоль :
Суперечка розпочато..
курка !
яйце !
яйце !
курка !
яйце !
курка !
яйце !
курка !
яйце !
курка !
Першою з'явилася курка !
Суперечка закінчено!
```

У наведеному прикладі два потоки паралельно на протязі 5 секунд виводять інформацію на консоль. Точно передбачити, який потік закінчить висловлюватися останнім, неможливо. Можна спробувати, і можна навіть вгадати, але є велика ймовірність того, що та ж програма при наступному запуску буде мати іншого « переможця ». Це відбувається через так званого « асинхронного виконання коду».

Асинхронність означає те, що не можна стверджувати, що будь-яка інструкція одного потоку, виконається раніше чи пізніше інструкції іншого. Або, іншими словами, паралельні потоки незалежні один від одного, за винятком тих випадків, коли програміст сам описує залежності між потоками за допомогою передбачених для цього засобів мови.

1.1.2. Завершення процесу і процеси-демони

У Java процес завершується тоді, коли завершується останній його потік. Навіть якщо метод main () вже завершився, але ще виконуються породжені ним потоки, система буде чекати їх завершення.

Однак це правило не стосується до особливого виду потоків - демонам. Якщо завершився останній звичайний потік процесу, і залишилися тільки потоки - демони, то вони будуть примусово завершені і виконання процесу закінчиться. Найчастіше потоки - демони використовуються для виконання фонових завдань, обслуговуючих процес протягом його життя.

Оголосити потік демоном досить просто - потрібно перед запуском потоку викликати його метод setDaemon (true);

Перевірити, чи є потік демоном, можна викликавши його метод boolean isDaemon(). [6]

1.1.3. Завершення потоків

У Java існують (існували) засоби для примусового завершення потоку. Зокрема метод Thread.stop () завершує потік негайно після свого виконання. Однак цей метод, а також Thread.suspend (), що припиняє потік, і Thread.resume (), що продовжує виконання потоку, були оголошені застарілими і їх використання відтепер вкрай небажано. Справа в тому що потік може бути « убитий » під час виконання операції, обрив якої на півслові залишить деякий об'єкт в неправильному стані, що призведе до появи ситуації, котру тяжко буде відловити, і випадковим чином виникає помилка.

Замість примусового завершення потоку застосовується схема, в якій кожен потік сам відповідальний за своє завершення. Потік може зупинитися або тоді, коли він закінчить виконання методу run (), (main () - для головного потоку) або за сигналом з іншого потоку. Причому як реагувати на такий сигнал - справа, знову ж, самого потоку.

Отримавши його, потік може виконати деякі операції і завершити виконання, а може і зовсім його проігнорувати і продовжити виконуватися. Опис реакції на сигнал завершення потоку лежить на плечах програміста.

Java має вбудований механізм оповіщення потоку, який називається Interruption (переривання, втручання). Спочатку розглянемо лістинг, зображений нижче.

```
class Incremenator extends Thread {
   private volatile boolean mIsIncrement = true;
   private volatile boolean mFinish = false;

   public void changeAction () {/ / Змінює дію на протилежне
      mIsIncrement = ! mIsIncrement;
   }
   public void finish (){/ / Ініціює завершення потоку
      mFinish = true;
   }
   @ Override
   public void run () {
      do {
         if (! mFinish)  {/ / Перевірка на необхідність завершення
            if  (mIsIncrement)
               Program.mValue + +; / / Інкремент
            else
               Program.mValue - -; / / Декремент

            / / Висновок поточного значення змінної
            System.out.print  (Program.mValue + "");
         } else
            return; / / Завершення потоку
         try {
```

```
            Thread.sleep (1000); / / Призупинення потоку на 1 с
          } catch  (InterruptedException e) { }
        }
        while  (true);
    }
}

public class Program { / / Змінна, яка оперує інкременатор
    public static int mValue = 0;
    static Incremenator mInc; / / Об'єкт побічного потоку

    public static void main (String [ ] args)    {
        mInc = new Incremenator (); / / Створення потоку
        System.out.print  (" Значення =");
             mInc.start (); / / Запуск потоку

        for  (int i = 1; i < = 3; i + +)        {
          try {
             Thread.sleep  (i * 2 * 1000); / / Очікування в перебігу i * 2 с
          } catch  (InterruptedException e) { }

          mInc.changeAction (); / / Переключення дії
        }
             mInc.finish (); / / Ініціація завершення побічного потоку
    }
}
```

консоль :

Значення = 1 2 1 0 -1 -2 -1 0 1 2 3 4

Incremenator - потік, який щосекунди додає чи віднімає одиницю із значення статичної змінної Program.mValue. Incremenator містить два закритих поля - mIsIncrement і mFinish.

Те, яка дія виконується, визначається булевої змінної mIsIncrement - якщо воно дорівнює true, то виконується додаток одиниці, інакше - віднімання. А завершення потоку відбувається, коли значення mFinish стає одно true. [7]

1.1.4. Переривання

Клас Thread містить у собі приховане булево поле, подібне полю mFinish в програмі Incremenator, яке називається прапором переривання. Встановити цей прапор можна викликавши метод interrupt () потоку. Перевірити ж, чи встановлений цей прапор, можна двома способами. Перший спосіб - викликати метод bool isInterrupted () об'єкта потоку, другий - викликати статичний метод bool Thread.interrupted (). Перший метод повертає стан прапора переривання і залишає цей прапор недоторканим. Другий метод повертає стан прапора і скидає його. Зауважте що Thread.interrupted () - статичний метод класу Thread, і його виклик повертає значення прапора переривання того потоку, з якого він був викликаний. Тому цей метод викликається тільки зсередини потоку і дозволяє потоку перевірити свій стан переривання. [8]

Отже, повернемося до нашої програми. Механізм переривання дозволить нам вирішити проблему з засипанням потоку. У методів, котрі зупиняють виконання потоку, таких як sleep (), wait () і join () є одна особливість - якщо під час їх виконання буде викликаний метод interrupt () цього потоку, вони, не чекаючи кінця часу очікування, згенерують виняток InterruptedException.

Змінимо програму Incremenator - тепер замість завершення потоку за допомогою методу finish () будемо використовувати стандартний метод interrupt (). А замість перевірки прапора mFinish будемо викликати метод bool Thread.interrupted ();

Так виглядатиме клас Incremenator після додавання підтримки переривань :

```
class Incremenator extends Thread
{
   private volatile boolean mIsIncrement = true;

   public void changeAction () / / Змінює дію на протилежне
   {
      mIsIncrement = ! mIsIncrement;
   }

   @ Override
   public void run () {
      do
      {
         if (! Thread.interrupted ()) / / Перевірка переривання
         {
            if  (mIsIncrement) Program.mValue + +; / / Інкремент
            else Program.mValue - -; / / Декремент

            / / Висновок поточного значення змінної
            System.out.print  (Program.mValue + "");
         }
         else
            return; / / Завершення потоку
         try {
            Thread.sleep (1000); / / Призупинення потоку на 1 с
         } catch  (InterruptedException e) {
            return; / / Завершення потоку після переривання
         }
      }
```

```
        while  (true);
    }
}

class Program { / / Змінна, яка оперує інкременатор
    public static int mValue = 0;
    static Incremenator mInc; / / Об'єкт побічного потоку
    public static void main (String [ ] args)
    {
        mInc = new Incremenator (); / / Створення потоку

        System.out.print  (" Значення ="");
        mInc.start (); / / Запуск потоку
        for  (int i = 1; i < = 3; i + +)
        {
            try {
                Thread.sleep  (i * 2 * 1000); / / Очікування в перебігу i * 2 c
            } catch  (InterruptedException e) { }

            mInc.changeAction (); / / Переключення дії
        }

        mInc.interrupt (); / / Переривання побічного потоку
    }
}
```

консоль :

Значення = 1 2 1 0 -1 -2 -1 0 1 2 3 4

Ми позбулися методу finish () і реалізували той же механізм завершення потоку за допомогою вбудованої системи переривань. У цій реалізації ми отримали одну перевагу - метод sleep () поверне управління (згенерує виняток) негайно після переривання потоку. [9]

Методи sleep () і join () обгорнуті в конструкції try-catch. Це необхідна умова роботи цих методів. Код, що викликається, повинен перехоплювати виключення InterruptedException, яке вони кидають при перериванні під час очікування.

1.1.5. Метод Thread.sleep ()

Thread.sleep () - статичний метод класу Thread, який призупиняє виконання потоку, в якому він був викликаний. Під час виконання методу sleep () система перестає виділяти потоку процесорний час, розподіляючи його між іншими потоками. Метод sleep () може виконуватися або задана кількість часу (мілісекунди або наносекунди), або до тих пір поки він не буде зупинений перериванням (в цьому випадку він згенерує виключення InterruptedException).

Thread.sleep (1500); / / Чекає півтори секунди

Thread.sleep (2000, 100); / / Чекає 2 секунди і 100 наносекунд

Незважаючи на те, що метод sleep () може приймати як час очікування наносекунди, не варто приймати це всерйоз. У багатьох системах час очікування все одно округлюється до мілісекунд, а то і до їх десятків. [10]

1.1.6. Метод Thread.yield ()

Статичний метод Thread.yield () змушує процесор переключитися на обробку інших потоків системи. Метод може бути корисним, наприклад, коли потік очікує настання якої-небудь події і необхідно щоб перевірка його настання відбувалася якомога частіше. [11] У цьому випадку можна помістити перевірку події і метод Thread.yield () в цикл :

```
/ / Очікування надходження повідомлення
while (! msgQueue.hasMessages ()) / / Поки в черзі немає повідомлень
{
   Thread.yield (); / / Передати управління іншим потокам
}
```

1.1.7. Метод Thread.join ()

У Java передбачений механізм, що дозволяє одному потоку чекати завершення виконання іншого. Для цього використовується метод join (). Наприклад, щоб головний потік почекав завершення побічного потоку myThready, необхідно виконати інструкцію myThready.join () в головному потоці. Як тільки потік myThready завершиться, метод join () поверне управління, і головний потік зможе продовжити виконання.

Метод join () має перевантажену версію, яка отримує як параметр час очікування. У цьому випадку join () повертає управління або коли завершиться очікуваний потік, або коли закінчиться час очікування. Подібно до методу Thread.sleep () метод join може чекати протягом мілісекунд і наносекунд - аргументи ті ж. [12]

За допомогою завдання часу очікування потоку можна, наприклад, виконувати оновлення анімованої картинки поки головний (або будь-який інший) потік чекає завершення побічного потоку, що виконує ресурсомісткі операції:

```
Thinker brain = new Thinker (); / / Thinker - нащадок класу Thread.
brain.start (); / / Нове " обмірковування ".
do
{
   mThinkIndicator.refresh (); / / mThinkIndicator - анімована картинка.
   try {
      brain.join  (250); / / Почекати закінчення думки чверть секунди.
   } catch  (InterruptedException e) { }
```

```
}
while  (brain.isAlive ()); / / Поки brain думає..
/ / brain закінчив думати .
```

У цьому прикладі потік brain (мозок) думає над чимось, і передбачається, що це займає у нього тривалий час. Головний потік чекає його чверть секунди і, у разі, якщо цього часу на роздуми не вистачило, оновлює « індикатор роздумів» (деяка анімована картинка). У підсумку, під час роздумів, користувач спостерігає на екрані індикатор розумового процесу, що дає йому знати, що електронні мізки ніж те зайняті.

1.1.8. Пріоритети потоків

Кожен потік в системі має свій пріоритет. Пріоритет - це деяке число в об'єкті потоку, більш високе значення якого означає більший пріоритет. Система в першу чергу виконує потоки з великим пріоритетом, а потоки з меншим пріоритетом отримують процесорний час тільки тоді, коли їх більш привілейовані побратими простоюють. [13]

Працювати з пріоритетами потоку можна за допомогою двох функцій :

- void setPriority (int priority) - встановлює пріоритет потоку. Можливі значення priority - MIN_PRIORITY, NORM_PRIORITY і MAX_PRIORITY.
- int getPriority () - повертає пріоритет потоку.

1.2. Основні визначення та поняття про процеси

Процес - це сукупність коду і даних, які поділяють спільний віртуальний адресний простір. Найчастіше одна програма складається з одного процесу, але бувають і винятки (наприклад, браузер Google Chrome створює окремий процес для кожної вкладки, що дає йому деякі переваги, начебто незалежності вкладок один від одного). Процеси ізольовані один від одного, тому прямий доступ до пам'яті чужого процесу неможливий.

Для кожного процесу ОС створює так званий «віртуальний адресний простір», до якого процес має прямий доступ. Цей простір належить процесу, містить тільки його дані і знаходиться в повному його розпорядженні. ОС ж відповідає за те, як віртуальний простір процесу проектується на фізичну пам'ять.

Схема цієї взаємодії представлена на Рис.1.2. ОС оперує так званими сторінками пам'яті, які представляють собою просто область певного фіксованого розміру. Якщо процесу стає недостатньо пам'яті, система виділяє йому додаткові сторінки з фізичної пам'яті. Сторінки віртуальної пам'яті можуть проектуватися на фізичну пам'ять в довільному порядку.

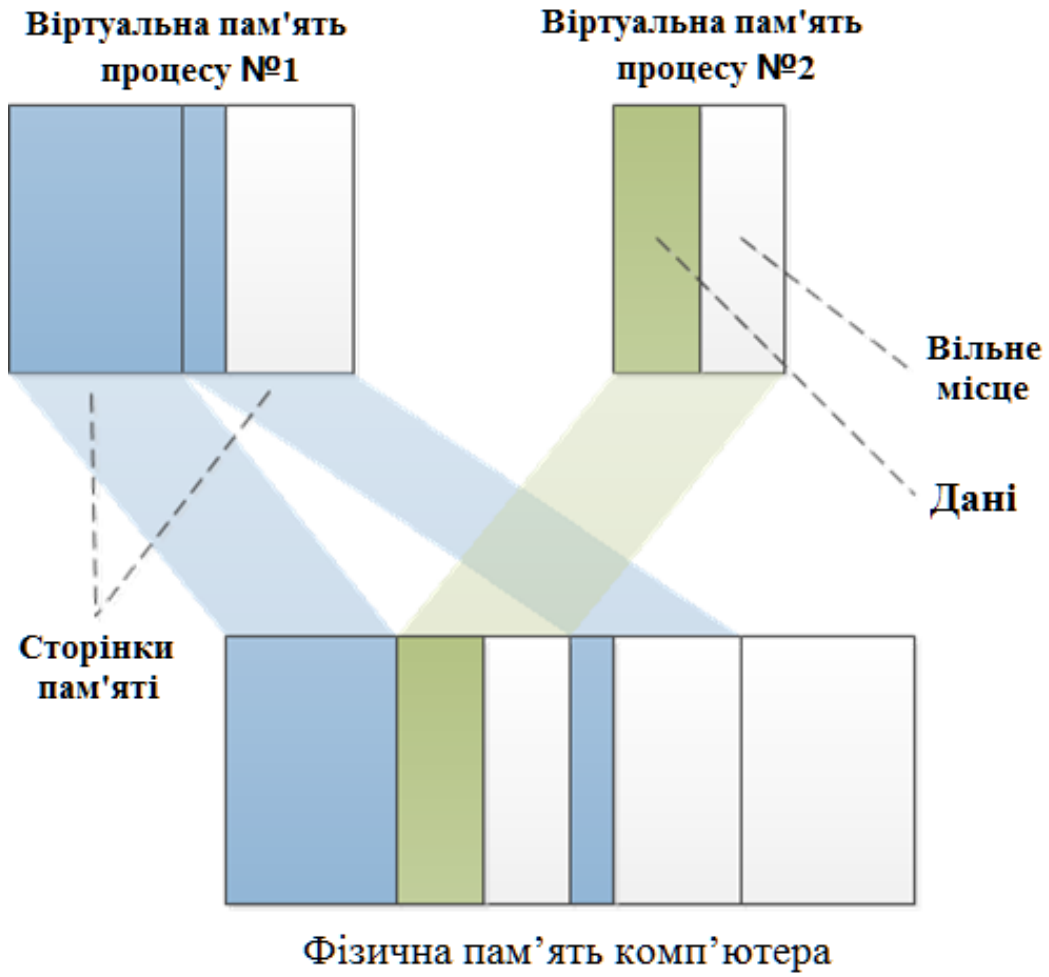

Рис. 1.2. Фізична пам’ять комп’ютера

1.2.1. Переваги потоків

При правильному використанні, потоки можуть зменшити витрати на розробку та обслуговування, підвищити продуктивність додатку. Потоки корисні для додатків з графічним інтерфейсом для поліпшення інтерфейсу користувача і, в серверних додатках, для поліпшення використання ресурсів і пропускної здатності. Вони також спростили реалізацію JVM – сміття.

Колектор зазвичай працює в одному або більше виділених потоків. Найкращі нетривіальні Java програми повинні покладатися в деякій мірі на потоки в своїй організації.

1.2.2. Використання кількох процесорів

Багатопроцесорні системи раніше були дорогими і рідкісними, зустрічались тільки у великих центрах обробки даних і високопродуктивних обчислень об'єктів. Сьогодні вони дешеві і поширені, навіть серверні системи з низькими характеристиками часто мають кілька процесорів.

Оскільки основною одиницею планування є потік, програма, що має лише один потік, може працювати на одному процесорі в поточний момент.

З іншого боку, програми з безліччю активних потоків можуть виконуватися одночасно на декількох процесорах. При правильній розробці БП програм можливо поліпшити пропускну здатність, використовуючи наявні ресурси процесора більш ефективно.

З використанням декількох потоків також можна допомогти в досягненні більш високої пропускної здатності на однопроцесорних системах. Якщо програма є ОП, процесор простоює в той час як він чекає закінчення синхронних операцій введення/виведення. У багато потоковому режимі, інший потік може як і раніше працювати, в той час як перший потік очікує закінчення введення/виведення, дозволяючи додатку виконувати обробку даних протягом блокування введення/виведення.

1.2.3. Простота моделювання

Часто легше управляти своїм часом, коли у вас є тільки один тип для виконання операцій, ніж коли у вас є кілька. Якщо у вас є тільки один тип завдання для виконання, вам не доведеться витрачати розумову енергію, щоб з'ясувати, що ж буде працювати далі. З іншого боку, управління кількома пріоритетами, перехід від завдання до завдання зазвичай несе певні витрати. Те ж саме справедливо для програм : програму, яка обробляє послідовно завдання одного типу простіше писати, роблячи менше помилок, простіше тестувати. Призначення потоку для кожного типу завдання або для кожного елементу в моделюванні дає ілюзію послідовності та ізолює доменну логіку від деталей планування, чергуванням операцій асинхронного введення/виводу, а також очікування ресурсів.

Складний, асинхронний робочий процес може бути розкладений на ряд більш простих, синхронні процеси, кожний з яких працює в окремому потоці, взаємодіють один з одним тільки в окремих точках синхронізації. Цю перевагу часто використовують структури, такі як сервлети або RMI (віддалений виклик методів). Вони виконують необхідні дії для управління запитами, створення потоків і розподілу навантаження, диспетчеризація частини запиту при зверненні до відповідного компоненту програми у відповідний момент в потоці. Засновникам Servlet технології було потрібно турбуватися про те, як багато інших запити обробляються в один і той же час, а коли метод сервлета service () викликається у відповідь на веб-запит, він може обробляти синхронні запити, як ОП програма. Це може спростити розробку компонентів і зменшити час навчання для використання таких структур.

1.2.4. Спрощена обробка асинхронних подій

Сервер додатків, який приймає підключення по сокетам з декількох віддалених клієнтів можливо буде простіше розвивати, коли кожному з'єднанню відводиться свій потік і дозволяється використовувати синхронне введення/виведення.

Якщо додаток розгортається для читання з сокету, коли ніякі дані не доступні, читати сенсу немає, так як деякі блоки даних відсутні. У ОП додатку, це означає, що не тільки обробка відповідного запиту зупиняється, але й зупиняється обробка всіх запитів, в той час як один потік блокується. Щоб уникнути цієї проблеми, одно поточні додатки сервера змушені використовувати не блокуючі засоби введення/виведення, які є набагато складнішими і схильними до помилок, ніж синхронні засоби введення/виведення. Однак, якщо кожен запит має свій власний потік, то блокування не впливає на обробку інших запитів.

Історично склалося так, що операційні системи мають відносно низькі ліміти на кількість потоків, які може створити один процес. В результаті, в операційних системах були розроблені ефективні засоби для складних засобів введення/виведення, наприклад, система Unix опитує систему для доступу до цих об'єктів за допомогою бібліотек Java: пакети (java.nio) для НЕ блокуючого введення/виведення. Таким чином, підтримка операційною системою більшого числа потоків має значно покращитися, роблячи потоко - клієнтську модель практичною навіть для великого числа клієнтів на деяких платформах. [15]

1.2.5. Більш відповідальні користувацькі інтерфейси

GUI додатки (Graphic User Interface – графічний користувацький інтерфейс), що використовуються як ОП, що означає, що вони повинні або часто опитуватись по всьому коду для подій введення (що дещо брудно і нав'язливо з точки зору програмування) або виконувати весь код програми побічно, через точку управління всіма доступними джерелами подій. Якщо код викликається з цієї точки, це виконання займає надто багато часу, користувацький інтерфейс не можливо «заморозити » до моменту завершення виконання попереднього коду, тому що наступні події користувача інтерфейсу не можуть розглядатися, як ті, що будуть чекати. Сучасні засоби GUI, заміняють точку управління всіма доступними джерелами подій на потік відправки події.

Коли ця подія, така як натискання кнопки, для користувача інтерфейсу відбувається, то застосовується визначений обробник події, що викликається в потоці подій. Більшість GUI засобів є ОП підсистемами, так що точка управління доступними джерелами фактично ще присутня, але вона працює у своєму власному потоці під контролем GUI інструментарія, а не безпосередньо в додатку. [16]

Якщо короткострокові завдання виконуються в потоці подій, інтерфейс залишається працездатним, тому що потік подій завжди здатний обробляти дії користувача досить швидко. Однак обробка довгострокової задачі в потоці подій, таких, як виклик перевірки великого документа або завантаження файлу по мережі, погіршує працездатність. Якщо користувач виконує дії, в той час як це завдання виконується, то виникає велика затримка у потоці подій і не тільки користувацький інтерфейс перестає відповідати на запити, але й неможливо відмінити роботу довгострокового завдання, що призвело до затримки.

Навіть якщо користувач на інтерфейсі натискає кнопку скасування, так як потік подій зайнятий і не може впоратися з кнопкою скасування, допоки довгострокова задача завершується. Якщо ж дану довгострокову задачу виконати в окремому потоці, що потік подій залишиться вільним для обробки подій користувацького інтерфейсу, що робить для користувача інтерфейс більш чуйним.

1.2.6. Ризики потоків

Java підтримує потоки і хоча це спрощує одночасну розробку додатків за допомогою синтаксису мови і бібліотек та формальної кросс - платформенної моделі пам'яті (саме ця формальна кросс - платформенна модель пам'яті, що дає можливість написати програму один раз, а працювати буде скрізь), воно також піднімає планку для розробників, оскільки більше програм буде використовувати потоки.

Якщо б потоки були більш таємничі, то паралелізм був би більш «просунутий» темою,однак зараз, основні розробники повинні бути в курсі питань безпеки при роботі з потоками. [17]

1.3. Використання пулів потоків

Вище згадувалося, що потоки після виконання своїх функцій автоматично завершуються віртуальною машиною. При цьому у виділених їм областях пам'яті залишається непотрібна інформація. Що вона собою являє ? Це службова інформація, яку процесору доводиться обробляти при кожному створенні і завершенні потоку. Тому для оптимізації БП найчастіше виділяється група (яку називають пулом потоків), потоки з якої використовуються багаторазово. У такому випадку функція, відповідна потоку, записана в пулі і чекає завдання для виконання. Після виконання отриманої завдання потік повертається в пул.

При розробці БП програм та створенні пулів потоків виникає ще одне питання: кількість потоків. Воно залежить від того, як ви плануєте використовувати свої потоки. Якщо потоки призначені для виконання декількох окремих завдань, створіть стільки потоків, скільки завдань необхідно обробити.

Наприклад, у текстовому процесорі один потік повинен використовуватися для виводу на екран (практично в усіх програмах головний потік призначений для оновлення інтерфейсу користувача), інший - для розбиття документа на сторінки, третій - для перевірки правопису, четвертий - для якихось фонових функцій. Тобто в даному випадку оптимальна кількість потоків - чотири; це робить розробку програмного забезпечення значно простіше. [18]

ВИСНОВОК ДО РОЗДІЛУ 1

Незважаючи на перший погляд простоту створення і підтримку потоків, насправді це не є ідеальне рішення для систем з великим навантаженням та вимогами високої продуктивності і швидкості обробки, виконання, оновлення інформації. Перша причина – це складність підтримки великої кількості потоків, які фізично представляють собою різні об’єкти в пам’яті віртуальної машини Java, нічим між собою не зв’язаних. Розробник логічно пов’язує всі потоки разом, так як вони обробляють ті ж самі елементи бізнес-логіки, але насправді це не так згідно концепції віртуальної машини Java. Друга причина – взаємні блокування потоків. Це найбільш поширена проблема великої кількості потоків, не об’єднаних в пул. Так як трапляються випадки, коли потік А очікує на доступність ресурсів сервера, котрі вже використовуються потоком Б, а потік Б. в свою чергу, очікує ресурси, зайняті потоком А. В результаті отримуємо блокування, що може продовжуватись вічно.

Кожна програма Java використовує потоки. Більшість програмістів використовує Java- інструментарії для користувача інтерфейсу (AWT або Swing), сервлети Java, RMI, сторінки JavaServer або технологію Enterprise JavaBeans, що є самі по собі багато потоковими.

Існує безліч ситуацій, коли ви можете побажати використовувати потоки в явній формі для підвищення продуктивності, чуйності або організованості своїх програм. Ці ситуації включають :

- підвищення чуйності інтерфейсу користувача під час виконання тривалих завдань;
- застосування багатопроцесорних систем для паралельного управління декількома завданнями;
- спрощення моделювання або систем на основі агентів
- виконання асинхронної або фонової обробки

І хоча програмний інтерфейс потоків досить простий, про написання поточно-орієнтованих програм цього не скажеш. При спільному використанні

змінних декількома потоками потрібно ретельно стежити за тим, щоб процеси читання і запису цих змінних були відповідним чином синхронізовані. При виконанні запису змінної, яку може використовувати інший потік, або при виконанні читання змінної, яка може бути записана іншим потоком, ви повинні використовувати синхронізацію, щоб зміни даних були видні всім потокам.

При використанні синхронізації для захисту спільних змінних потрібно не тільки застосовувати синхронізацію, але і стежити за тим, щоб потік, що читає, і потік, що пише, синхронізувалися від одного і того ж монітора. Крім того, якщо покладатися на те, що стан об'єкта залишиться незмінним протягом кількох операцій або розраховуєте, що кілька змінних залишаться узгодженими між собою (або зі своїми попередніми значеннями), потрібно використовувати синхронізацію, щоб це гарантовано сталося. Але проста синхронізація всіх методів класу не робить його поточно-орієнтованим - вона лише робить його більш схильним до виникнення взаємного блокування.

Для полегшення розробки та підтримки кінцевого продукту, досягнення кращої продуктивності необхідно використовувати пул потоків. Яскравим представником пулу потоків в Java є TPE. Об’єднання потоків в пулі під одним контролюючим механізмом значно зменшує ризики взаємного блокування. Пул потоків має функціонал вивільнення потоків і повернення їх до пулу, якщо потоки закінчили обробку своїх завдань.

РОЗДІЛ 2

КОНЦЕПЦІЯ ПУЛУ ПОТОКІВ НА ПРИКЛАДІ СТРУКТУРИ EXECUTOR

2.1. Основи структури Executor

Набір executor - інструментів вводить інтерфейс для управління виконання завдань.

Інтерфейс Executor використовується для виконання та затвердження завдань, представлених у вигляді Runnable об’єктів. Цей інтерфейс також ізолює затвердження завдання від його виконання. Кожне окреме завдання має власний життєвий цикл, що складається з двох фаз:

1. Затвердження – під час даної фази розробником підтверджується дійсна необхідність виконання поточного завдання в структурі типу Executor.
2. Виконання – безпосередньо обробка поточного завдання в структурі типу Executor.

Для затвердження об’єкта типу Runnable для виконання в структурі типу Executor. Необхідно написати код за схожим шаблоном:

```
Executor exec = …;
exec.execute (runnable);
```

2.1.1. Пули потоків

Важливо зазначити, що з початку роботи в структурі Executor не вказано як само потрібно обробляти (виконувати) завдання, що потраплять в структуру. Це залежить від конкретного типу виконавця, що використовується. Інфраструктура забезпечує декілька різних типів виконавців, кожен з яких має певну політику з ураху

ванням використання для різних випадків.

Найбільш поширений тип виконавців – це виконавці пулу потоків, яскравими представниками якого є екземпляри класу ThreadPoolExecutor (і його підкласи). ThreadPoolExecutor керує пулом робочих потоків, котрі виконують завдання, і робочою чергою. Робоча черга – структура даних типу черга (FIFO – First In First Out – Перший Зайшов Перший Вийшов).

Основна перевага використання пулу - це зменшення накладних витрат на створення ресурсів, повторне використання структур (в даному випадку, потоків), які звільняють ресурси, виділенні для їхньої роботи, після використання. Інша неявна перевага пулу - можливість зміни розміру використання ресурсів: ви можете налаштувати розмір пулу потоків для досягнення бажаного навантаження, не ставлячи під загрозу системні ресурси. [19]

Інфраструктура забезпечує клас-фабрику для пулів потоків, що носить назву Executors. Використання цієї фабрики дає можливість створювати пули потоків різних характеристик. Часто в основі такої реалізації лежить ThreadPoolExecutor, але клас-фабрика допомагає швидко налаштувати пул потоків без використання його складних конструкторів.

Клас-фабрика містить в собі такі методи:

- newFixedThreadPool : цей метод повертає пул потоків, максимальний розмір фіксований. Він може створювати нові потоки по мірі необхідності, але обмежений заданою максимальною місткістю.
- newCachedThreadPool : цей метод повертає необмежений пул потоків, тобто пул потоків без заданого максимального розміру. Пул потоків може прибрати потік, що на даний момент не використовується, щоб зменшити навантаження.
- newSingleThreadedExecutor : цей метод повертає Executor, що містить лише один потік.
- newScheduledThreadPool: цей метод повертає пул потоків фіксованого розміру, який підтримує наперед задану затримку в часі для виконання завдань.

Виконавці також надають інші об'єкти:

- Методи управління життєвим циклом, оголошені в інтерфейсі ExecutorService (наприклад, shutdown () і awaitTermination ()).
- Завершення послуг для опитування стану завдання і знаходити його значення, що повертається, якщо це доречно.

ExecutorService інтерфейс особливо важливий, оскільки він надає можливість для завершення роботи пулу потоків, після виклику котрого потоки закінчують виконувати завдання і ресурси, надані потокам для використання, звільняються.

Надсилається сигнал завершення за допомогою методу ExecutorService shutdown (), після чого пул потоків не прийматиме нових завдань, але буде продовжувати обробку завдань, що містяться в черзі. За необхідності можливо перевіряти статус виконавця (завершено роботу чи ні) за допомогою методу isTerminated (), або дочекатися припинення використання awaitTermination () методом.

2.1.2. Реалізація багатопоточності

БП, яку організовують для підвищення продуктивності, значно ускладнює програмний код, особливо якщо необхідно передбачити взаємодію потоків. Важливо розібратися з премудростями організації потоків, так як чим більше ядер в процесорах, тим більше потоків доводиться передбачати в програмному коді. Недостатня компетенція розробника при організації багатопоточності безсумнівно призведе до виникнення безлічі помилок у програмі, тому давайте перейдемо до розгляду рішень деяких можливих складнощів.

Очікування завершення іншого потоку : Припустимо, нам потрібно обробити цілочисельний масив. Це можна робити послідовно, обробляючи одне значення масиву за іншим, або, що більш продуктивно, паралельно - створенням декількох потоків, кожному з яких призначена певна область масиву. Припустимо, щоб перейти до наступної стадії обробки, нам необхідно дочекатися завершення всіх потоків. Для синхронізації взаємодії потоків використовується метод join (). При його задіюванні запуск певного потоку відбудеться після завершення другого. Таким чином, потік У чекатиме завершення потоку А. При перевищенні часу очікування, призначеного в

методі join (), потік У займеться обробкою інших даних, якщо потік А ще не завершений. Отже, ми обговорили найскладнішу задачу при організації багатопоточності - проблему очікування завершення потоків. Повернемося до неї пізніше.

Очікування доступу до заблокованих об'єктів : Припустимо, ми займаємося розробкою системи бронювання квитків для аеропорту. Найчастіше багатопоточність в таких програмах організована таким чином, що кожному підключеному до системи користувачеві, а також усім касирам призначені окремі потоки (організація дуже великих систем може відрізнятися від описаної). Якщо два користувача одночасно спробують забронювати одне і те ж місце, а в програмі ця можливість не передбачена, відбудеться наступне: один потік звернеться до конкретного ресурсу одночасно з іншим потоком. В результаті обидва користувача вирішать, що вони забронювали собі потрібний квиток на літак.

Для того щоб запобігти зміні одних і тих же даних різними потоками, необхідно, щоб один потік заблокував доступ до них для іншого потоку. Тоді при спробі другого потоку змінити дані, йому доведеться чекати, поки перший потік не зніме свою блокування. У нашому прикладі другий потік виявить, що дане місце вже було зарезервовано, і видасть відповідне попередження користувачеві. Ситуація, описана вище, називається " станом гонки " (race condition), і її наслідки можуть бути дуже згубними. Ось чому на час виконання операцій з даними одним потоком прийнято блокувати доступ до цих даних для інших потоків.

2.1.3. Чому пул потоків?

Робота багатьох серверних додатків, таких як Web -сервери, сервери бази даних, файлові сервери або поштові сервери, пов'язана із виконанням великої кількості коротких завдань, що надходять від будь-якого віддаленого джерела. Запит прибуває на сервер певним чином, наприклад, через мережеві протоколи (такі як HTTP, FTP або POP), через чергу JMS, або, можливо, шляхом опитування бази даних. Незалежно від того, як запит надходить, в серверних додатках часто буває, що обробка кожної індивідуальної задачі короткочасна, а кількість запитів велике.

Однією з спрощених моделей для побудови серверних додатків є створення нового потоку щоразу, коли запит прибуває і обслуговування запиту в цьому новому потоці. Цей підхід у дійсності хороший для розробки прототипу, але має значні недоліки, що стало б очевидним, якби знадобилося розгорнути серверний додаток, що працює таким чином. Один з недоліків підходу "потік-на-запит" полягає в тому, що системні витрати створення нового потоку для кожного запиту значні; а сервер, який створив новий потік для кожного запиту, буде витрачати більше часу і споживати більше системних ресурсів, створюючи і руйнуючи потоки, ніж на витрати, обробляючи фактичні запити користувача. [20]

Додатково до витрат створення й руйнування потоків, активні потоки споживають системні ресурси. Створення занадто великої кількості потоків в одній JVM (віртуальної Java- машині) може призвести до нестачі системної пам'яті або пробуксовці через надмірне споживання пам'яті. Для запобігання пробуксовки ресурсів, серверним додаткам потрібні деякі заходи з обмеження кількості запитів, що обробляються в заданий час.

Потік пулів пропонує рішення і проблеми витрат життєвого циклу потоку, і проблеми пробуксовки ресурсів. При багаторазовому використанні потоків для вирішення численних завдань, витрати створення потоку поширюються на багато завдань. Як бонус, оскільки потік вже існує, коли прибуває запит, затримка, що відбулася через створення потоку, усувається. Таким чином, запит може бути оброблений негайно, що робить додаток більш швидким за наданням відповіді. Більше того, правильно налаштувавши кількість потоків в пулі потоків, можна запобігти пробуксовку ресурсів, змусивши будь-які запити, якщо їх кількість виходить за певні межі, чекати доти, поки потік не стане доступним, щоб його обробити.

2.1.4. Альтернативи пулів потоків

Пули потоку - це далеко не єдиний спосіб використовувати множинні потоки в серверному додатку. Як згадувалося раніше, іноді досить розумно генерувати новий потік для кожної нової задачі. Однак, якщо частота створення завдань висока, а їх

середня тривалість низька, породження нового потоку для кожного завдання призведе до проблем з продуктивністю.

Інша поширена модель організації потокової обробки - наявність єдиного фонового потоку і черги завдань для задач певного типу. AWT (набір інструментальних засобів для абстрактних вікон) і Swing використовують цю модель, в якій є потік подій GUI (графічного інтерфейсу користувача), і вся робота, що викликає зміни в інтерфейсі, повинна виконуватися в цьому потоці. Однак, оскільки існує тільки один AWT - потік, небажано виконувати завдання в потоці AWT, завершення якого може зайняти значну кількість часу. В результаті, додатки Swing часто вимагають додаткових потоків для вирішення довгострокових, пов'язаних з призначеним для користувача інтерфейсом (UI) завдань.

Підходи " потік-на-задачу" і "єдиний фоновий потік" можуть досить добре функціонувати в певних ситуаціях. Підхід "потік -на-задачу" добре працює з невеликою кількістю довгострокових завдань. Підхід "єдиний фоновий потік " функціонує досить добре, якщо не важлива передбачуваність розподілу (scheduling predictability), як у випадку низько пріоритетних фонових (low - priority background) завдань. [21] Однак більша частина серверних додатків орієнтовані на обробку великої кількості короткострокових завдань або під задач, і потрібно мати механізм для ефективного здійснення цих завдань з невеликими витратами, а також будь-яку міру управління ресурсами і передбачуваністю часу виконання.

2.1.5. Налаштування ThreadPoolExecutor

Для створення ThreadPoolExecutor вручну, а не за допомогою класу-фабрики Executors, потрібно використати один з його конструкторів. Найбільш великі конструкторі цього класу :

public ThreadPoolExecutor (int corePoolSize, int maxPoolSize, long keepAlive, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler);

Таким чином можливо налаштувати:

- Основний розмір пулу потоків (кількість потоків, котру пул потоків буде старатися дотримуватися)
- Максимальний розмір пулу потоків (максимальна кількість потоків, котру пул потоків не в змозі перевищити)
- Час «простою» потоку, після закінчення якого потік може бути викинутий з пулу.
- Робоча черга для зберігання завдань, що очікують на виконання
- Політика, котрої дотримуються потоки у випадку, коли завдання не може бути виконано.

2.1.6. Черги завдань

Звичайно, ми могли легко застосовувати клас пулу потоків, в якому клас клієнтів очікував би доступного потоку, передавав би завдання цьому потоку для виконання і потім повертав би потік до пулу, коли всі закінчено; але цей підхід має кілька потенційно небажаних ефектів. Що, наприклад, якщо пул пустий ? Будь-яка сторона, яка зробила спробу передати завдання пулу потоків, виявила б, що пул пустий, і її потік заблокувався б, очікуючи доступного пулу потоків.

Те, що нам зазвичай потрібно - це робоча черга в поєднанні з фіксованою групою робочих потоків, в якій використовуються методи wait () та notify (), щоб сигналізувати потокам, котрі очікують, про те, що прибула нова робота. Черга завдань головним чином застосовується як зв'язаний список з приєднаним об'єктом монітора. Лістинг нижче показує приклад простої, об'єднаної в пул черги. [22]

```
public class WorkQueue {
   private final int nThreads;
   private final PoolWorker [ ] threads;
   private final LinkedList queue;

   public WorkQueue  (int nThreads) {
```

```
    this.nThreads = nThreads;
    queue = new LinkedList ();
    threads = new PoolWorker [ nThreads ];

    for  (int i = 0; i < nThreads; i + +) {
        threads [ i ] = new PoolWorker ();
        threads [ i ]. start ();
    }
}

public void execute  (Runnable r) {
    synchronized  (queue) {
        queue.addLast  (r);
        queue.notify ();
    }
}

private class PoolWorker extends Thread {
    public void run () {
        Runnable r;

        while  (true) {
            synchronized  (queue) {
                while  (queue.isEmpty  ()) {
                    try
                    {
                        queue.wait ();
                    }
                    catch  (InterruptedException ignored) {}
                }
                r =  (Runnable) queue.removeFirst ();
            }
```

```
            try {
               r.run ();
            }
            catch  (RuntimeException e) {
            }
         }
      }
   }
}
```

2.2. Можливий ризик при використанні пулів потоків

Хоча пул потоків - потужний механізм для структурування БП додатків, він пов'язаний з певним ризиком. Додатки, побудовані за допомогою пулів потоків, схильні до всіх паралельних ризиків, що й будь-який інший БП додаток, як, наприклад, помилки синхронізації і взаємне блокування, і також декільком іншим ризикам, специфічних також для пулів потоків, таких, як залежне від пулів взаємне блокування, пробуксовка ресурсів.

2.2.1. Взаємне блокування

У будь-якому БП додатку є ризик взаємного блокування. Кажуть, що набір процесів або потоків перебувають у взаємному блокуванні, коли кожен очікує події, яка може бути викликана іншим процесом. Найпростіший випадок такого блокування - коли потік A повністю блокує об'єкт X і очікує блокування об'єкта Y, в той час як потік B повністю блокує об'єкт Y і очікує блокування об'єкта X. І якщо немає будь-якого способу вирватися з очікування блокування, заблоковані потоки будуть очікувати вічно. [23]

Оскільки взаємне блокування - ризик у будь-якій БП програмі, пули потоків припускають іншу можливість взаємного блокування, де всі потоки пулів здійсню-

ють завдання, які блокуються в очікуванні результатів іншої задачі в черзі, але це завдання не може запуститися, оскільки немає доступного незайнятого потоку. Це може статися, коли пули потоків використовуються для проведення імітаційних експериментів, що включають велику кількість взаємодіючих об'єктів, імітаційні об'єкти можуть посилати запити один одному, які потім виконуються як завдання з черги, і запитуваний об'єкт синхронно очікує відповіді.

2.2.2. «Пробуксовка» ресурсів

Одна з переваг пулів потоків полягає в тому, що вони зазвичай добре виконують операції, що мають відношення до альтернативних розподіляє механізмам, деякі з яких ми вже обговорили. Але це вірно тільки в тому випадку, якщо розмір пулу потоків налаштований правильно. Потоки споживають численні ресурси, включаючи пам'ять і інші системні ресурси. Крім пам'яті, що вимагається для об'єкта Thread, кожен потік вимагає двох списків викликів виконання, які можуть бути великими. На додаток до цього, JVM, можливо, створить " рідний" потік для кожного Java- потоку, що пов'язано зі споживанням додаткових системних ресурсів. Нарешті, оскільки розподіляються витрати перемикання між потоками малі, багатьом потоків перемикання процесів може стати значним уповільненням в роботі програми. [24]

Якщо пул потоків занадто великий, ресурси, що споживаються цими потоками, можуть значною мірою вплинути на роботу системи. Час буде марно витрачено на перемикання між потоками, і якщо потоків більше, ніж необхідно, це може викликати проблеми нестачі ресурсів, так як пули потоків споживають ресурси, які могли б бути більш ефективно використані іншими завданнями. На додаток до ресурсів, що використовуються самими потоками, робота, виконувана з обслуговування запитів, може також вимагати додаткових ресурсів, таких як JDBC, сокети, або файли. Ці ресурси також обмежені, і занадто багато паралельних запитів можуть викликати збої, такі як неможливість отримати JDBC - з'єднання до бази даних.

2.2.3. Проблема витоку потоків

Істотний ризик в самих різних пулах потоків полягає у витоку потоку, який трапляється, коли потік виділяється з пулу для виконання завдання, але не повертається в пул, коли завдання виконане. По-перше, це відбувається, коли завдання породжує помилку, що не дає змоги потоку продовжувати роботу. Якщо потік не зможе проігнорувати помилку, тоді потік просто припиняється і розмір пулу скорочується на один. Якщо дана ситуація відбудеться кількість разів, рівну кількості потоків, що знаходяться в пулі, то система заблокується, тому, що немає потоків, доступних для здійснення завдань.

Завдання, які постійно блокуються, наприклад, ті, що потенційно чекають на доступність ресурсів, можуть також викликати ефект, еквівалентний витоку потоку. Якщо потік постійно займається таким завданням, він скоріш всього не буде доступний в пулі. Таким завданням слід або виділяти власний потік, або обмежити час очікування.

2.2.4. Перевантаження запитами

У випадках. коли сервер переповнений запитами, не раціонально розміщувати кожний вхідний запит в робочу чергу, так як завдання, які очікують виконання, можуть споживати дуже багато системних ресурсів і викликати нестачу ресурсів. В деяких ситуаціях можна проігнорувати запит, сподіваючись, що запит буде надіслано повторно пізніше, або, повідомивши, що сервер тимчасово зайнятий. [25]

2.3. Правила ефективного використання пулів потоків

Не потрібно ставити в чергу завдання, які одночасно очікують результатів інших завдань. Це може викликати взаємоблокування описаної вище форми, де всі потоки зайняті завданнями, що очікують результатів від завдань в черзі, не виконуються, оскільки всі потоки зайняті.

Якщо програма повинна чекати ресурс, необхідно вказувати максимальний час очікування, а потім повертати в чергу завдання для виконання в більш пізній час. Це гарантує, що деякий прогрес буде досягнутий шляхом звільнення потоку для задач, які могли б успішно здійснитися.

Якщо існує в системі різні типи (класи) завдань, що радикально відрізняються одне від одного своїми характеристиками, має сенс мати кілька робочих черг для різних типів завдань, так, щоб кожен пул можна було налаштувати окремо і керувати ним без впливу на інші пули потоків.

2.3.1. Налаштування розміру пулу

Налаштовуючи розмір пулу потоків, важливо уникнути двох помилок : занадто мало потоків або занадто багато потоків. На щастя, для більшості додатків спектр між дуже великим і дуже малою кількістю потоків досить широкий.

Існує дві основні переваги в організації потокової обробки повідомлень в додатках: можливість продовження процесу під час очікування повільних операцій, таких, як операції вводу/виводу, і використання можливостей декількох процесорів. У додатках з обмеженням по швидкості обчислень, що функціонують на N-процесорної машині, додавання додаткових потоків може поліпшити пропускну здатність, у міру того як кількість потоків підходить до N, але додавання додаткових потоків понад N не виправдане. Дійсно, занадто багато потоків руйнують якість функціонування через додаткові витрати перемикання процесів

Оптимальний розмір пулу потоків залежить від кількості доступних процесорів і природи завдань в робочій черги. На N-процесорної системі для робочої черги, яка буде виконувати виключно завдання з обмеженням по швидкості обчислень, можливо досягнути максимального використання CPU з пулом потоків, в якому міститься N або N +1 потік. [26]

Для завдань, які можуть чекати здійснення операції вводу/виводу- наприклад, завдання, що зчитує HTTP-запит з сокету - вам може знадобитися збільшення розміру пулу більше кількості доступних процесорів, тому, що не всі потоки будуть пра-

цювати весь час.

А ThreadPoolExecutor автоматично скоректує розмір пулу згідно з межами, встановленими в атрибуті corePoolSize і в атрибуті maximumPoolSize. Коли нова задача представляється у методі execute (java.lang.Runnable), і менше ніж corePoolSize потоків працюють, новий потік створюється, щоб обробити запит, навіть якщо інші робочі потоки неактивні. Якщо потоків буде більше ніж corePoolSize, але менше, ніж maximumPoolSize залученні у виконанні завдань, то новий потік буде створюватися, тільки якщо черга повна. Встановлюючи corePoolSize і maximumPoolSize те ж саме, ви створюєте пул потоків фіксованого розміру. Встановлюючи maximumPoolSize до надзвичайно необмеженому значенням такий як Integer.MAX_VALUE, ви дозволяєте пулу розміщувати довільне число паралельних завдань. Зазвичай базові та максимальні розміри пулу встановлюються тільки на конструкцію, але вони можуть також бути змінені, динамічно використовуючи setCorePoolSize (int) і setMaximumPoolSize (int).

ВИСНОВОК ДО РОЗДІЛУ 2

Пул потоків - корисний інструмент для організації серверів та додатків. Він досить простий по суті, але є деякі моменти, з якими слід бути обережними під час застосування та використання, такі як взаємне блокування, пробуксовка ресурсів, і складнощі, пов'язані з wait () і notify (). Якщо вам буде потрібно пул потоків для вашої програми, розгляньте використання одного з класів Executor з util.concurrent, такий як ThreadPoolExecutor, замість створення нового. Якщо потрібно створити потоки для вирішення короткострокових завдань, безумовно слід розглянути використання замість цього пул потоків.

Більшість реалізацій структури Executor в java.util.concurrent використовують пули потоків, які складаються з робочих потоків. Цей вид потоку існує окремо від Runnable і Callable завдань і часто використовується, щоб виконати багаторазові завдання.

Пул потоків, використовуючи робочі потоки мінімізує витрати, належно розпаралелювати виконання завдань. Об'єкти потоків використовують істотну кількість пам'яті, і у великомасштабному додатку, створюють істотні витрати управління пам'яттю.

Один загальний тип пулу потоків є фіксованим пулом потоків. У цього типу пулу завжди є конкретна кількість виконання потоків; якщо потік так чи інакше завершується, в той час як пул потоків знаходиться все ще у використанні, то автоматично замінюється новим потоком. Завдання для пулу зберігаються у внутрішній черці.

Важлива перевага фіксованого пулу потоків полягає в тому, що додатки, використовуючи його погіршуються коректно. Щоб зрозуміти це, розгляньте заяву веб - сервера, де кожен запит HTTP обробляється окремим потоком. Якщо додаток просто створить новий потік для кожного нового запиту HTTP, і система отримує більше запитів, ніж може відразу обробити, то додаток раптово припинить відповідати на всі запити, коли витрати всіх тих потоків перевищать ємність системи. З межею на числі потоків, які можуть бути створені, програма не буде обслуговувати запити

HTTP так швидко, як вони входять, але буде обслуговувати їх так швидко, як система може витримати.

Хоча пул потоків - потужний механізм для структурування БП додатків, він пов'язаний з певним ризиком. Додатки, побудовані за допомогою пулів потоків, схильні до всіх паралельних ризиків, що й будь-який інший БП додаток, як, наприклад, помилки синхронізації і взаємне блокування, і також декільком іншим ризикам, специфічних також для пулів потоків, таких, як залежне від пулів взаємне блокування, пробуксовка ресурсів.

РОЗДІЛ 3

ПРОЕКТ БАГАТОПОТОКОВОЇ ОБРОБКИ ДАНИХ ЗАПИТІВ ГРОМАДЯН

3.1. Обґрунтування розробки проекту багатопотокової обробки даних

Модель обробки даних заяв громадян, що звертаються до органів державної влади з метою отримання інформації, а саме: подання, зміна, оформлення, відкликання персональних документів, котра існує в державних закладах в поточний момент є необхідною умовою, але не достатньою для швидкої обробки інформації громадян.

Існує дві можливості обробки даних:

- одно потокова
- багато потокова

Звісно, і це очевидно, що засоби багато потокової обробки інформації за виділений проміжок часу зможуть обробити набагато більшу кількість даних (заяв), ніж засоби одно потокової обробки.

Суть способу одно потокової обробки інформації, в свою чергу, полягає у використанні одного потоку класу Thread, котрий зможе в конкретний момент часу обробляти лише одну заяву, що буде викликати питань щодо доцільності даного способу.

Суть способу багато потокової обробки інформації полягає у використанні ТРЕ, що буде працювати в декілька потоків і, на відміну від одного потоку, одночасно буде мати в роботі декілька заяв користувачів.

Серед факторів, що визначають необхідність створення даного проекту, знаходиться, безумовно, спроможність TPE підтримувати моніторинг ресурсів системи, що знаходиться у використанні, і не поглинати постійно нові ресурси, цим самим даючи можливість іншим процесам системи успішно функціонувати поруч з собою.

Зважаючи на те, що TPE містить в собі такі ж самі потоки, що будуть використовуватися при одно потоковій обробці, але й має ще набір інструментів та засобів для управління системними ресурсами, доцільно використовувати саме TPE.

3.2. Модель проекту багато потокової обробки даних

Як було зазначено вище, ціль проекту – створити систему багато потокової обробки даних громадян, що подають звернення до органів державної влади, котра дозволить громадян менший час очікувати офіційної відповіді на їхнє звернення, а також, безумовно, автоматизувати процес обробки подібних даних.

3.2.1. Технології, що були використані при створенні проекту

Нижче наведений вичерпний список технологій, задіяні для розробки проекту. Більшість з них є набором інструментів (frameworks), що розширюють стандартні можливості мови програмування Java та полегшують роботу програміста при створенні проекту «з нуля».

Список технологій:

- Java
- Spring Framework
- Spring MVC
- Spring Data
- Hibernate
- HSQLDB
- Apache Maven
- Apache Tomcat

- Apache Commons
- XStream

3.2.2. Опис моделі

Модель зводиться до маніпуляції (створення, наповнення, зміни, видалення) обєктів користувача (User.java), закладу (Department.java), бажаної дії (Action.java) та, безпосередньо, запиту користувача (Request. java).

Вихідний код (source code) даних класів знаходиться в Додатку А. Обєкти моделі, вказані вище, використовуються фреймворком Hibernate для швидкого доступу до них та збереження їх в базі даних.

Таблиця 3.1

Опис полів об’єкту користувача (User)

Назва поля	Тип та обмеження	Коментар
userId	NUMERIC	ID користувача
firstName	VARCHAR (50)	Ім’я користувача
lastName	VARCHAR (50)	Прізвище
phone	VARCHAR (14)	Контактний телефон
address	VARCHAR (100)	Домашня адреса
serialNumber	VARCHAR (8)	Серія та номер паспорту
identificationNumber	NUMERIC	Ідентифікаційний код

Таблиця 3.2

Опис полів об’єкту закладу (Department).

Назва поля	Тип та обмеження	Коментар
departmentId	NUMERIC	ID закладу
description	VARCHAR (100)	Назва закладу
address	VARCHAR (100)	Адреса закладу
phone	VARCHAR (14)	Контактний телефон

Таблиця 3.3

Опис полів об’єкту бажаної дії (Action)

Назва поля	Тип та обмеження	Коментар
actionId	NUMERIC	ID бажаної дії
description	VARCHAR (100)	Назва бажаної дії

Таблиця 3.4

Опис полів об’єкту запиту користувача (Request)

Назва поля	Тип та обмеження	Коментар
requestId	NUMERIC	ID запиту
description	VARCHAR (100)	Причина звернення
status	VARCHAR (40)	Поточний статус запиту
interval	VARCHAR (15)	Час обробки запиту

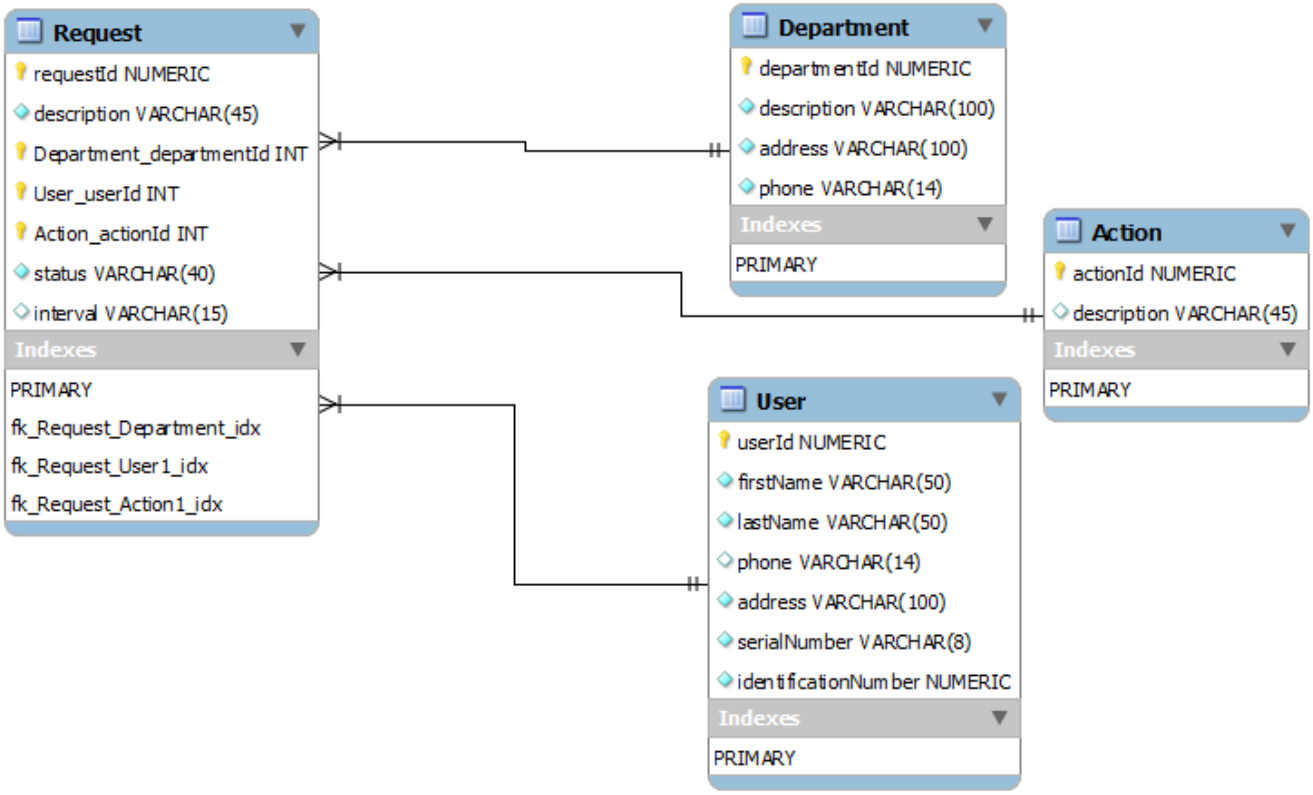

Рис. 3 1. Схема таблиць бази даних проекту

3.2.3. Опис схеми формування структури проекту

Проект створений на основі системи збірки декларативного типу Apache Maven, що визначає:

- структуру проекту, розміщення файлів в ньому
- залежності (сторонні бібліотеки), котрі використовуються локальним кодом
- тип пакету, в якому будуть розміщені скомпільовані класи та файли, потрібні під час виконання програми на сервері
- контекстний шлях, за яким буде доступна програма через браузер

Варто зауважити, що Apache Maven, як система збірки проекту, має основний конфігураційний файл, що має фіксовану назву pom.xml та міститься в корені директорії проекту. У конфігураційному файлі визначаються важливі аспекти, що були вказані вище.

3.2.4. Структура проекту

Таблиця 3.5

Основні директорії структури системи збірки Apache Maven

Шлях до директорії	Файли, що знаходяться в директорії
src/main/java	Вихідний код програми (*.java)
src/main/resources	Ресурси програми (XML, properties)
src/main/webapp	Ресурси WEB - додатку (HTML, CSS, JS)
src/test/java	Вихідний код тестів програми (*.java)
src/test/resources	Ресурси для тестів (XML, properties)

3.2.5. Залежності

Основним компонентом POM файлу є його список залежностей. Майже кожен проект залежить від інших. Для правильної побудови проекту за допомогою Maven необхідно правильно задати список залежностей (бібліотек). Maven завантажує та створює посилання на інші залежності для режиму компіляції та інших цілей, які вимагають інші проекти або підпроекти. В якості додаткового бонусу, Maven підтримує залежності інших залежностей (транзитивні залежності), що дозволяє зменшити список залежностей, щоб зосередитися виключно на залежностях, що вимагає поточний проект. Нижче наведено приклад визначення залежностей у файлі pom.xml [27]

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
..
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
```

```
      <artifactId>junit</artifactId>
      <version>4.0</version>
      <type>jar</type>
      <scope>test</scope>
      <optional>true</optional>
    </dependency>
..
  </dependencies>
..
</project>
```

Деталізуємо елементи файла конфігурації:

groupId, artifactId, version

Ці основні три компоненти залежності є важливими, так як задають місцеположення поточної залежності в локальному репозиторії. Кожна залежність буде знаходитися згідно наступного шляху: ${groupId}/${artifactId}/${version}.

classifier

Класифікатор дозволяє розрізняти артефакти, які були побудовані з того ж POM, але відрізняються за своїм змістом. Це свого необов'язковим і довільним рядком, що, якщо він присутній, додається до імені артефакту відразу після номера версії.

Розглянемо, наприклад, проект, який пропонує артефакт з орієнтацією на JRE 1.5, але в той же час і на артефакти, що, як і раніше, підтримує JRE 1.4. Перший артефакт може бути оснащений класифікатором типу jdk15, а другий - з jdk14, таким чином, що клієнти можуть вибрати, який з них використовувати.

Інший поширений випадок використання класифікаторів - необхідність додати вторинні артефакти до основного артефакту проекту. Якщо переглянути центральне

сховище Maven, ви помітите, що джерела класифікатори та Javadoc використовуються для розгортання вихідного коду проекту та API документацію разом з упакованими файлами класу.

type

Відповідає типу пакування поточного артефакту. За умовчанням, це JAR (Java Archive). Тип може бути зіставлений з іншим розширенням і класифікатора. Тип часто відповідає типу пакування, що використовується, хоча це також не завжди так. Деякі приклади JAR: EJB – клієнт (EJB – Enterprise Java Bean - специфікація технології написання і підтримки серверних компонентів, що містять бізнес-логіку. Є частиною Java Enterprise Edition) і test-JAR (JAR файли, що використовуються для запуску тестів та успішного їх виконання).

scope

Цей елемент визначає область та інтервал, коли дана залежність буде використана (компіляція і виконання, тестування і т.д.).

Є п'ять доступних областей :

- compile - це область за замовчуванням, використовується, якщо нічого не вказано. Компіляція залежностей доступна у всіх класах. Крім того, ці залежності поширюються на залежні проекти.
- provided - область схожа на область compile, але означає, що система збірки очікує, що JDK або контейнер забезпечить залежність під час виконання програми. Вона доступна тільки на етапі компіляції та виконанні тестів, і не є транзитивною.
- runtime - область показує, що залежність не потрібна для компіляції, але необхідна для виконання. Ця область потрібна саме під час виконання та тестування класів, а не для етапу компіляції.
- test - область показує, що залежність не потрібна для нормального використання програми, і доступна тільки для тестової компіляції на етапах виконання тестів.
- system - це область використовує залежності, що присутні як системні ресурси, бібліотеки.

optional

Дана особливість дозволяє іншим проектам знати, що, коли при використанні поточного проекту, він не вимагає цю залежність для того, щоб працювати коректно.

3.2.6. Типи пакування системи збірки Apache Maven

Існує декілька типів пакування вихідного коду:

- JAR - тип пакування проектів за замовчуванням, найбільш поширеним, і, таким чином, найбільш часто зустрічаються у файлах конфігурації.
- POM - простий тип упаковки. Артефакт, що генерує самий себе на відміну від JAR або EAR. Там немає коду для тестування або компіляції, і немає ніяких ресурсів для обробки.
- Maven-plugin - тип пакування аналогічний до JAR типу з трьома доповненнями : плагін : дескриптор, плагін : addPluginArtifactMetadata, і плагін : updateRegistry. Ці цілі створення файлу дескриптора і виконувати деякі зміни в даних репозиторію. Даний тип пакування представляє собою кінцеву програму, що в подальшому може маніпулювати файлами проекту.
- EJB або Enterprise Java Bean, є поширеним механізмом доступу до даних для розробки на основі моделей в Enterprise Java. Maven забезпечує підтримку EJB 2 і 3. Хоча необхідно налаштувати плагін EJB спеціально пакет для EJB3, інакше плагін за замовчуванням створюватиме пакет згідно до 2,1, не дивлячись на наявність певних конфігураційних файлів EJB.
- WAR - тип пакування схожий до JAR і EJB типів. Виняток становить пакет мета war: war. Зверніть увагу, що war:war мета вимагає конфігурацію web.xml у вашому src/main/webapp/WEB- INF.
- EAR ймовірно, є найпростішими Java EE конструкціями, що складаються в основному з файлу application.xml, деяких ресурсів і деяких модулів.
- Інші види упаковки. Це далеко не повний список всіх типів упаковки доступні для Maven. Є цілий ряд пакувальних форматів, доступних через зовнішні про-

екти і плагіни. Щоб використовувати один з цих типів пакування, необхідно дві речі: плагін, який визначає життєвий цикл для користувальницького типу упаковки і сховище, яке містить цей плагін. [28]

3.2.7. Опис робочого процесу проекту

Для перегляду потужностей (генерація тестових даних для запуску багато потокової обробки) проекту існує два шляхи: ручний та автоматичний.

Ручний:

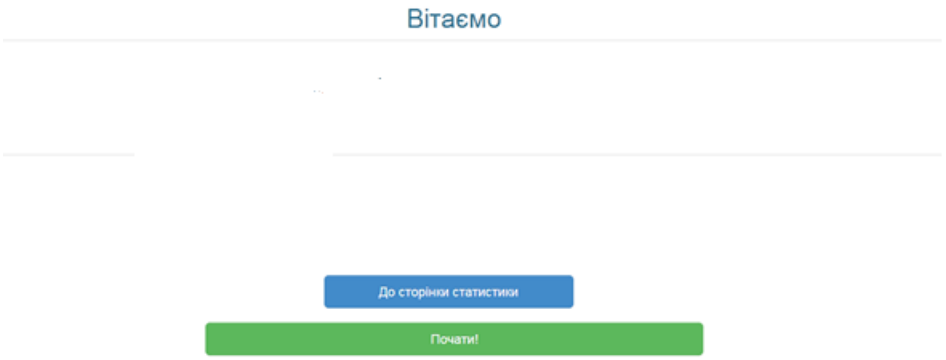

Рис. 3.2. Сторінка №1 проекту

1. Для переходу до форми заповнення даних необхідно натиснути на кнопку «Почати!»

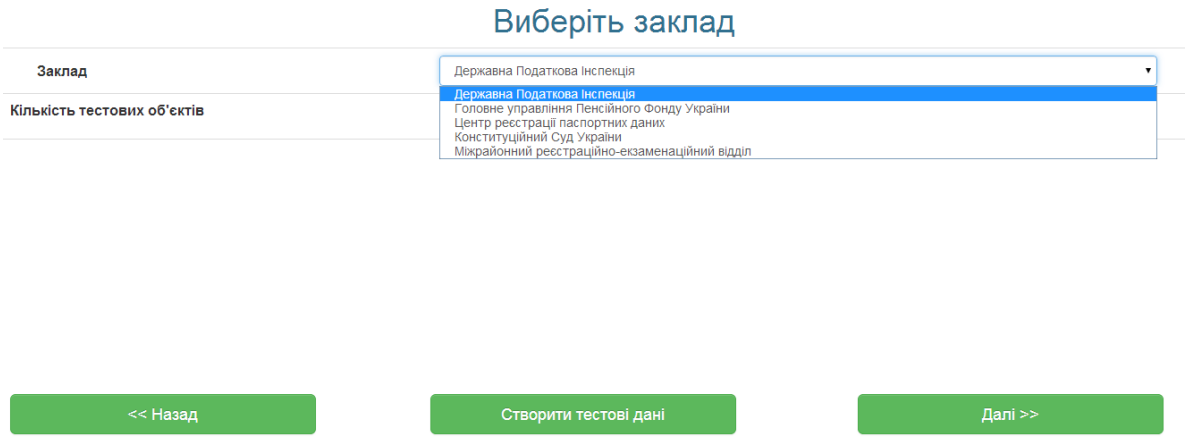

Рис. 3.3. Сторінка №2 проекту «Виберіть заклад»

2. Необхідно вибрати заклад в полі «Заклад» та натиснути «Далі>>»

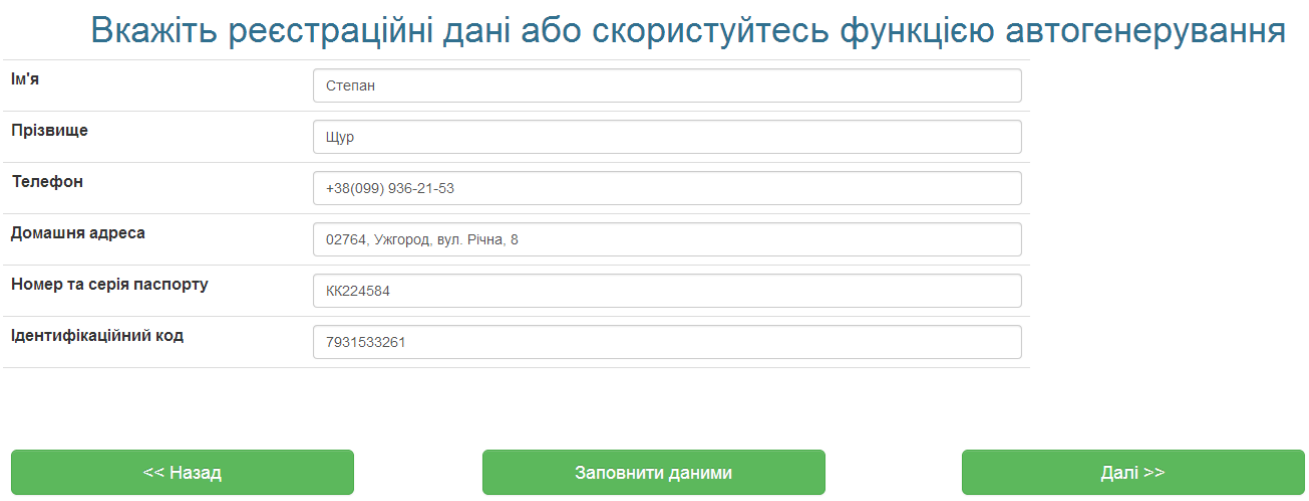

Рис. 3.4 Сторінка №3 проекту «Реєстрація користувача»

3. Для заповнення даних користувача можна скористатись функцією авто генерування даних, натиснувши на кнопку «Заповнити даними», або заповнити вручну, та натиснути кнопку «Далі>>».

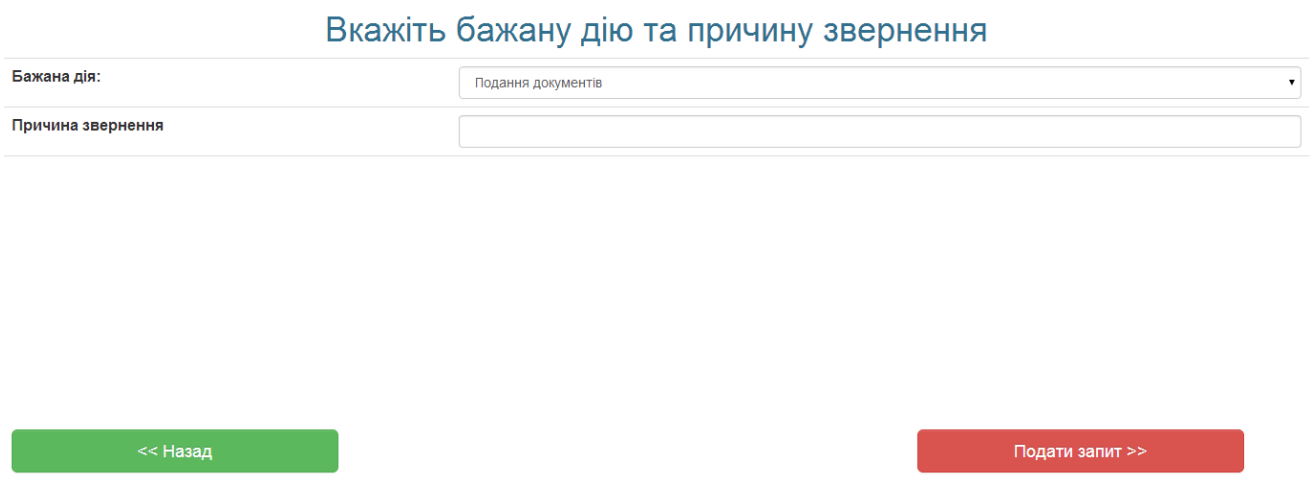

Рис. 3.5. Сторінка №4 проекту «Заповнення бажаної дії та причини звернення»

4. Необхідно вибрати бажану дію із доступних дій та заповнити «Причину звернення» та натиснути кнопку «Подати запит>>».

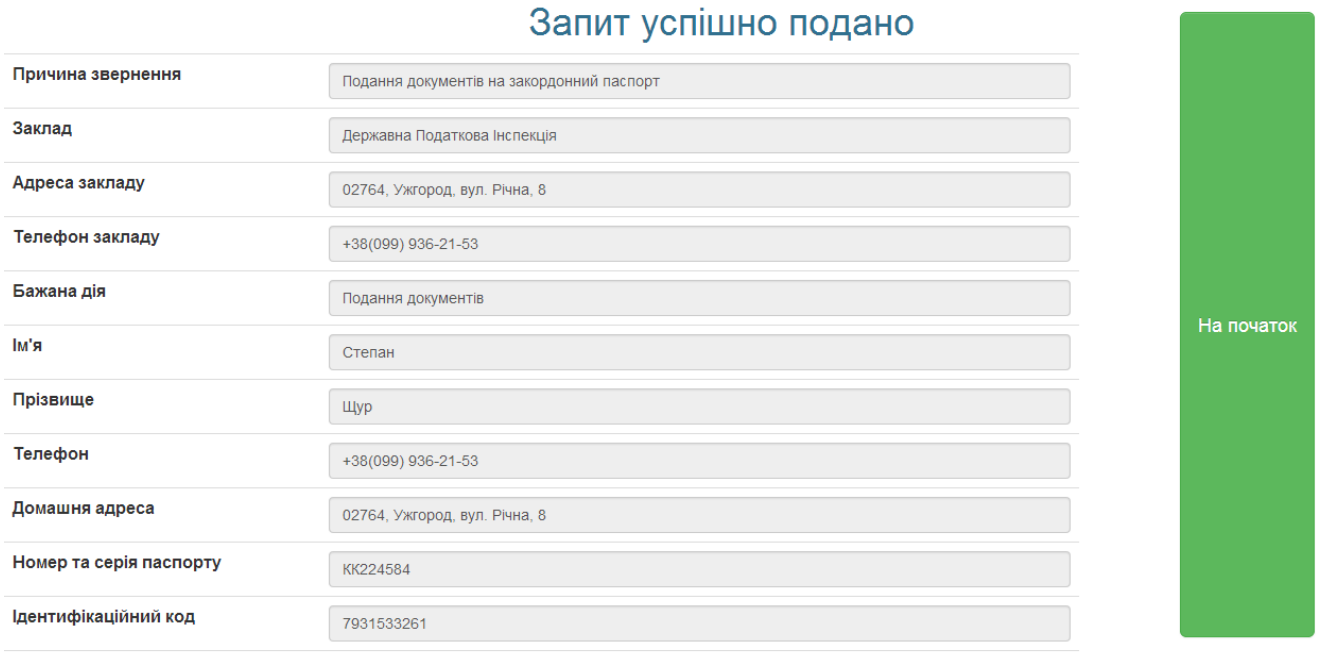

Рис. 3.6. Сторінка №5 проекту «Запит успішно подано»

5. На сторінці №5 показано всі дані, що заповнив користувач для звернення до конкретного закладу. Для переходу до сторінки №1 та повторного оформлення звернення необхідно натиснути на кнопку «На початок».
6. Після переходу на сторінку №1 і для перегляду статистики, необхідно натиснути кнопку «До сторінки статистики».

Рис. 3.7. Сторінка статистики

7. На сторінці статистики справа зверху сторінки знаходиться таблиця, що показує кожні 10 секунд кількість оброблених запитів користувачів. Таблиця, що займає нижню частину сторінки, показує всі запити, що були подані користувачами та статус кожного запиту (оброблено, в обробці, ще не оброблено). Кнопка «Оновити сторінку статистики» відповідає за перевантаження сторінки статистики. Кнопка «Видалити всі оброблені об’єкти» відповідає за видалення із сторінки всіх оброблених запитів (для зменшення кількості запитів, що відображаються, та полегшення пошуку потрібного запиту).

Автоматичний спосіб:

Для автоматичного способу справедливі 1 та 2 пункт ручного способу.

Після виконання дій, що зазначені в пункті 2, необхідно виконати:

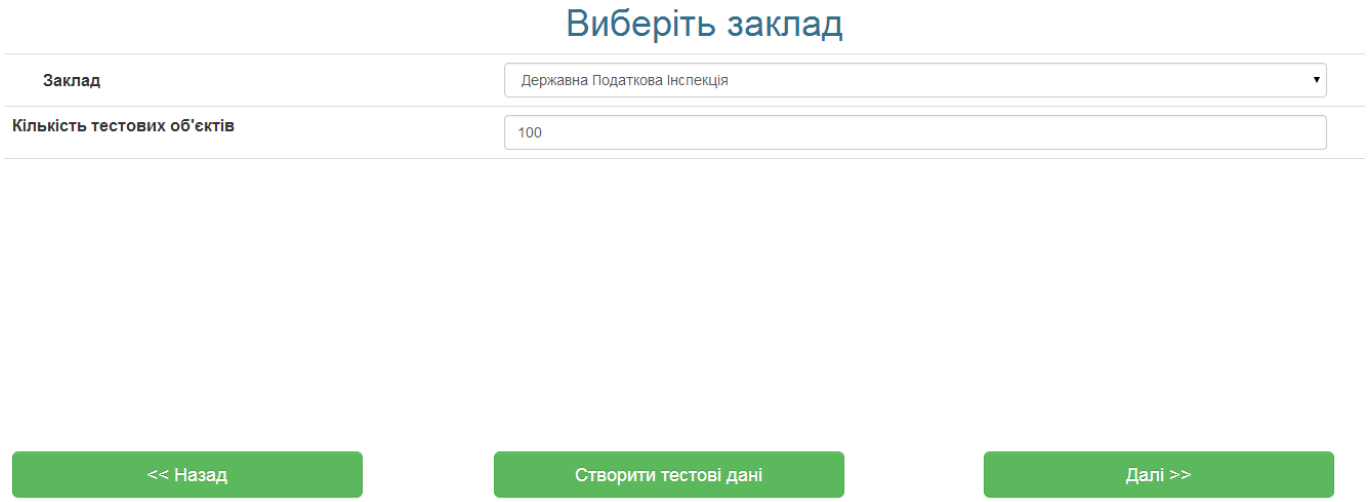

Рис. 3.8. Сторінка №2 «Вибір закладу» з можливістю генерації тестових даних

1. Необхідно ввести у поле «Кількість тестових об’єктів» число у діапазоні від 1 до 99999.

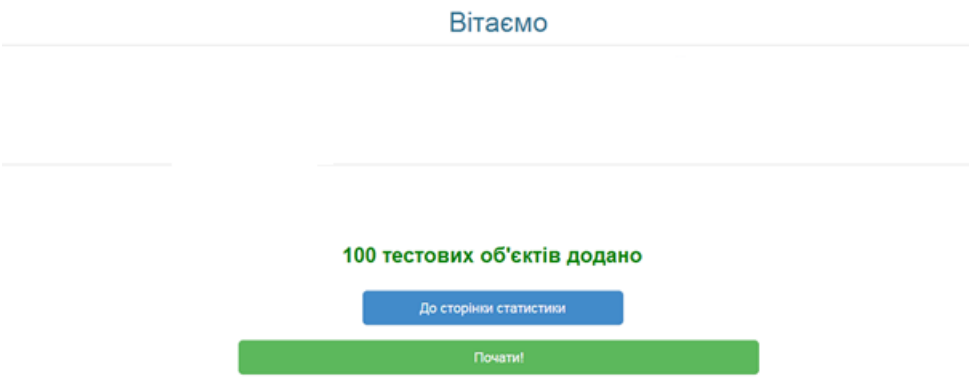

Рис. 3.9. Початкова сторінка з інформацією про кількість створених об’єктів

2. Для перегляду сторінки статистики необхідно натиснути на кнопку «До сторінки статистики». Далі – інформація справедлива пункту 8 ручного способу наповнення даних, що викладена вище.

Також на сторінках реалізовано фунціонал переходу на попередні сторінки, зберігаючи параметри.

3.2.8. Модель класів

Нижче описані всі класи проекту та ціль їхнього використання.

- com.diploma.constants.Constants – клас містить константи з назвами змінних, параметрів, аргументів, що використовуються у наступних класах: Action, Request, Department, User.
- com.diploma.domain.Action – клас містить в собі дані бажаної дії, котру може здійснити користувач
- com.diploma.domain.Request – клас містить в собі дані запиту користувача
- com.diploma.domain.User – клас містить в собі особисті дані користувача
- com.diploma.domain.Department – клас містить в собі дані про заклад
- com.diploma.mvc.wizard.WizardFlowComponent – клас, що використову-

ється класом WizardFormController та відповідає за маніпуляції із об’єктами за допомогою різних типів класу JPARepository, таких як ActionRepository, RequestRepository, DepartmentRepository та UserRepository

- com.diploma.mvc.wizard.WizardFormController – клас-контролер, що відповідає за обробку запиту до серверу та повернення правильної та адекватної відповіді користувачу
- com.diploma.repository.ActionRepository – клас, що відповідає за зв'язок із базою даних для класу Action
- com.diploma.repository.DepartmentRepository – клас, що відповідає за зв'язок із базою даних для класу Department
- com.diploma.repository.RequestRepository - – клас, що відповідає за зв'язок із базою даних для класу Request
- com.diploma.repository.UserRepository – клас, що відповідає за зв'язок із базою даних для класу User
- com.diploma.schedule.TaskScheduler – клас, що, згідно запланованого розкладу, запускається кожні 10 секунд та для всіх запитів користувачів, що маю статус «ще не оброблено», встановлює статус «в обробці». Таким чином запити потрапляють в ТРЕ
- com.diploma.schedule.TimerOfPerformedWork – клас, що, згідно запланованого розкладу, запускається кожні 10 секунд та запитує у бази даних кількість оброблених запитів, що існують на поточний момент, а потім зберігає цю кількість у об’єкті статистики для відображення на сторінці статистики
- com.diploma.singlethread.SingleThread – клас-виконавець задач-запитів в одно потоковому режимі
- com.diploma.task.WizardTask – клас, що служить «обгорткою» об’єкту-запиту для надання йому можливості бути виконаним за допомогою ТРЕ.

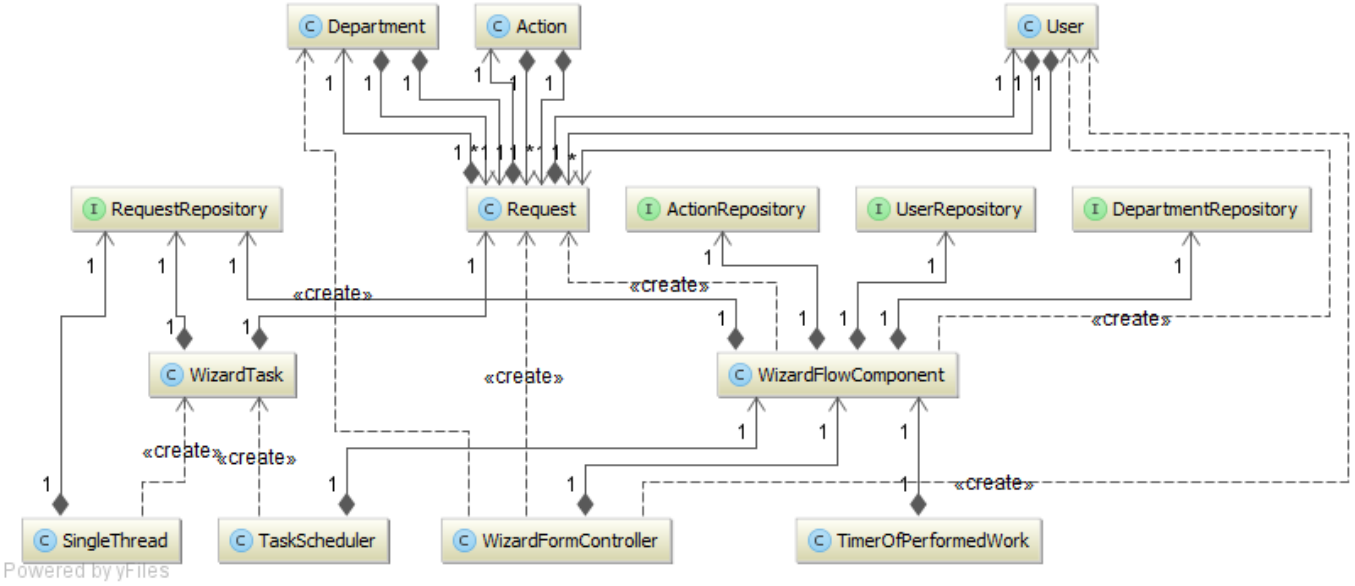

Рис. 3.10. Діаграма класів проекту

ВИСНОВОК ДО РОЗДІЛУ 3

Проект багато потокової обробки даних заяв громадян, що звертаються до органів державної влади, створено для збільшення швидкості і якості обробки запитів громадян та покращення цього обслуговування. Для демонстрації та перевірки роботи проекту під впливом великих даних була створена функціональність генерації тестових даних у розмірі від 1 до 99999 об'єктів, що дозволяє в один момент додати обрану кількість об'єктів. Також реалізовано два класи-планувальники (класи, що виконують свої запрограмовані дії згідно конкретного, вказаного заздалегідь, графіку). Один клас-планувальник відповідає за додання нових запитів громадян до системи багато потокової обробки (ТРЕ) кожні 10 секунд, а інший – за моніторинг кількості виконаних та оброблених запитів, що відбувається кожні 10 секунд, та надання цих даних для відображення на сторінці статистики.

Дана конструкція обробки даних працює на основі пулу потоків, що надає переваги у ефективному та оптимальному управлінні системними ресурсами, а також у використанні абсолютно всіх потоків, що знаходяться у пулі. Тобто у випадку, коли потік закінчив успішно обробляти конкретний запит, то він (потік) повертається назад у пул і стає доступним для залучення до обробки наступного запиту в черзі. Наступні дії повторюються згідно описаних вище дій до того часу, допоки не будуть оброблені всі запити.

У порівнянні двох варіантів реалізації: ТРЕ та один потік, ТРЕ має надзвичайні переваги. Всі 10 потоків, що наперед задані як ємність ТРЕ у конфігурації, включаються (запускаються) одночасно і відразу беруть у роботу по одній задачі кожному. Відповідно, умовно, за першу секунду роботи ТРЕ бере в роботу вже 10 задач (об'єктів), а один потік – лише одну.

Якщо взяти за одиницю виконання роботи час, що необхідний для виконання один потоком однієї задачі, то можна припустити, що за той самий час, коли один потік буду обробить одну задачу, десять потоків ТРЕ оброблять 10 задач. Але необхідно зауважити, що можуть виникнуть «накладки» в часі, зв'язані з ресурсами, тобто навіть за найоптимальніших обставин десять потоків не виконають та не оброб-

лять десять задач за час, що знадобиться одному потокові для обробки цієї самої кількості задач. В залежності від системних алгоритмів, що відповідають за виділення, управління та моніторинг ресурсів системи цей час буде звісно менший, ніж час роботи одного потоку над 10 задачами, але, в той же час, наскільки він буде менший залежить безпосередньо від оптимальності вищенаведених алгоритмів системи.

РОЗДІЛ 4

КОМПОНЕНТИ ТА ПОРІВНЯЛЬНІ ХАРАКТЕРИСТИКИ ПРОЕКТУ БАГАТОПОТОКОВОЇ ОБРОБКИ ДАНИХ

4.1. Опис основних компонентів проекту

Spring Framework забезпечує вирішення багатьох завдань, з якими стикаються Java розробники та організації, які хочуть створити інформаційну систему, засновану на платформі Java. За широкої функціональності важко визначити найбільш значущі структурні елементи, з яких він складається. Spring Framework НЕ повністю пов'язаний з платформою Java Enterprise, незважаючи на його масштабну інтеграцію з нею, що є важливою причиною його популярності.

Spring Framework, ймовірно, найбільш відомий як джерело розширень (особливості), потрібних для ефективної розробки складних бізнес -додатків поза великовагових програмних моделей, які історично були домінуючими в промисловості. Ще одне його достоїнство в тому, що він ввів раніше не використовувані функціональні можливості в сьогоднішні панівні методи розробки, навіть поза платформи Java.

Цей фреймворк пропонує послідовну модель і робить її застосовною до більшості типів додатків, які вже створені на основі платформи Java. Вважається, що Spring Framework реалізує модель розробки, засновану на кращих стандартах індустрії, і робить її доступною в багатьох областях Java. [29]

Spring Data JPA надає модель програмування репозиторію, яка починається з інтерфейсу на один об'єкт:

```
public interface AccountRepository extends JpaRepository<Account, Long> { … }
```

Створення даного інтерфейсу має дві причини:

1. за рахунок розширення JpaRepository отримуємо велику кількість універсальних методів CRUD, орієнтований на тип конкретного обєкту, що дозволяє зберігати, видаляти їх і так далі.

2. це дозволить інфраструктура репозиторіїв Spring Data JPA сканувати класи для цього інтерфейсу і створити Spring bean для нього.

Для того, щоб Spring-контейнер зміг створити компонент, який реалізує цей інтерфейс, все, що необхідно, це використовувати простір імен Spring JPA і активувати підтримку репозиторія, використовуючи відповідний елемент.

<jpa:repositories base-package="com.acme.repositories" />

На лістингу вище наведено приклад сканування всіх пакетів нижче com.acme.repositories для інтерфейсів, що наслідують JpaRepository і створює Spring bean для нього, який має за основу реалізацію класу SimpleJpaRepository.

Autowiring – зв’язування між класами, що знаходяться у Spring-контейнері, котре реалізовується на ступним чином: якщо клас заданий із анотацією @Autowired в іншому класі, то для знаходження екземпляру даного класу відбувається пошук у контексті Spring-контейнеру, після чого відбувається присвоює посилання знайденого екземпляру до посилання класу, що був анотований. [31]

Існує декілька видів властивості Autowiring:

- @ Autowired і властивість класу

Властивості класу з анотацією @ Autowired заповнюються відповідними значеннями відразу після створення bean'a (екземпляр класу у термінології Spring) і перед тим, як будь-який з методів класу буде викликаний. Нижче наведено приклад Autowiring для властивості класу.

@Autowired

private FieldService propertyService;

- @ Autowired і сетер

Традиційне використання анотації. Значення передається як аргумент до методу, в якому відбувається присвоєння. Приклад такого застосування наведено нижче.

```
@ Autowired
public void setSetterService (SetterService setterService) {
    this.setterService = setterService;
}
```

- @ Autowired і конструктор класу

Тільки один конструктор може виконувати цю анотацію. Цей конструктор може бути будь-якого типу (private, protected), а не тільки public. Приклад такого застосування наведено нижче.

```
@ Autowired
public BeanContainer (ConstructorService constructorService) {
    this.constructorService = constructorService;
}
```

- @ Autowired і метод

Анотація @Autowired може бути використана в методі з будь-яким ім'ям і з будь-якою кількістю прийнятих параметрів. У цьому випадку Spring спробує привласнити кожному аргументу значення відповідних bean'a. Метод не зобов'язаний бути public. Приклад такого застосування наведено нижче.

```
@ Autowired
public void multipleArguments (SetterAService setterAService,
    SetterBService setterBService) {
    this.setterAService = setterAService;
    this.setterBService = setterBService;
}
```

Hibernate - бібліотека для мови програмування Java, призначена для вирішення завдань об'єктно-реляційного відображення. Вона являє собою вільне програмне

забезпечення з відкритим вихідним кодом (Open Source), яке розповсюджується на умовах GNU Lesser General Public License. Дана бібліотека надає легкий у використанні каркас (фреймворк) для відображення об'єктно-орієнтованої моделі даних в традиційні реляційні бази даних. [32]

Apache Tomcat (у старих версіях - Catalina) - контейнер сервлетів з відкритим вихідним кодом, що розробляється Apache Software Foundation. Реалізує специфікацію сервлетів і специфікацію JavaServer Pages (JSP) і JavaServer Faces (JSF). Написаний на мові Java.

Tomcat дозволяє запускати веб-додатки, містить ряд програм для самоконфігурації. [33]

HSQLDB - реляційна СУБД з відкритим вихідним кодом. Розповсюджується по власної ліцензії, близької до ліцензії BSD. Підтримує стандарти SQL-92, SQL: 1999, SQL: 2003 і SQL: 2008.

HSQLDB повністю написана на Java і відрізняється невеликим розміром (розмір близько 1100 кБ для версії 2.0). Може використовуватися і як окремий сервер з підтримкою мережевих з'єднань по JDBC, і у вигляді бібліотеки для використання безпосередньо в коді програми.

HSQLDB використовується в багатьох відомих програмних продуктах, зокрема, в LibreOffice, OpenOffice.org, JBoss, Openfire, JAMWiki. [34]

Apache Maven - фреймворк для автоматизації збирання проектів, специфіковані на XML-мові POM (англ. проектно - об'єктна модель).

Слово Maven походить з мови ідиш і означає приблизно «збирач знання».

Maven, на відміну від іншого збирача проектів Apache Ant, забезпечує декларативну, а не імперативну збірку проекту. Тобто, в файлах проекту pom.xml міститься його декларативний опис, а не окремі команди. Усі завдання з обробки файлів Maven виконує через плагіни.

Java Persistence API (JPA) - API, що входить з версії Java 5 до складу платформ Java SE і Java EE, надає можливість зберігати в зручному вигляді Java -об'єкти в базі даних. [35]

Існує декілька реалізацій цього інтерфейсу, одна з найпопулярніших викорис-

товує для цього Hibernate.

Підтримка збереження даних, що надається JPA, покриває області:

- безпосередньо API, заданий в пакеті javax.persistence;
- платформо - незалежний об'єктно -орієнтована мова запитів Java Persistence Query Language;
- метаінформація, що описує зв'язки між об'єктами.
- генерація DDL для сутностей

Entity (Сутність) - клас, пов'язаний з БД за допомогою анотації (@ Entity) або через XML. До такого класу ставляться такі вимоги:

- Повинен мати порожній конструктор (public або protected)
- Не може бути вкладеним, інтерфейсом або enum
- Не може бути final і не може містити final-полів
- Повинен містити хоча б одне @Id -поле
- При цьому entity може :
- Утримувати не порожні конструктори
- Успадковуватися і бути успадкованим
- Утримувати інші методи і реалізовувати інтерфейси

Сутності можуть бути пов'язані один з одним (один-до-одного, один-до-багатьох, багато-до-одного і багато-до-багатьох).

4.2. Опис важливих компонентів («ядра» проекту)

Основні два компоненти робочого процесу проекту – це класи WizardFlowComponent та WizardFormController, що відповідають за більшість процесів та дій, що відбуваються в межах проекту під час додання, зміни, видалення запитів.

В пакеті com.diploma.repository знаходяться чотири класи для маніпуляції даними об’єктів User, Request, Department, Action, котрі мають відповідні назви: UserRepository, RequestReporitory, DepartmentRepository, ActionRepository. Кожний із репозиторіїв є дочірнім класом до класу JpaRepository, що дає змогу використовува-

ти основні CRUD (англ. Create Read Update Delete – Створення Читання Оновлення Видалення) операції, зокрема, пошук всіх елементів в таблиці, пошук елементу по його унікальному ідентифікатору, збереження, оновлення, звичайне та видалення із затримкою елементів, а також забезпечує підтримку додаткових запитів.

WizardFlowComponent містить посилання на всіх JPA репозиторії, описані вище. Для утримання посилань на дані репозиторії використовується дуже важлива властивість Spring-контейнеру така, як зв'язування (autowiring).

4.2.1. WizardFlowComponent: опис методів

Клас WizardFlowComponent є одним із двох важливих елементів «ядра» системи та містить наступні методи:

- registerUser – відповідає за збереження всіх даних, що були введені користувачем на сторінці «введення даних користувача»
- chooseDepartment – відповідає за вибір та збереження даних про вибраний заклад згідно його ID, що передається в метод як параметр
- init – відповідає за наповнення таблиць Actions та Departments заздалегідь приготованими даними із XML файлів availableActions.xml та availableDepartments.xml відповідно
- defineActionMap – відповідає за створення і повернення карти (структура даних за формою ключ-значення), що містить actionId як ключ та description як значення. Дана карта використовується при виборі бажаної дії користувачем
- saveSelectedActionInRequest – відповідає за збереження даних про вибрану бажану дію
- saveRequestOnFinish – відповідає за остаточне збереження даних про запит, користувача, закладу та бажаної дії до БД на етапі подачі запиту
- fillUserByTestData – відповідає за заповнення об'єктів користувача тестовими даними. Набір для генерації визначений у класі, що розглядається. Дана функціональність дозволяє генерувати тестові дані у великій кількості, не проходячи вручну всі кроки і не заповнюючи всі обов'язкові форми та поля

- fillDepartmentByTestData - відповідає за заповнення об’єктів закладу тестовими даними. Набір для генерації визначений у класі, що розглядається.
- fillActionByTestData - відповідає за заповнення об’єктів бажаних дій тестовими даними. Набір для генерації визначений у класі, що розглядається.
- fillReasonsByTestData - відповідає за заповнення причин звернення до закладу тестовими даними. Набір для генерації визначений у класі, що розглядається.
- fillRequestByTestData - відповідає за заповнення об’єктів запитів тестовими даними. Набір для генерації визначений у класі, що розглядається.

4.2.2. WizardFormController: опис методів

Клас WizardFormController є одним із двох важливих елементів «ядра» системи та містить наступні методи:

- processPage – відповідає за керування руху між сторінками, реалізує функціонал переходу на попередні та наступні сторінки
- returnPage – відповідає за повернення попередньої сторінки із повідомленнями помилок, якщо дані на сторінці не вірно заповнено
- getInitialPage – відповідає за формування першої сторінки проекту
- getCancelPage – відповідає за повернення до першої сторінки проекту у випадку фатальних помилок на інших сторінках проекту
- finish – відповідає за завершення реєстрації запиту до закладу, викликає збереження даних до БД
- getStatisticsPage – відповідає за формування сторінки статистики
- deletePerformedEntries – відповідає за «видалення» оброблених запитів лише зі сторінки статистики для полегшення і пришвидшення завантаження сторінки при повторних оновленнях останньої.

4.3. Порівняння результатів засобів обробки даних

Очевидно, що обробка даних за допомогою багатьох потоків та одного потоку

відрізняється. Зокрема, у швидкості та оптимальності рішення зі сторони першого.

Як варіант БП обробки даних у проекті взято ТРЕ із ємністю 10 потоків, що працюють одночасно. Як спосіб ОП обробки даних обрано один потік – com.diploma.singlethread.SingleThread – реалізація класу-потоку Thread в Java.

Виміри проведено у розрізі кількісних характеристик: кількість оброблених об’єктів \ час виконання та подано у вигляді графіків залежностей даних величин.

Вимірювання проведено згідно наступних критеріїв:

1. 100 об’єктів (запитів громадян)
2. 1000 об’єктів
3. 3000 об’єктів

Вимірювання за кожним із вищенаведених пунктів вважається успішним, якщо обробка всієї кількості об’єктів одним \ багатьма потоками остаточно завершена. Фіксація кількості оброблених об’єктів проводиться класом TimerOfPerformedWork кожні 10 секунд. Період часу, за який виконується обробка об’єктів, зазначений у секундах.

Перед кожним вимірюванням сервер рішень Apache Tomcat було запущено заново, щоб уникнути впливу різного роду можливих проблем, що зв’язані з ресурсами чи фоновими процесами системи. Відповідно кожного разу не існує жодних факторів, що могли б погіршити результати того чи іншого вимірювання.

4.3.1. Результати виміру швидкості обробки 100 об’єктів

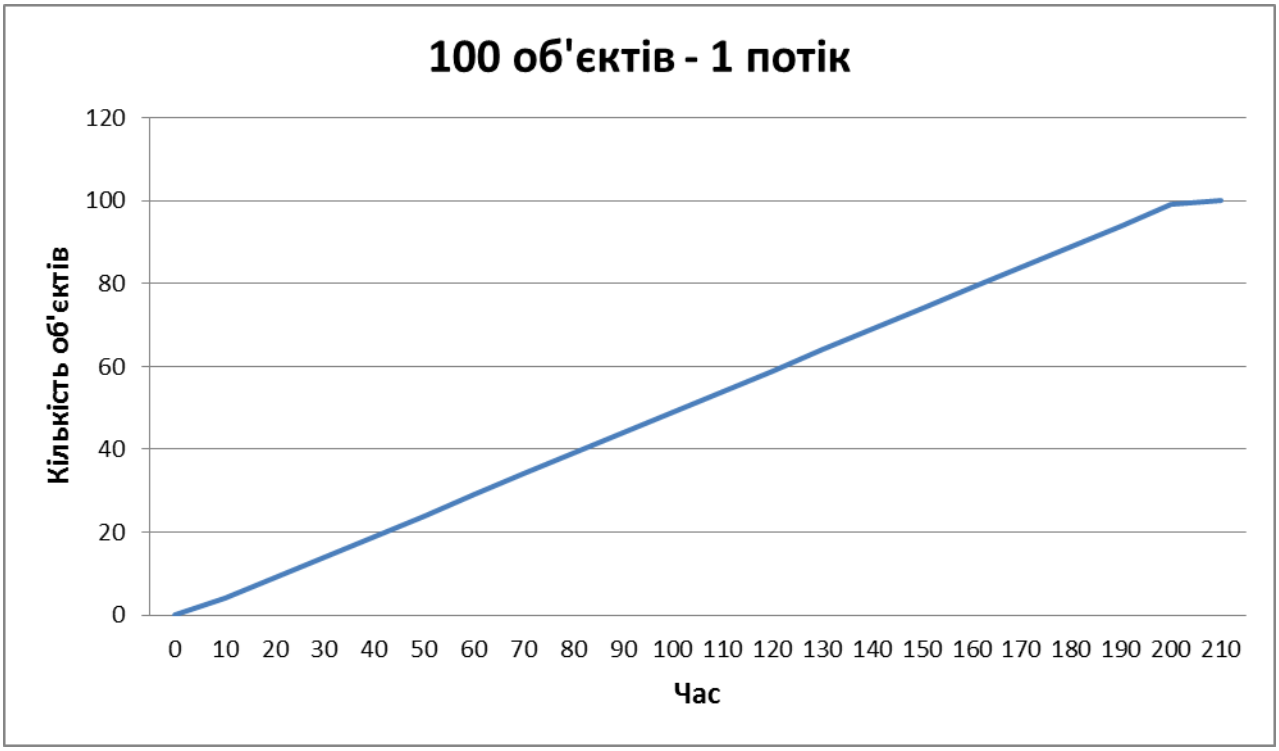

Рис. 4.1. Графік залежності кількості оброблених об’єктів (100 об’єктів) на момент часу за допомогою одного потоку

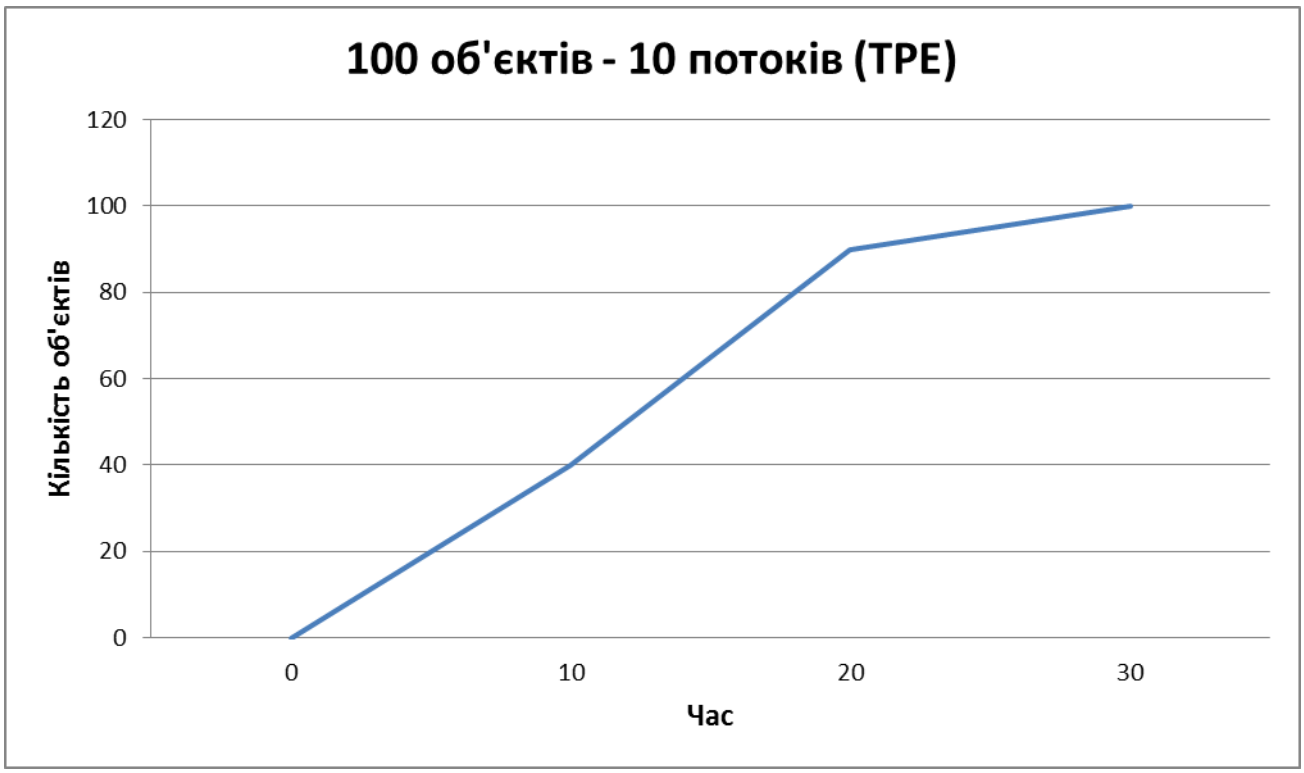

Рис. 4.2. Графік залежності кількості оброблених об’єктів (100 об’єктів) на момент часу за допомогою TPE (10 потоків)

Результати вимірювання:

Згідно графіків робимо висновок, що 100 об’єктів одним потоком було оброблено за 210 секунд (3 хв. 30 с), а TPE обробив об’єкти за 30 с.

Відповідно: 100 об’єктів TPE обробив швидше у 7 разів.

4.3.2. Результати виміру швидкості обробки 1000 об’єктів

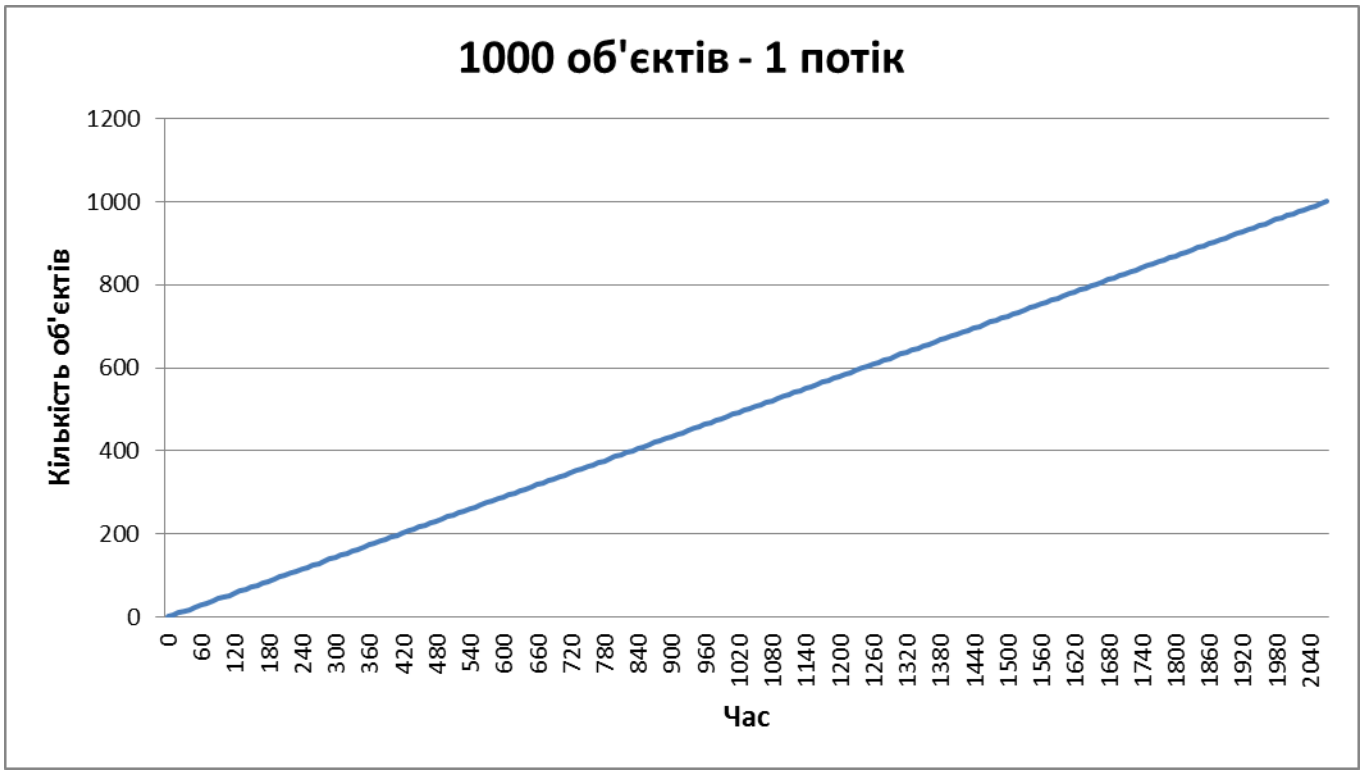

Рис. 4.3. Графік залежності кількості оброблених об’єктів (1000 об’єктів) на момент часу за допомогою одного потоку

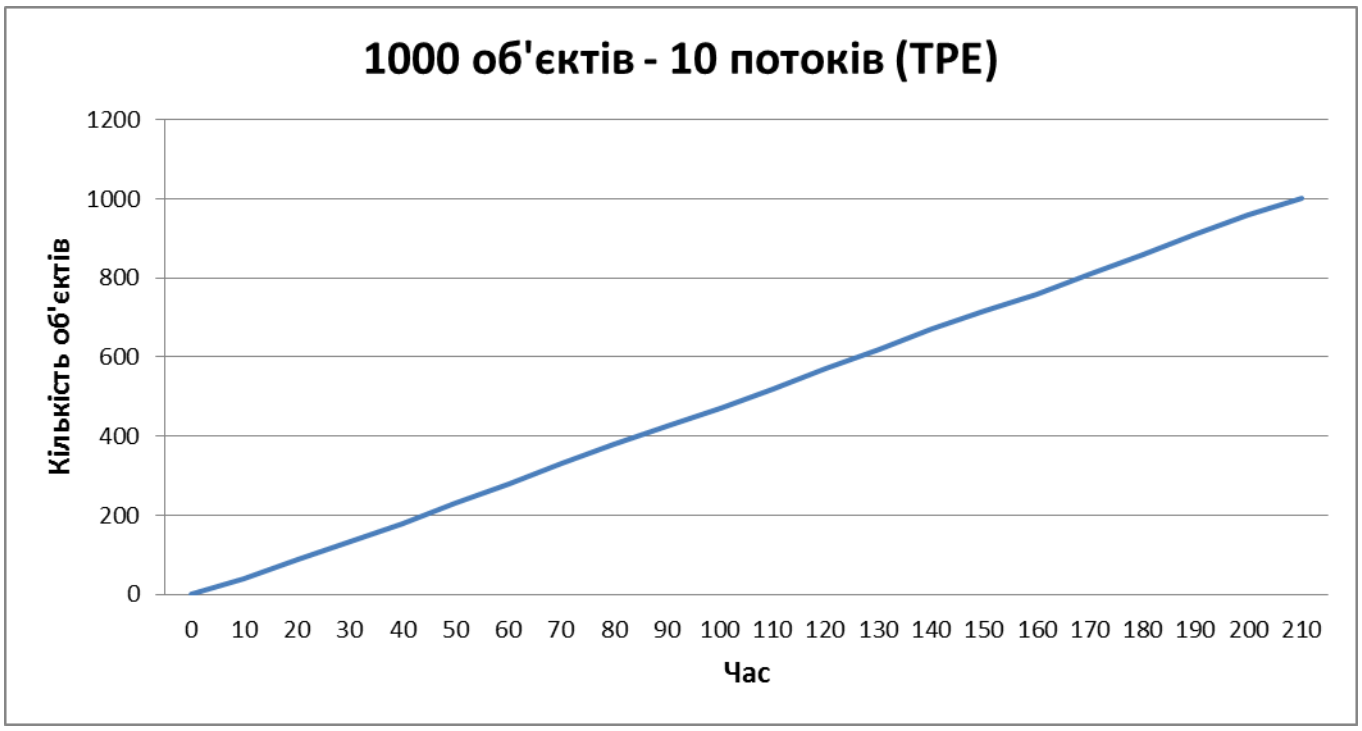

Рис. 4.4. Графік залежності кількості оброблених об’єктів (1000 об’єктів) на момент часу за допомогою TPE (10 потоків)

Результати вимірювання:

Згідно графіків робимо висновок, що 1000 об’єктів одним потоком було оброблено за 2040 секунд (34 хв.), а TPE обробив об’єкти за 210 с (3 хв. 30 с).

Відповідно: 1000 об’єктів TPE обробив швидше у 9,72 разів.

4.3.3. Результати виміру швидкості обробки 3000 об’єктів

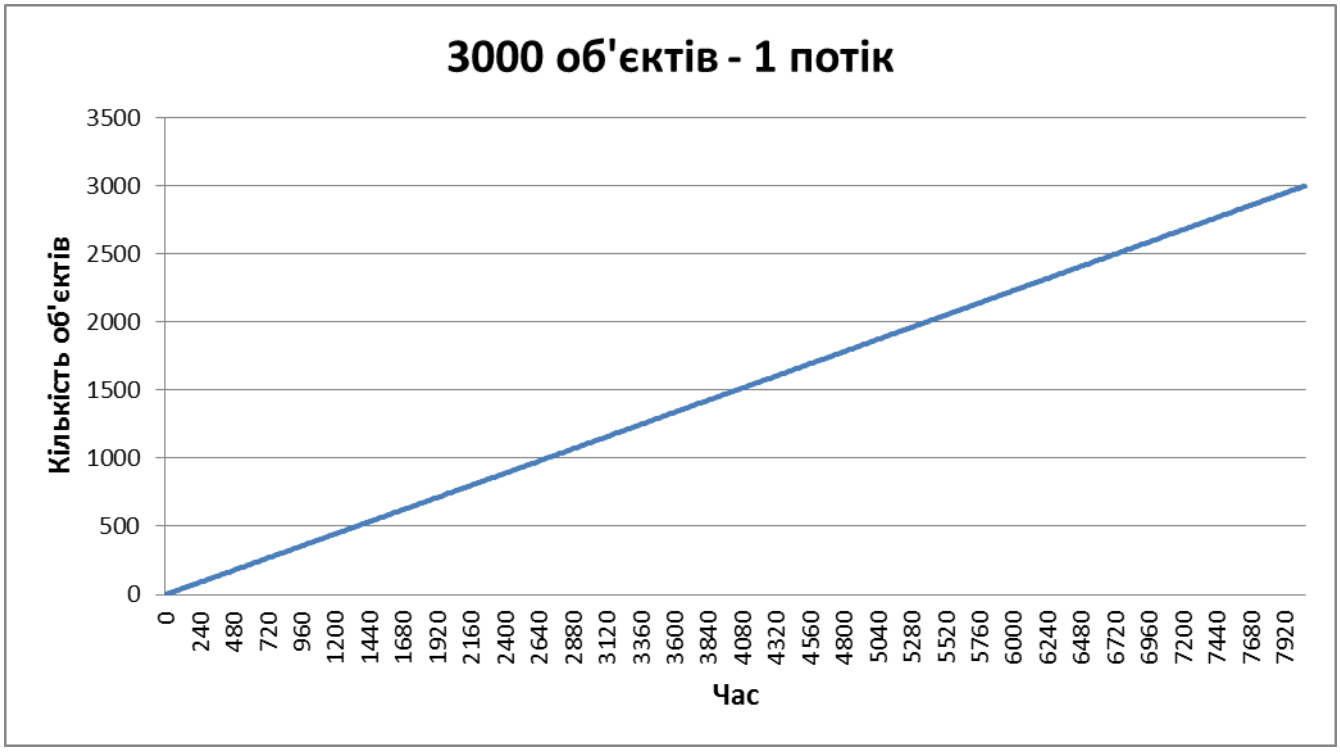

Рис. 4.5. Графік залежності кількості оброблених об’єктів (3000 об’єктів) на момент часу за допомогою одного потоку.

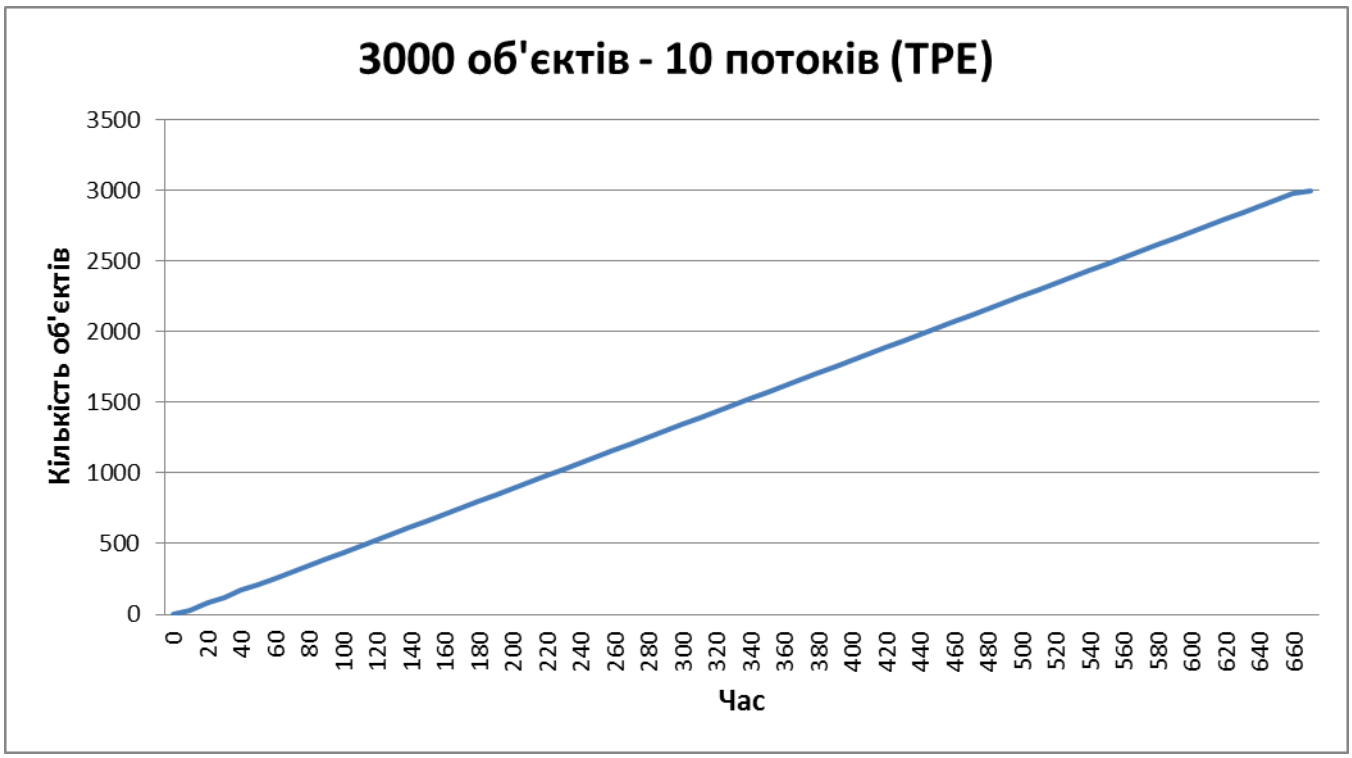

Рис. 4.6. Графік залежності кількості оброблених об’єктів (3000 об’єктів) на момент часу за допомогою TPE (10 потоків)

Результати вимірювання:

Згідно графіків робимо висновок, що 3000 об’єктів одним потоком було оброблено за 8050 секунд (2 год. 14 хв. 10 с), а ТРЕ обробив об’єкти за 660 с (11 хв.).

Відповідно: 3000 об’єктів ТРЕ обробив швидше у 12,2 разів.

ВИСНОВОК ДО РОЗДІЛУ 4

Для проведення досліджень було вирішено використовувати два засоби: ТРЕ з ємністю 10 потоків, що працюють паралельно та незалежно одне від одного, а також реалізацію одного самостійного та незалежного потоку, котру виконано у класі com.diploma.singlethread.SingleThread .

Заміри часу було проведено за допомогою класу TimerOfPerformedWork, що, згідно запланованого розкладу, запускається кожні 10 секунд та запитує у бази даних кількість оброблених запитів, що існують на поточний момент, а потім зберігає цю кількість у об’єкті статистики для відображення на сторінці статистики.

На протязі проведення досліджень по замірюванні кількісних характеристик для наборів тестових об’єктів у кількісті: 100, 1000 та 3000 відповідно було виявлено, що при збільшенні кількості тестових об’єктів час їхньої обробки зростає для обидвох засобів, але для реалізації ОП обробки часовий інтервал збільшується надзвичайно стрімко, що зменшує можливість отримання статистики за весь інтервал.

Перше замірювання було проведено для кількості 100 тестових об’єктів. І вже на такому малому об’єму даних структура ТРЕ показала перші результати.

Отож, згідно дослідження обробки 100 об’єктів і графіків-результатів робимо висновок, що 100 об’єктів одним потоком було оброблено за 210 секунд (3 хв. 30 с), а ТРЕ обробив об’єкти за 30 с. Відповідно ТРЕ впорався із поставленою задачею в СІМ разів швидше.

Друге замірювання було проведено для кількості 1000 тестових об’єктів. Перед початком замірювання очікувалось, що так як кількість об’єктів збільшена в 10 разів, то, логічно, що і часові інтервали повинні були збільшитись у 10 разів відносно першого замірювання. Тобто очікувалось, що ТРЕ обробить дані за 300 с, а один потік – за 2100 с.

Але згідно дослідження обробки 1000 об’єктів і графіків-результатів робимо висновок, що 1000 об’єктів одним потоком було оброблено за 2040 секунд (34 хв.), а ТРЕ обробив об’єкти за 210 с (3 хв. 30 с). Відповідно ТРЕ впорався із поставленою задачею у 9,72 разів.швидше, ніж олин потік, що є дось істотним приростом у швид-

кості обробки даних.

В результаті згідно дослідження отримані результати перевищують очікування: 1000 об’єктів оброблено за 210 с за допомогою TPE, що на 90 с швидше, ніж очікувалось.

Третє замірювання було проведено для кількості 3000 об’єктів. Результати третього вимірювання не залишаюсь сумніви у потужності такого засобу БП обробки даних як TPE.

Отже, результати третього вимірювання показали що 3000 об’єктів одним потоком було оброблено за 8050 секунд (2 год. 14 хв. 10 с), а TPE обробив об’єкти за 660 с (11 хв.). Відповідно: TPE впорався із поставленою задачею у 12,2 разів швидше, ніж один потік.

Варто зауважити, що час обробки одним потоком з кожним вимірюванням надзвичайно збільшується, що не залишає будь-який аргументів на користь такої обробки.

ВИСНОВКИ

Незважаючи на перший погляд простоту створення і підтримку потоків, насправді це не є ідеальне рішення для систем з великим навантаженням та вимогами високої продуктивності і швидкості обробки, виконання, оновлення інформації. Перша причина – це складність підтримки великої кількості потоків, які фізично представляють собою різні об’єкти в пам’яті віртуальної машини Java, нічим між собою не зв’язаних. Розробник логічно пов’язує всі потоки разом, так як вони обробляють ті ж самі елементи бізнес-логіки, але насправді це не так згідно концепції віртуальної машини Java. Друга причина – взаємні блокування потоків. Це найбільш поширена проблема великої кількості потоків, не об’єднаних в пул. Так як трапляються випадки, коли потік А очікує на доступність ресурсів сервера, котрі вже використовуються потоком Б, а потік Б. в свою чергу, очікує ресурси, зайняті потоком А. В результаті отримуємо блокування, що може продовжуватись вічно.

Кожна програма Java використовує потоки. Більшість програмістів використовує Java- інструментарії для користувача інтерфейсу (AWT або Swing), сервлети Java, RMI, сторінки JavaServer або технологію Enterprise JavaBeans, що є самі по собі багато потоковими.

Існує безліч ситуацій, коли ви можете побажати використовувати потоки в явній формі для підвищення продуктивності, чуйності або організованості своїх програм. Ці ситуації включають:

- підвищення чуйності інтерфейсу користувача під час виконання тривалих завдань;
- застосування багатопроцесорних систем для паралельного управління декількома завданнями;
- спрощення моделювання або систем на основі агентів
- виконання асинхронної або фонової обробки.

І хоча програмний інтерфейс потоків досить простий, про написання поточно-орієнтованих програм цього не скажеш. При спільному використанні змінних декількома потоками потрібно ретельно стежити за тим, щоб процеси читання і запису цих змінних були відповідним чином синхронізовані. При виконанні запису змінної, яку може використовувати інший потік, або при виконанні читання змінної, яка може бути записана іншим потоком, ви повинні використовувати синхронізацію, щоб зміни даних були видні всім потокам.

Пул потоків - корисний інструмент для організації серверів та додатків. Він досить простий по суті, але є деякі моменти, з якими слід бути обережними під час застосування та використання, такі як взаємне блокування, пробуксовка ресурсів, і складнощі, пов'язані з wait () і notify (). Якщо вам буде потрібно пул потоків для вашої програми, розгляньте використання одного з класів Executor з util.concurrent, такий як ThreadPoolExecutor, замість створення нового. Якщо потрібно створити потоки для вирішення короткострокових завдань, безумовно слід розглянути використання замість цього пул потоків.

Більшість реалізацій структури Executor в java.util.concurrent використовують пули потоків, які складаються з робочих потоків. Цей вид потоку існує окремо від Runnable і Callable завдань і часто використовується, щоб виконати багаторазові завдання.

Пул потоків, використовуючи робочі потоки мінімізує витрати, належно розпаралелювати виконання завдань. Об'єкти потоків використовують істотну кількість пам'яті, і у великомасштабному додатку, створюють істотні витрати управління пам'яттю.

Один загальний тип пулу потоків є фіксованим пулом потоків. У цього типу пулу завжди є конкретна кількість виконання потоків; якщо потік так чи інакше завершується, в той час як пул потоків знаходиться все ще у використанні, то автоматично замінюється новим потоком. Завдання для пулу зберігаються у внутрішній черці.

Важлива перевага фіксованого пулу потоків полягає в тому, що додатки, використовуючи його погіршуються коректно. Щоб зрозуміти це, розгляньте заяву веб -

сервера, де кожен запит HTTP обробляється окремим потоком. Якщо додаток просто створить новий потік для кожного нового запиту HTTP, і система отримує більше запитів, ніж може відразу обробити, то додаток раптово припинить відповідати на всі запити, коли витрати всіх тих потоків перевищать ємність системи. З межею на числі потоків, які можуть бути створені, програма не буде обслуговувати запити HTTP так швидко, як вони входять, але буде обслуговувати їх так швидко, як система може витримати.

Хоча пул потоків - потужний механізм для структурування БП додатків, він пов'язаний з певним ризиком. Додатки, побудовані за допомогою пулів потоків, схильні до всіх паралельних ризиків, що й будь-який інший БП додаток, як, наприклад, помилки синхронізації і взаємне блокування, і також декільком іншим ризикам, специфічних також для пулів потоків, таких, як залежне від пулів взаємне блокування, пробуксовка ресурсів.

Проект багато потокової обробки даних заяв громадян, що звертаються до органів державної влади, створено для збільшення швидкості і якості обробки запитів громадян та покращення цього обслуговування. Для демонстрації та перевірки роботи проекту під впливом великих даних була створена функціональність генерації тестових даних у розмірі від 1 до 99999 об’єктів, що дозволяє в один момент додати обрану кількість об’єктів. Також реалізовано два класи-планувальники (класи, що виконують свої запрограмовані дії згідно конкретного, вказаного заздалегідь, графіку). Один клас-планувальник відповідає за додання нових запитів громадян до системи багато потокової обробки (ТРЕ) кожні 10 секунд, а інший – за моніторинг кількості виконаних та оброблених запитів, що відбувається кожні 10 секунд, та надання цих даних для відображення на сторінці статистики.

Дана конструкція обробки даних працює на основі пулу потоків, що надає переваги у ефективному та оптимальному управлінні системними ресурсами, а також у використанні абсолютно всіх потоків, що знаходяться у пулі. Тобто у випадку, коли потік закінчив успішно обробляти конкретний запит, то він (потік) повертається назад у пул і стає доступним для залучення до обробки наступного запиту в черзі. Наступні дії повторюються згідно описаних вище дій до того часу, допоки не будуть

оброблені всі запити.

У порівнянні двох варіантів реалізації: ТРЕ та один потік, ТРЕ має надзвичайні переваги. Всі 10 потоків, що наперед задані як ємність ТРЕ у конфігурації, включаються (запускаються) одночасно і відразу беруть у роботу по одній задачі кожному. Відповідно, умовно, за першу секунду роботи ТРЕ бере в роботу вже 10 задач (об’єктів), а один потік – лише одну.

Для проведення досліджень було вирішено використовувати два засоби:
ТРЕ з ємністю 10 потоків, що працюють паралельно та незалежно одне від одного, а також реалізацію одного самостійного та незалежного потоку, котру виконано у класі com.diploma.singlethread.SingleThread .

Заміри часу було проведено за допомогою класу TimerOfPerformedWork, що, згідно запланованого розкладу, запускається кожні 10 секунд та запитує у бази даних кількість оброблених запитів, що існують на поточний момент, а потім зберігає цю кількість у об’єкті статистики для відображення на сторінці статистики.

На протязі проведення досліджень по замірюванні кількісних характеристик для наборів тестових об’єктів у кількісті: 100, 1000 та 3000 відповідно було виявлено, що при збільшенні кількості тестових об’єктів час їхньої обробки зростає для обидвох засобів, але для реалізації ОП обробки часовий інтервал збільшується надзвичайно стрімко, що зменшує можливість отримання статистики за весь інтервал.

Перше замірювання було проведено для кількості 100 тестових об’єктів. І вже на такому малому об’єму даних структура ТРЕ показала перші результати.

Отож, згідно дослідження обробки 100 об’єктів і графіків-результатів робимо висновок, що 100 об’єктів одним потоком було оброблено за 210 секунд (3 хв. 30 с), а ТРЕ обробив об’єкти за 30 с. Відповідно ТРЕ впорався із поставленою задачею в СІМ разів швидше.

Друге замірювання було проведено для кількості 1000 тестових об’єктів. Перед початком замірювання очікувалось, що так як кількість об’єктів збільшена в 10 разів, то, логічно, що і часові інтервали повинні були збільшитись у 10 разів відносно першого замірювання. Тобто очікувалось, що ТРЕ обробить дані за 300 с, а один потік – за 2100 с.

Але згідно дослідження обробки 1000 об’єктів і графіків-результатів робимо висновок, що 1000 об’єктів одним потоком було оброблено за 2040 секунд (34 хв.), а ТРЕ обробив об’єкти за 210 с (3 хв. 30 с). Відповідно ТРЕ впорався із поставленою задачею у 9,72 разів.швидше, ніж олин потік, що є дось істотним приростом у швидкості обробки даних.

В результаті згідно дослідження отримані результати перевищують очікування: 1000 об’єктів оброблено за 210 с за допомогою ТРЕ, що на 90 с швидше, ніж очікувалось.

Третє замірювання було проведено для кількості 3000 об’єктів. Результати третього вимірювання не залишаюсь сумніви у потужності такого засобу БП обробки даних як ТРЕ.

Отже, результати третього вимірювання показали що 3000 об’єктів одним потоком було оброблено за 8050 секунд (2 год. 14 хв. 10 с), а ТРЕ обробив об’єкти за 660 с (11 хв.). Відповідно: ТРЕ впорався із поставленою задачею у 12,2 разів швидше, ніж один потік.

Варто зауважити, що час обробки одним потоком з кожним вимірюванням надзвичайно збільшується, що не залишає будь-який аргументів на користь такої обробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Шилдт Г. Java.Полное руководство.8-е издание - СПб. : Питер, 2012. - 856 с.
2. Фримен Эр., Фримен Эл., Бейтс Б., Сьерра К. Паттерны проектирования - СПб. : Питер, 2011. - 102 с.
3. Joshua Bloch Effective Java (2nd Edition). - New York : Addison-Wesley Professional, 2008, - 201 p.
4. Andrew Hunt, David Thomas The Pragmatic Programmer: From Journeyman to Master - London : Addison-Wesley Professional, 1999, - 111 p.
5. Robert C. Martin Clean Code: A Handbook of Agile Software Craftsmanship - Manchester : Prentice Hall, 2008, - 36 p.
6. Martin Fowler Refactoring: Improving the Design of Existing Code - New York : Addison-Wesley Professional, 1999, - 68 p.
7. Robert C. Martin The Clean Coder: A Code of Conduct for Professional Programmers (Robert C. Martin Series) - Manchester : Prentice Hall, 2011, - 165 p.
8. Michael Feathers Working Effectively with Legacy Code - Seattle : Prentice Hall, 2004, - 406 p.
9. Tim Peierls, Brian Goetz, Joshua Bloch Java Concurrency in Practice. - New York : Addison-Wesley Professional, 2005, - 35 p.
10. Хорстманн К., Корнелл Г. Java 2 Том 1 (7-е издание) - Москва : Издательский дом "Вильямс", 2007. - 608 с.
11. Эккель Б. Философия Java (4-е издание) - СПб. : Питер, 2009. - 95 с.
12. Флэнаган Д. Java в примерах. Справочник - Москва : Символ-Плюс, 2003. - 55 с.
13. Горнаков С.Г. Программирование мобильных телефонов на Java 2 Micro Edition - Уфа : ДМК, 2001. - 145 с.
14. Томас М., Пател П., Хадсон А., Болл Д. Секреты программирования для Internet на Java - СПб. : Питер, 2002. - 163 с.
15. Романчик В.С., Блинов И.Н. Практическое руководство по изучению языка Java - Москва : Издательский дом "Вильямс", 2004. - 70 с.

16. Хабибуллин И.Ш. Самоучитель Java — СПб.: БХВ-Петербург, 2001 — 464 с.

17. Хорстманн К., Корнелл Г. Java 2 Том 2 (7-е издание) - Москва : Издательский дом "Вильямс", 2007. - 431 с.

18. Аккуратов Е.Е. Знакомьтесь: Java - Москва : Издательский дом "Вильямс", 2006. - 210 с.

19. Шилдт Г., Холмс Д. Искусство программирования на Java - СПб. : Питер, 2005. - 85 с.

20. Бондарев В.М. Учебное пособие по программированию на Java - Харьков : СМИТ, 2003. – 296 с.

21. Дунаев С.Б. Доступ к базам данных из Java-программ и проблемы русификации - Краснодар : Лира, 1999. - 152 с

22. Цишевский В. Язык и архитектура Java - СПб. : Питер, 2001. - 28 с.

23. Семихатов С. Технологии WWW, Corba и Java в построении распределенных объектных систем - Москва : Символ-Плюс, 2000. - 55 с.

24. Машнин Т.С. Web-сервисы Java - Москва : Издательский дом "Вильямс", 2012. - 105 с.

25. Седжвик Р., Уэйн К. Алгоритмы на Java - СПб. : Питер, 2013. - 659 с.

26. Васильев А.Н. Java. Объектно-ориентированное программирование - Екатеринбург : Символ, 2011. - 55 с.

27. Sonatype Company Maven: The Definitive Guide - New York : O'Reilly Media, 2008, - 154 p.

28. Srirangan S. Apache Maven 3 Cookbook - Munchen : Packt Publishing, 2011, - 68 p.

29. Wheeler W., White J. Spring in Practice - London : Manning Publications, 2013, - 392 p.

30. Pollack M., Gierke O., Risberg T., Brisbin J., Hunger M. Spring Data - New York : O'Reilly Media, 2012, - 50 p.

31. Walls C. Spring in Action - London : Manning Publications, 2011, - 156 p.

32. Bauer C., King G., Gregory G. Java Persistence with Hibernate - London :

Manning Publications, 2013, - 105 p.

33. Brittain J., Darwin I.F. Tomcat: The Definitive Guide - New York : O'Reilly Media, 2007, - 39 p.

34. Bouchrika I. Learn Database Systems with Implementation and Examples - Liverpool : lulu.com, 2014, - 73 p.

35. Keith M., Schincariol M. Pro JPA 2 - Seattle : Apress, 2013, - 331 p.

Лістинг класу Constants

```
package com.diploma.constants;

public class Constants {
    private Constants(){
        throw new AssertionError("Call isn't allowed");
    }

    public static final String USERS_TABLE = "users";
    public static final String USER_ID = "userId";
    public static final String USER_FIRST_NAME = "firstName";
    public static final String USER_LAST_NAME = "lastName";
    public static final String USER_SERIAL_NUMBER = "serialNumber";
    public static final String USER_IDENTIFICATION_NUMBER = "identificationNumber";

    public static final String REQUEST_ID = "requestId";
    public static final String REQUESTS_TABLE = "requests";

    public static final String DEPARTMENTS_TABLE = "departments";
    public static final String DEPARTMENT_ID = "departmentId";

    public static final String ACTION_ID = "actionId";
    public static final String ACTIONS_TABLE = "actions";



    public static final String ADDRESS = "address";
    public static final String PHONE = "phone";
    public static final String DESCRIPTION = "description";
}
```

Лістинг класу Action

```
package com.diploma.domain;

import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.annotations.XStreamAlias;

import javax.persistence.*;
import java.util.HashSet;
import java.util.Set;

import static com.diploma.constants.Constants.*;

@XStreamAlias("action")
@Entity
@Table(name = ACTIONS_TABLE, uniqueConstraints = {@UniqueConstraint(columnNames =
DESCRIPTION)})
public class Action {
   @XStreamAlias("actionId")
   private Long actionId;
   @XStreamAlias("description")
   private String description;
   private Set<Request> requests = new HashSet<Request>();

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(name = ACTION_ID, nullable = false)
   public Long getActionId() {
      return actionId;
   }

   public void setActionId(Long actionId) {
      this.actionId = actionId;
   }
```

```
@Column(name = DESCRIPTION, nullable = true)
public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

@OneToMany(fetch = FetchType.EAGER, mappedBy = "action")
public Set<Request> getRequests() {
    return requests;
}

public void setRequests(Set<Request> requests) {
    this.requests = requests;
}

@Override
public String toString() {
    return "Action{" +
            "actionId=" + actionId +
            ", description='" + description + '\'' +
            '}';
}

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

    Action action = (Action) o;

    if (actionId != null ? !actionId.equals(action.actionId) : action.actionId != null) return false;
    if (description != null ? !description.equals(action.description) : action.description != null)
```

```
return false;

            return true;
        }

        @Override
        public int hashCode() {
            int result = actionId != null ? actionId.hashCode() : 0;
            result = 31 * result + (description != null ? description.hashCode() : 0);
            return result;
        }

        public static Set<Action> getAvailableActionsFromXml() {
            XStream xStream = new XStream();
            xStream.autodetectAnnotations(true);
            xStream.alias("actions", HashSet.class);
            xStream.alias("action", Action.class);
            return (HashSet)
xStream.fromXML(Action.class.getClassLoader().getResource("availableActions.xml"));
        }
    }
```

Лістинг класу Department

```
package com.diploma.domain;

import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.annotations.XStreamAlias;

import javax.persistence.*;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
```

```
import static com.diploma.constants.Constants.*;

@XStreamAlias("department")
@Entity
@Table(name = DEPARTMENTS_TABLE, uniqueConstraints = {@UniqueConstraint(columnNames = DESCRIPTION)})
public class Department {
   @XStreamAlias("departmentId")
   private Long departmentId;
   @XStreamAlias("description")
   private String description;
   @XStreamAlias(("address"))
   private String address;
   @XStreamAlias("phone")
   private String phone;
   private Set<Request> requests =  new HashSet<Request>();

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(name = DEPARTMENT_ID, nullable = false)
   public Long getDepartmentId() {
      return departmentId;
   }

   public void setDepartmentId(Long departmentId) {
      this.departmentId = departmentId;
   }

   @Column(name = DESCRIPTION, nullable = true)
   public String getDescription() {
      return description;
   }

   public void setDescription(String description) {
      this.description = description;
```

```
}

@Column(name = ADDRESS, nullable = false)
public String getAddress() {
   return address;
}

public void setAddress(String address) {
   this.address = address;
}

@Column(name = PHONE, nullable = true)
public String getPhone() {
   return phone;
}

public void setPhone(String phone) {
   this.phone = phone;
}

@OneToMany(fetch = FetchType.EAGER, mappedBy = "department")
public Set<Request> getRequests() {
   return requests;
}

public void setRequests(Set<Request> requests) {
   this.requests = requests;
}

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

   Department that = (Department) o;
```

```
        if (address != null ? !address.equals(that.address) : that.address != null) return false;
        if (departmentId != null ? !departmentId.equals(that.departmentId) : that.departmentId != null) return false;
        if (description != null ? !description.equals(that.description) : that.description != null) return false;
        if (phone != null ? !phone.equals(that.phone) : that.phone != null) return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = departmentId != null ? departmentId.hashCode() : 0;
        result = 31 * result + (description != null ? description.hashCode() : 0);
        result = 31 * result + (address != null ? address.hashCode() : 0);
        result = 31 * result + (phone != null ? phone.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return "Department{" +
                ", phone='" + phone + '\'' +
                ", address='" + address + '\'' +
                ", description='" + description + '\'' +
                ", departmentId=" + departmentId +
                '}';
    }

    public static List<Department> getDepartmentsFromXml() {
        XStream xStream = new XStream();
        xStream.autodetectAnnotations(true);
        xStream.alias("departments", LinkedList.class);
        xStream.alias("department", Department.class);
```

```
            return (LinkedList)
xStream.fromXML(Department.class.getClassLoader().getResource("availableDepartments.xml"));
        }
    }
```

Лістинг класу Request

```
package com.diploma.domain;

import javax.persistence.*;
import javax.persistence.Entity;
import javax.persistence.Table;

import static com.diploma.constants.Constants.*;

@Entity
@Table(name = REQUESTS_TABLE)
public class Request {
    public static final String IN_PROGRESS = "In Progress";
    public static final String NOT_PROCESSED = "Not Processed";
    public static final String PROCESSED = "Processed";
    public static final String PROCESSED_AND_DELETED = "Processed And Deleted";

    private Long requestId;
    private Department department;
    private User user;
    private String description;
    private Action action;
    private String status = NOT_PROCESSED;
    private String interval = "";

    public String getInterval() {
        return interval;
    }
```

```
public void setInterval(String interval) {
   this.interval = interval;
}

public String getStatus() {
   return status;
}

public void setStatus(String status) {
   this.status = status;
}

@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = ACTION_ID)
public Action getAction() {
   return action;
}

public void setAction(Action action) {
   this.action = action;
}

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = REQUEST_ID, nullable = false, length = 5)
public Long getRequestId() {
   return requestId;
}

public void setRequestId(Long requestId) {
   this.requestId = requestId;
}

@ManyToOne(fetch = FetchType.EAGER)
```

```
@JoinColumn(name = DEPARTMENT_ID)
public Department getDepartment() {
   return department;
}

public void setDepartment(Department department) {
   this.department = department;
}

@Column(name = DESCRIPTION, nullable = true, length = 50)
public String getDescription() {
   return description;
}

public void setDescription(String description) {
   this.description = description;
}

@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = USER_ID)
public User getUser() {
   return user;
}

public void setUser(User user) {
   this.user = user;
}

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

   Request request = (Request) o;
```

```
            if (department != null ? !department.equals(request.department) : request.department != null)
return false;
            if (description != null ? !description.equals(request.description) : request.description != null)
return false;
            if (user != null ? !user.equals(request.user) : request.user != null) return false;

            return true;
        }

        @Override
        public int hashCode() {
            int result = department != null ? department.hashCode() : 0;
            result = 31 * result + (user != null ? user.hashCode() : 0);
            result = 31 * result + (description != null ? description.hashCode() : 0);
            return result;
        }

        @Override
        public String toString() {
            return "Request{" +
                    "requestId=" + requestId +
                    ", department=" + department +
                    ", user=" + user +
                    '}';
        }
    }
```

Лістинг класу User

```
package com.diploma.domain;

import com.diploma.constants.Constants;

import javax.persistence.*;
import java.util.HashSet;
```

```
import java.util.Set;

import static com.diploma.constants.Constants.*;

@Entity
@Table(name = USERS_TABLE,
    uniqueConstraints = {@UniqueConstraint(columnNames = USER_IDENTIFICATION_NUMBER)})
public class User {
   private Long userId;
   private String firstName;
   private String lastName;
   private String phone;
   private String address;
   private String serialNumber;
   private Long identificationNumber;
   private Set<Request> requests = new HashSet<Request>();

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(name = USER_ID, nullable = false)
   public Long getUserId() {
      return userId;
   }

   public void setUserId(Long userId) {
      this.userId = userId;
   }

   @Column(name = USER_FIRST_NAME, nullable = false)
   public String getFirstName() {
      return firstName;
   }

   public void setFirstName(String firstName) {
```

```
    this.firstName = firstName;
}

@Column(name = USER_LAST_NAME, nullable = false)
public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

@Column(name = PHONE, nullable = true)
public String getPhone() {
    return phone;
}

public void setPhone(String phone) {
    this.phone = phone;
}

@Column(name = ADDRESS, nullable = false)
public String getAddress() {
    return address;
}

public void setAddress(String address) {
    this.address = address;
}

@Column(name = USER_SERIAL_NUMBER, nullable = false)
public String getSerialNumber() {
    return serialNumber;
}
```

```
public void setSerialNumber(String serialNumber) {
    this.serialNumber = serialNumber;
}

@Column(name = USER_IDENTIFICATION_NUMBER, nullable = false, unique = true)
public Long getIdentificationNumber() {
    return identificationNumber;
}

public void setIdentificationNumber(Long identificationNumber) {
    this.identificationNumber = identificationNumber;
}

@OneToMany(fetch = FetchType.EAGER, mappedBy = "user")
public Set<Request> getRequests() {
    return requests;
}

public void setRequests(Set<Request> requests) {
    this.requests = requests;
}

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

    User user = (User) o;

    if (!address.equals(user.address)) return false;
    if (!firstName.equals(user.firstName)) return false;
    if (!identificationNumber.equals(user.identificationNumber)) return false;
    if (!lastName.equals(user.lastName)) return false;
    if (!phone.equals(user.phone)) return false;
    if (!serialNumber.equals(user.serialNumber)) return false;
```

```
        return true;
    }

    @Override
    public int hashCode() {
        int result = firstName.hashCode();
        result = 31 * result + lastName.hashCode();
        result = 31 * result + phone.hashCode();
        result = 31 * result + address.hashCode();
        result = 31 * result + serialNumber.hashCode();
        result = 31 * result + identificationNumber.hashCode();
        return result;
    }

    @Override
    public String toString() {
        return "User{" +
                "identificationNumber=" + identificationNumber +
                ", serialNumber='" + serialNumber + '\'' +
                ", address='" + address + '\'' +
                ", phone='" + phone + '\'' +
                ", lastName='" + lastName + '\'' +
                ", firstName='" + firstName + '\'' +
                ", userId=" + userId +
                '}';
    }
}
```

Лістинг класу WizardFlowComponent

```
package com.diploma.mvc.wizard;

import com.diploma.domain.Action;
import com.diploma.domain.Department;
```

```
import com.diploma.domain.Request;
import com.diploma.domain.User;
import com.diploma.repository.ActionRepository;
import com.diploma.repository.DepartmentRepository;
import com.diploma.repository.RequestRepository;
import com.diploma.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.validation.Errors;

import java.util.*;

@Component
public class WizardFlowComponent {
    @Autowired
    private DepartmentRepository departmentRepository;
    @Autowired
    private ActionRepository actionRepository;
    @Autowired
    private RequestRepository requestRepository;
    @Autowired
    private UserRepository userRepository;

    private static List<Long> idNumbers = new LinkedList<Long>();
    private static List<String> firstNames = new LinkedList<String>();
    private static List<String> lastNames = new LinkedList<String>();
    private static List<String> phones = new LinkedList<String>();
    private static List<String> addresses = new LinkedList<String>();
    private static List<String> passports = new LinkedList<String>();
    private static List<String> reasons = new LinkedList<String>();

    static {
        firstNames.add("Петро");
        firstNames.add("Василь");
```

```
firstNames.add("Георгій");
firstNames.add("Степан");
firstNames.add("Ігор");
firstNames.add("Сергій");
firstNames.add("Андрій");
firstNames.add("Дмитро");
firstNames.add("Федір");
Collections.shuffle(firstNames);

lastNames.add("Олійник");
lastNames.add("Щур");
lastNames.add("Огойко");
lastNames.add("Камінський");
lastNames.add("Кузьмич");
lastNames.add("Левчук");
lastNames.add("Порядинський");
lastNames.add("Волков");
lastNames.add("Якубов");
Collections.shuffle(lastNames);

phones.add("+38(063)751-31-62");
phones.add("+38(097)995-75-69");
phones.add("+38(098)234-00-00");
phones.add("+38(095)960-78-24");
phones.add("+38(067)525-25-25");
phones.add("+38(093)431-18-25");
phones.add("+38(098)174-27-65");
phones.add("+38(044)247-74-82");
phones.add("+38(099)936-21-53");
Collections.shuffle(phones);

addresses.add("04367, Жмеринка, вул. Черняхвіського, 15");
addresses.add("05398, Овруч, вул. Богдана Хмельницького, 26, кв.5");
addresses.add("06146, Одеса, вул. Раїси Окіпної, 214, кв.116");
addresses.add("05599, Львів, вул. Медова, 18, кв.49");
```

```
        addresses.add("03662, Київ, вул. Якуба Коласа, 28, кв.2");
        addresses.add("01746, Вінниця, вул. Тернопільська, 87, кв.36");
        addresses.add("02913, Фастів, вул. Залізнична, 9, кв.45");
        addresses.add("02764, Ужгород, вул. Річна, 8");
        addresses.add("03793, Харків, вул. Комісара Рикова, 129, кв.17");
        Collections.shuffle(addresses);

        passports.add("ВХ402961");
        passports.add("УЖ021327");
        passports.add("АО784512");
        passports.add("РО240568");
        passports.add("ЕП796580");
        passports.add("НЕ821346");
        passports.add("ДК785632");
        passports.add("КК224584");
        passports.add("ІЯ753249");
        Collections.shuffle(passports);

        idNumbers.add(1234567892L);
        idNumbers.add(7834965270L);
        idNumbers.add(2533300981L);
        idNumbers.add(9335271800L);
        idNumbers.add(1000854546L);
        idNumbers.add(2846167861L);
        idNumbers.add(6542132288L);
        idNumbers.add(2669822271L);
        idNumbers.add(3053259584L);
        Collections.shuffle(idNumbers);

        reasons.add("Первинне оформлення документів");
        reasons.add("Запит інформації");
        reasons.add("Зміна документів");
        reasons.add("Подача документів");
        Collections.shuffle(reasons);
    }
```

```
public void registerUser(Request userRequest, User user, Errors errors) {
    userRequest.setUser(user);
}

public void chooseDepartment(Long departmentId, Request userRequest) {
    if (departmentId != null && userRequest != null) {
        Department currentDepartment = departmentRepository.findOne(departmentId);
        userRequest.setDepartment(currentDepartment);
    }
}

@Transactional
public void init() {
    if (actionRepository.count() == 0 || departmentRepository.count() == 0) {
        Set<Action> actions = Action.getAvailableActionsFromXml();
        List<Department> departments = Department.getDepartmentsFromXml();
        actionRepository.save(actions);
        departmentRepository.save(departments);
    }
}

public Map<Long, String> defineActionMap() {
    List<Action> actions = actionRepository.findAll();
    Map<Long, String> actionMap = new LinkedHashMap<Long, String>(actions.size());
    for (Action action : actions) {
        actionMap.put(action.getActionId(), action.getDescription());
    }
    return actionMap;
}

public void saveSelectedActionInRequest(Request userRequest, long selectedActionId) {
    Action selectedAction = actionRepository.findOne(selectedActionId);
    if (selectedAction != null) {
        userRequest.setAction(selectedAction);
```

```
    }
}

public void saveRequestOnFinish(Request userRequest) {
    departmentRepository.save(userRequest.getDepartment());
    userRepository.save(userRequest.getUser());
    requestRepository.save(userRequest);
    actionRepository.save(userRequest.getAction());
}

public DepartmentRepository getDepartmentRepository() {
    return departmentRepository;
}

public ActionRepository getActionRepository() {
    return actionRepository;
}

public RequestRepository getRequestRepository() {
    return requestRepository;
}

public UserRepository getUserRepository() {
    return userRepository;
}

public void fillUserByTestData(Request userRequest) {
    User user = new User();
    long LOWER_RANGE = 1000000000L; //assign lower range value
    long UPPER_RANGE = 9999999999L; //assign upper range value
    Random random = new Random();

    long randomValue = LOWER_RANGE +
            (long)(random.nextDouble()*(UPPER_RANGE - LOWER_RANGE));
    user.setAddress(addresses.get(random.nextInt(9)));
```

```
    user.setFirstName(firstNames.get(random.nextInt(9)));
    user.setLastName(lastNames.get(random.nextInt(9)));
    user.setIdentificationNumber(randomValue);
    user.setPhone(phones.get(random.nextInt(9)));
    user.setSerialNumber(passports.get(random.nextInt(9)));
    userRequest.setUser(user);
}

private void fillDepartmentByTestData(Request userRequest, List<Department> departments) {
    Random random = new Random();
    if (departments != null && departments.size() > 0) {
        int size = departments.size();
        userRequest.setDepartment(departments.get(random.nextInt(size)));
    }
}

private void fillActionByTestData(Request userRequest, List<Action> actions) {
    Random random = new Random();
    if (actions != null && actions.size() > 0) {
        int size = actions.size();
        userRequest.setAction(actions.get(random.nextInt(size)));
    }
}

private void fillReasonsByTestData(Request userRequest) {
    Random random = new Random();
    userRequest.setDescription(reasons.get(random.nextInt(reasons.size())));
}

public void fillRequestByTestData(int count) {
    List<Action> actions = actionRepository.findAll();
    List<Department> departments = departmentRepository.findAll();
    List<User> usersIns = new LinkedList<User>();
    List<Request> requestsIns = new LinkedList<Request>();
    List<Department> departmentsIns = new LinkedList<Department>();
```

```
        List<Action> actionsIns = new LinkedList<Action>();
        for (int i = 0; i < count; i++) {
            Request userRequest = new Request();
            fillUserByTestData(userRequest);
            fillActionByTestData(userRequest, actions);
            fillDepartmentByTestData(userRequest, departments);
            fillReasonsByTestData(userRequest);
            usersIns.add(userRequest.getUser());
            requestsIns.add(userRequest);
            departmentsIns.add(userRequest.getDepartment());
            actionsIns.add(userRequest.getAction());
        }
        userRepository.save(usersIns);
        requestRepository.save(requestsIns);
        departmentRepository.save(departmentsIns);
        actionRepository.save(actionsIns);
    }
}
```

Лістинг класу WizardFormController

```
package com.diploma.mvc.wizard;

import com.diploma.domain.Department;
import com.diploma.domain.Request;
import com.diploma.domain.User;
import com.diploma.schedule.TimerOfPerformedWork;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
```

```
import org.springframework.web.bind.annotation.SessionAttributes;

import java.util.List;
import java.util.concurrent.TimeUnit;

@Controller
@RequestMapping("/register")
@SessionAttributes({"userRequest", "user", "department"})
public class WizardFormController {

   @Autowired
   private WizardFlowComponent flowComponent;

   private static final String STEP = "step";
   private static final int INITIAL_STEP = 0;
   private static final int DEPARTMENT_CHOICE_STEP = 1;
   private static final int USER_REGISTRATION__STEP = 2;
   private static final int REQUEST_DESCRIPTION_AND_ACTION_CHOICE_STEP = 3;
   private static final int FINAl_STEP = 4;

   @RequestMapping(method = RequestMethod.POST, params = "_page")
   public String processPage(@RequestParam("_page") int page,
                             @ModelAttribute("userRequest") Request userRequest,
                             @ModelAttribute("user") User user,
                             @ModelAttribute("department") Department department,
                             @RequestParam(value = "departmentId", required = false) Long departmentId,
                             @RequestParam(value = "test", required = false) boolean fillTestData,
                             @RequestParam(value = "muchData", required = false) boolean muchData,
                             @RequestParam(value = "countOfData", required = false) Integer
countOfData,
                             final Errors errors,
                             final ModelMap model) {
      switch (page) {
         case DEPARTMENT_CHOICE_STEP:
            flowComponent.init();
```

```
                model.addAttribute("departments",
flowComponent.getDepartmentRepository().findAll());
                break;
            case USER_REGISTRATION__STEP:
                if (muchData && countOfData > 0) {
                    flowComponent.fillRequestByTestData(countOfData);
                    model.addAttribute("successfulMessage", countOfData + " тестових об'єктів дода-
но");
                    return getInitialPage(model);
                } else {
                    flowComponent.chooseDepartment(departmentId, userRequest);
                    model.addAttribute("department", userRequest.getDepartment());
                }
                break;
            case REQUEST_DESCRIPTION_AND_ACTION_CHOICE_STEP:
                if (fillTestData) {
                    flowComponent.fillUserByTestData(userRequest);
                    model.addAttribute("user", userRequest.getUser());
                    page = USER_REGISTRATION__STEP;
                } else {
                    flowComponent.registerUser(userRequest, user, errors);
                    model.addAttribute("actionMap", flowComponent.defineActionMap());
                }
                break;
        }
        model.addAttribute(userRequest);

        return returnPage(page, errors);
    }

    private String returnPage(int page, Errors errors) {
        if (errors.hasErrors()) {
            return STEP + (page - 1);
        } else {
            if (page < 1) {
```

```
            return STEP + INITIAL_STEP;
        } else {
            return STEP + page;
        }
    }
}

@RequestMapping(method = RequestMethod.GET)
public String getInitialPage(final ModelMap model) {
    model.addAttribute("userRequest", new Request());
    model.addAttribute("user", new User());
    model.addAttribute("department", new Department());
    return STEP + INITIAL_STEP;
}

@RequestMapping(params = "cancel", method = RequestMethod.POST)
public String getCancelPage(@ModelAttribute("userRequest") Request userRequest,
                            @ModelAttribute("user") User user,
                            final Errors errors,
                            final ModelMap model) {
    return getInitialPage(model);
}

@RequestMapping(method = RequestMethod.POST, params = "finish")
public String finish(@ModelAttribute("userRequest") Request userRequest,
                     @ModelAttribute("user") User user,
                     @RequestParam("actionId") Long selectedActionId,
                     final Errors errors,
                     final ModelMap model) {
    flowComponent.saveSelectedActionInRequest(userRequest, selectedActionId);
    flowComponent.saveRequestOnFinish(userRequest);
    model.addAttribute(userRequest);
    return STEP + FINAl_STEP;
}
```

```
    @RequestMapping(method = RequestMethod.GET, value = "/stats")
    public String getStatisticsPage(final ModelMap model) {
        List<Request> requests = flowComponent.getRequestRepository().findAll();
        model.addAttribute("requests", requests);
        model.addAttribute("statisticsMap", TimerOfPerformedWork.getStatistics());
        return "stats";
    }

    @RequestMapping(method = RequestMethod.POST, value = "/stats")
    public String deletePerformedEntries(final ModelMap model) {
        List<Request> performedRequests =
flowComponent.getRequestRepository().findAllProcessedTasks();
        for (Request performedRequest : performedRequests) {
            performedRequest.setStatus(Request.PROCESSED_AND_DELETED);
        }
        flowComponent.getRequestRepository().save(performedRequests);
        return getStatisticsPage(model);
    }
}
```

Лістинг класу ActionRepository

```
package com.diploma.repository;

import com.diploma.domain.Action;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ActionRepository extends JpaRepository<Action, Long> {
}
```

Лістинг класу DepartmentRepository

```
package com.diploma.repository;
```

```
import com.diploma.domain.Department;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.orm.jpa.JpaTransactionManager;

public interface DepartmentRepository extends JpaRepository<Department, Long>{

}
```

Лістинг класу RequestRepository

```
package com.diploma.repository;

import com.diploma.domain.Request;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

public interface RequestRepository extends JpaRepository<Request, Long> {
    @Transactional
    @Query("select a from Request a where a.status = ''' + Request.IN_PROGRESS + ''')
    public List<Request> findAllRequestsInProgress();

    @Transactional
    @Query("select a from Request a where a.status = ''' + Request.NOT_PROCESSED + ''')
    public List<Request> findAllNotProcessedRequests();

    @Transactional
    @Query("select a from Request a where a.status = ''' + Request.PROCESSED + ''')
    public List<Request> findAllProcessedTasks();
```

```
    @Transactional
    @Query("select a from Request a where a.status = '" + Request.PROCESSED + "' or " +
            "a.status = '" + Request.PROCESSED_AND_DELETED + "'")
    public List<Request> findAllProcessedAndDeletedTasks();
}
```

Лістинг класу UserRepository

```
package com.diploma.repository;

import com.diploma.domain.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long>{
}
```

Лістинг коду TaskScheduler

```
package com.diploma.schedule;

import com.diploma.domain.Request;
import com.diploma.mvc.wizard.WizardFlowComponent;
import com.diploma.repository.RequestRepository;
import com.diploma.task.WizardTask;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;

import java.util.List;

public class TaskScheduler {

    private ThreadPoolTaskExecutor taskExecutor;

    private WizardFlowComponent flowComponent;
```

```
    public ThreadPoolTaskExecutor getTaskExecutor() {
        return taskExecutor;
    }

    public void setTaskExecutor(ThreadPoolTaskExecutor taskExecutor) {
        this.taskExecutor = taskExecutor;
    }

    public WizardFlowComponent getFlowComponent() {
        return flowComponent;
    }

    public void setFlowComponent(WizardFlowComponent flowComponent) {
        this.flowComponent = flowComponent;
    }

    public void execute() {
        RequestRepository requestRepository = flowComponent.getRequestRepository();
        List<Request> requests = requestRepository.findAllNotProcessedRequests();
        for (Request request : requests) {
            request.setStatus(Request.IN_PROGRESS);
        }
        requestRepository.save(requests);
        for (Request request : requests) {
            taskExecutor.execute(new WizardTask(request, requestRepository));
        }
    }
}
```

Лістинг класу TimerOfPerformedWork

```
package com.diploma.schedule;

import com.diploma.domain.Request;
```

```
import com.diploma.mvc.wizard.WizardFlowComponent;
import com.diploma.repository.RequestRepository;
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

public class TimerOfPerformedWork {
    private static Map<String, Integer> statistics = new LinkedHashMap<String, Integer>();
    private static final DateTimeFormatter DATE_TIME_FORMATTER =
DateTimeFormat.forPattern("HH:mm:ss");

    private WizardFlowComponent flowComponent;

    public void setFlowComponent(WizardFlowComponent flowComponent) {
        this.flowComponent = flowComponent;
    }

    public WizardFlowComponent getFlowComponent() {
        return flowComponent;
    }

    public void execute() {
        RequestRepository requestRepository = flowComponent.getRequestRepository();
        List<Request> requests = requestRepository.findAllProcessedAndDeletedTasks();
        int countItemsOnFinish = 0;
        if (requests != null && requests.size() != 0) {
            countItemsOnFinish = requests.size();
        }
        DateTime now = DateTime.now();

        statistics.put(now.toString(DATE_TIME_FORMATTER), countItemsOnFinish);
```

```
    }

    public static Map<String, Integer> getStatistics() {
        return statistics;
    }
}
```

Лістинг класу SingleThread

```
package com.diploma.singlethread;

import com.diploma.domain.Request;
import com.diploma.repository.RequestRepository;
import com.diploma.task.WizardTask;

import java.util.List;

public class SingleThread extends Thread {

    private RequestRepository requestRepository;

    public SingleThread(RequestRepository requestRepository) {
        this.requestRepository = requestRepository;
    }

    @Override
    public void run() {
        List<Request> requests = requestRepository.findAllRequestsInProgress();
        for (Request request : requests) {
            (new WizardTask(request, requestRepository)).run();
        }
    }
}
```

Лістинг класу WizardTask

```
package com.diploma.task;

import com.diploma.domain.Request;
import com.diploma.repository.RequestRepository;

import java.util.concurrent.TimeUnit;

public class WizardTask implements Runnable {
    private Request request;
    private RequestRepository requestRepository;

    public WizardTask(Request request, RequestRepository requestRepository) {
        this.request = request;
        this.requestRepository = requestRepository;
    }

    @Override
    public void run() {
        long startTime = System.currentTimeMillis();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        request.setStatus(Request.PROCESSED);
        long totalTime = System.currentTimeMillis() - startTime;
        request.setInterval(String.format("%d хв %d с",
                TimeUnit.MILLISECONDS.toMinutes(totalTime),
                TimeUnit.MILLISECONDS.toSeconds(totalTime))
        );
        requestRepository.save(request);
    }
}
```

Лістинг Spring configuration file

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/data/jpa
http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-
mvc.xsd http://www.springframework.org/schema/task
http://www.springframework.org/schema/task/spring-task.xsd">

    <context:component-scan base-package="com.diploma.*"/>

    <mvc:annotation-driven/>
    <mvc:resources mapping="/bootstrap/css/**" location="/bootstrap/css/"/>
    <mvc:resources mapping="/scripts/**" location="/scripts/"/>
    <mvc:resources mapping="/bootstrap/fonts/**" location="/bootstrap/fonts/"/>
    <mvc:resources mapping="/bootstrap/js/**" location="/bootstrap/js/"/>

    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <property name="prefix" value="/WEB-INF/pages/"/>
      <property name="suffix" value=".jsp"/>
    </bean>

    <jpa:repositories base-package="com.diploma.repository"/>
```

```
        <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
            <property name="persistenceUnitName" value="defaultPersistenceUnit"/>
        </bean>

        <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
            <property name="entityManagerFactory" ref="entityManagerFactory"/>
        </bean>

        <tx:annotation-driven transaction-manager="transactionManager"/>

        <bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
            <property name="basename" value="classpath:bundle/messages"/>
            <property name="defaultEncoding" value="windows-1251"/>
        </bean>

        <context:property-placeholder location="classpath:bundle/messages.properties" ignore-
resource-not-found="true" ignore-unresolvable="true"/>

        <task:annotation-driven/>

        <bean id="taskExecutor"
class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
            <property name="corePoolSize" value="10"/>
            <property name="waitForTasksToCompleteOnShutdown" value="true"/>
            <property name="maxPoolSize" value="10"/>
        </bean>

        <bean id="taskScheduler" class="com.diploma.schedule.TaskScheduler">
            <property name="taskExecutor" ref="taskExecutor"/>
            <property name="flowComponent" ref="wizardFlowComponent"/>
        </bean>
```

```
    <bean id="timer" class="com.diploma.schedule.TimerOfPerformedWork">
        <property name="flowComponent" ref="wizardFlowComponent"/>
    </bean>

    <task:scheduled-tasks>
        <task:scheduled ref="taskScheduler" method="execute" fixed-rate="10000"/>
        <task:scheduled ref="timer" method="execute" fixed-rate="10000" />
    </task:scheduled-tasks>

</beans>
```