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

РОЗРОБКА ФРЕЙМВОРКУ ДЛЯ ЗАСТОСУВАННЯ ПАТЕРНУ «EVENT SOURCING» НА МОВІ PYTHON

Текстова частина
магістерської роботи
за спеціальністю «Інженерія програмного забезпечення» - 121

Керівник магістерської роботи:
д.т.н., професор А.М. Глибовець
____
(підпис)
"____" ________ 2023р.

Виконав:
студент І.С. Янкін
____
(підпис)
"____" ________ 2023 р.

Київ-2023

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

ЗАТВЕРДЖУЮ
Зав. Кафедри інформатики, к.ф.-м.н.
______________ С.С. Гороховський
(підпис)
“____” __________ 2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

Студенту 2 р.н. магістерської програми «Інженерія програмного забезпечення» Янкіну Ігорю Сергійовичу

Розробити Фреймворк для застосування патерну «Event Sourcing» на мові Python

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1. Огляд патерну «Event Sourcing» та засобів його застосування
2. Розробка архітектури фреймворку для мови Python
3. Реалізація фреймворку на основі запропонованої архітектури
Висновки
Список літератури
Додатки

Дата видачі “____” ____________ 2023 р.
Керівник
А.М. Глибовець,
Доктор технічних наук, професор

____
(підпис)

Завдання отримав
І. С. Янкін

____
(підпис)

Тема: Розробка фреймворку для застосування патерну «Event Sourcing» на мові Python

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання завдання на магістерську роботу	25.09.2023	
2.	Пошук інформаційних джерел за темою роботи	08.10.2023	
3.	Аналіз існуючих рішень	30.10.2023	
4.	Розробка архітектури фреймворку	11.01.2024	
5.	Реалізація фреймворку	16.04.2024	
6.	Створення слайдів для доповіді та написання доповіді	07.05.2024	
7.	Аналіз отриманих реузльтатів з керівником, написання доповіді та попередній захист магістерської роботи	13.05.2024	
8.	Коригування роботи за результатами попереднього захисту	22.05.2024	
9.	Остаточне оформлення пояснювальної записки та слайдів	03.06.2024	
10.	Захист магістерської роботи	-	

Студент Янкін І.С.
Керівник Глибовець А.М.
“___” ______________ 2023.

ЗМІСТ

АНОТАЦІЯ **5**
ВСТУП **6**
1 ОГЛЯД ПАТЕРНУ «EVENT SOURCING» ТА ЗАСОБІВ ЙОГО ЗАСТОСУВАННЯ **9**
1.1 Опис та визначення патерну **9**
1.2 Основи та принципи роботи **12**
1.3 Переваги та недоліки **17**
1.4 Застосування **20**
1.5 Існуючі інструменти та технології **21**
2 РОЗРОБКА АРХІТЕКТУРИ ФРЕЙМВОРКУ ДЛЯ МОВИ PYTHON **24**
2.1 Вступ до архітектури фреймворку **24**
2.2 Вимоги до фреймворку **25**
2.3 Принципи проектування фреймворку **28**
2.4 Компоненти та структура фреймворку **30**
2.5 Тестування архітектури **34**
2.5 Документація і розповсюдження **35**
3 РЕАЛІЗАЦІЯ ФРЕЙМВОРКУ НА ОСНОВІ ЗАПРОПОНОВАНОЇ АРХІТЕКТУРИ **38**
3.1 Опис реалізації та документації **38**
3.2 Порівняння з існуючими рішеннями **44**
3.3 Майбутні удосконалення та дослідницькі напрямки **51**
ВИСНОВКИ **53**
Список використаної літератури **55**
Додаток А **59**
Додаток Б **59**
Додаток В **60**
Додаток Г **61**
Додаток Ґ **62**
Додаток Д **63**
Додаток Е **64**
Додаток Є **65**

АНОТАЦІЯ

Ця магістерська робота присвячена розробці фреймворку для мови Python, який дозволяє застосувати патерн «Event Sourcing», пропонуючи детальний аналіз теоретичних основ цього патерну та практичну реалізацію. В роботі розглядаються ключові аспекти «Event Sourcing», його переваги та недоліки, а також аналізується застосування цього патерну для вирішення різних задач. Окремо увага приділяється розробці архітектури фреймворку, його ключовим компонентам, порівнянню з існуючими рішеннями. Результатом роботи є комплексний фреймворк, спроектований для ефективної роботи з подіями в програмних проектах, що дозволяє значно підвищити якість та надійність розроблюваного програмного забезпечення.

Ключові слова: Event Sourcing, Python, архітектура фреймворку, патерни програмування, асинхронність

ВСТУП

Актуальність. Програмування систем, які потребують надійного збереження історії змін та забезпечення високої відмовостійкості, не може обійтися без ефективних підходів та архітектурних рішень. Патерн «Event Sourcing» дозволяє зберігати всю історію змін стану додатків у вигляді подій, що відкриває нові можливості для розробки програмного забезпечення. Застосування цього патерну у мові Python, яка залишається однією з найпопулярніших мов програмування станом на 2023 рік [1], стає актуальним завдяки потребі у впровадженні сучасних архітектурних практик і технологій. Особливо важливим стає створення спеціалізованого фреймворку, який враховує специфіку «Event Sourcing» і підвищує ефективність та надійність розроблюваних систем. У даній роботі описано створення такого фреймворку, який дозволяє оптимізувати процес розробки та покращити подальшу підтримку програмних проектів.

Мета дослідження. Мета даного дослідження полягає у всебічному аналізі існуючих рішень, які імплементують патерн «Event Sourcing», та вивченні сучасних технологій, що використовуються для його реалізації у програмуванні. Основний акцент дослідження робиться на виявленні недоліків і недорозвинутих областей у використанні цього патерну. Через це дослідження ми прагнемо знайти і запропонувати рішення, яке полегшить використання «Event Sourcing», забезпечуючи більш ефективне впровадження цього патерну в розробці програмного забезпечення.

Завдання дослідження. Завданням цього дослідження є розробка фреймворку, який спростить впровадження патерну «Event Sourcing» у програмні системи, розроблені на мові Python. В рамках дослідження передбачається створення архітектури фреймворку, яка дозволить ефективно інтегрувати і управляти подіями в системах, забезпечуючи легке додавання, відстеження та відновлення подій в історії змін даних. Фреймворк буде спрямований на зменшення складності технічної реалізації, підвищення швидкості розробки та забезпечення високої адаптивності до різноманітних

вимог проектів. Таким чином, основне завдання полягає не лише у розробці технічних рішень, а й у створенні універсального інструменту, що зможе задовольнити потреби широкого кола розробників та компаній, зацікавлених у підвищенні ефективності та надійності своїх програмних продуктів.

Об´єкт дослідження. Об'єктом дослідження у цій магістерській роботі є патерн «Event Sourcing» як методологія збереження стану системи шляхом фіксації послідовності подій, які змінюють цей стан. Цей патерн розглядається у контексті його впровадження в програмні системи на мові Python, з акцентом на легкості інтеграції та зручності впровадження патерну. Основна увага приділяється аналізу існуючих інструментів, технік і фреймворків, які підтримують реалізацію цього патерну, а також розробці нового рішення, яке поліпшить і спростить використання «Event Sourcing» у широкому спектрі програмних проектів.

Предмет дослідження. Предметом є розробка фреймворку для мови Python, який уможливлює впровадження патерну «Event Sourcing». Дослідження зосереджується на аналізі технічних аспектів і методів імплементації патерну в програмних проектах, зокрема, на створенні архітектурних рішень, які дозволяють ефективно управляти подіями і забезпечують повноту збереження історії змін. Особлива увага приділяється вирішенню проблем, пов'язаних із взаємодією з іншими компонентами системи. Розроблений фреймворк має на меті забезпечити розробників необхідними інструментами для легшого та більш надійного застосування «Event Sourcing» у їхніх проектах.

Джерела дослідження. Джерелами дослідження для цієї магістерської роботи служать наукові статті та книги в електронному форматі, які присвячені архітектурним патернам, системам заснованим на подіях, та відмовостійким системам. Ці матеріали дозволяють зрозуміти теоретичні основи та практичні застосування патерну «Event Sourcing». Також, значну роль у дослідженні відіграють спеціалізовані веб-сайти та форуми, де розробники обговорюють новітні розробки, виклики, інновації, а також діляться власним досвідом реалізації різноманітних підходів в програмуванні. Крім того, важливим

джерелом інформації є технічна документація на існуючі фреймворки та бібліотеки, які підтримують реалізацію «Event Sourcing», а також блоги від провідних розробників, які спеціалізуються на створенні високонадійних і масштабованих систем. Це забезпечує комплексний огляд сучасних методів та технологій, необхідних для успішного впровадження досліджуваного патерну в розробку програмного забезпечення.

Наукова новизна одержаних результатів. Наукова новизна одержаних результатів у даному дослідженні полягає у розробці нового фреймворку для мови Python, спеціалізованого на ефективному впровадженні патерну «Event Sourcing». Цей фреймворк розроблено з урахуванням недоліків та проблем у існуючих рішеннях. Розробка забезпечує спрощення процесу використання Event Sourcing, відповідаючи на виклики, з якими стикаються розробники при імплементації цього патерну, зокрема, складності в адаптації патерну до специфічних вимог проектів. Новий фреймворк значно підвищує якість та надійність розроблюваного програмного забезпечення, відкриваючи нові можливості для його застосування в індустрії інформаційних технологій.

Практичне значення одержаних результатів. Практичне значення одержаних результатів дослідження полягає у створенні фреймворку для мови Python, який спрощує інтеграцію патерну «Event Sourcing» у системи програмного забезпечення, а також його комбінацію з суміжними патернами, такими як «Domain-Driven Design» (DDD), «Event-Driven Architecture» (EDA), та «Command Query Responsibility Segregation» (CQRS), що дозволяє підвищити ефективність розробки складних систем. Фреймворк забезпечує розробникам потужні інструменти для створення масштабованих, відмовостійких та легко підтримуваних програмних рішень. Таке рішення дозволяє значно спростити впровадження цих архітектурних підходів у проекти, які вимагають високої точності відтворення подій та забезпечення цілісності даних, відкриваючи нові можливості для інновацій у галузі програмування.

1 ОГЛЯД ПАТЕРНУ «EVENT SOURCING» ТА ЗАСОБІВ ЙОГО ЗАСТОСУВАННЯ

1.1 Опис та визначення патерну

Станом на 2024 рік спостерігається тенденція до збільшення об'ємів даних, якими оперує програмне забезпечення [2]. Це обумовлено одразу низкою факторів, у числі яких є збільшення генерації даних а також зростання потреби в зборі та аналізі даних для прийняття рішень в різних галузях. Разом з такою тенденцією приходять не тільки переваги та можливості, але і необхідність контролювати і зберігати ці дані для подальшої роботи з ними.

Через це, дані і можливість вільно працювати з ними стають важливою задачею для компаній і розробників, що прагнуть покращити свої продукти. Одним з напрямків роботи є покращення можливостей аудиту даних, а саме можливість відновити стан системи на будь-який момент часу. Це може як спрощувати розробку і процес пошуку помилок, так і бути необхідним критерієм для відповідності регуляторним вимогам. Так, ще у 2013 році був описаний зсув від саморегуляції аудитів до регуляції, керованої організаціями, який триває і досі [3].

Саме вирішення цієї проблеми і пропонує Мартін Фаулер у своїй статті під назвою «Event Sourcing» де описується концепт однойменного патерну а також наводиться приклад реалізації [4].

В абстрактному розумінні патерн являє собою спосіб взаємодії зі сховищем даним, що дозволяє зберігати та відновлювати дані про стан системи в певний момент часу. Це досягається за допомогою того, що будь-яка зміна стану системи зберігається у вигляді атомарної і незмінної події, а сам стан реконструюється за допомогою застосування подій в тій черзі в якій вони відбувались.

Хоча цей патерн і може бути використано окремо від усіх інших, але згідно зі спостереженнями експертів, більшість розробників надають перевагу його

інтеграції з іншими патернами та підходами до дизайну систем [5]. Це пояснюється тим, що поєднання «Event Sourcing» з «Domain-Driven Design», «Event-Driven Architecture», та «Command Query Responsibility Segregation» дозволяє досягти синергії, що значно покращує розробку, підтримку та управління складними системами, роблячи їх більш гнучкими та масштабованими.

«Domain-Driven Design» є підходом до розробки програмного забезпечення, який зосереджується на моделюванні домену бізнесу та його логіки. «Event Sourcing» природньо інтегрується з DDD, оскільки дозволяє ефективно зафіксувати всі зміни в доменних об'єктах як послідовність подій. Ілюстрація (рис. 1.1.1) демонструє приклад такої інтеграції, де агрегатний корінь на запит користувача генерує певні зміни у вигляді подій, що потім передаються системі керування подіями, яка зберігає їх. Це не тільки сприяє кращому розумінню бізнес-процесів та їх еволюції в часі, але й уможливлює точне відтворення стану домену на будь-який момент часу, що є важливим для DDD.

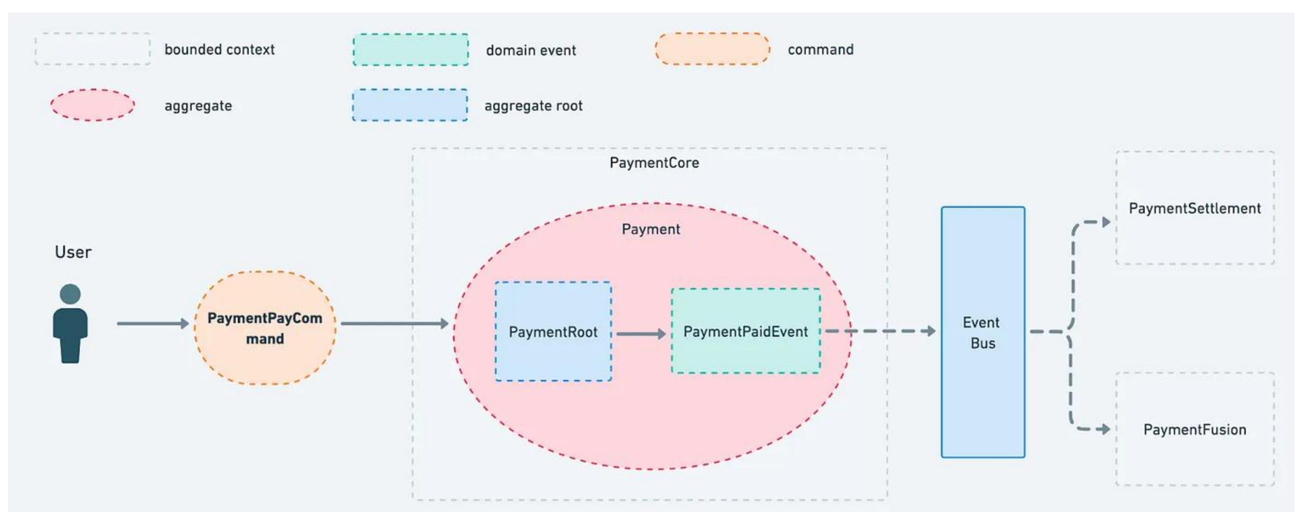

Рисунок 1.1.1 – Інтеграція Event Sourcing з DDD [6]

«Event-Driven Architecture» є шаблоном архітектури програмного забезпечення, який акцентує увагу на виробленні, споживанні та реагуванні на події. «Event Sourcing» добре вписується в EDA, оскільки події є основним елементом обох концепцій. Використання «Event Sourcing» у контексті EDA дозволяє зберігати історію змін у формі подій, що може бути використано для

асинхронної комунікації між різними частинами системи, підвищення масштабованості та забезпечення більшої гнучкості у відповідях на події. На відміну від звичайного застосування EDA, у випадку такої інтеграції (рис. 1.1.2) події використовуються не тільки для сповіщення системи про зміни, але й зберігаються відповідно до патерну «Event Sourcing», що дозволяє використовувати їх в подальшому для аудиту і відновленню станів іншими компонентами системи.

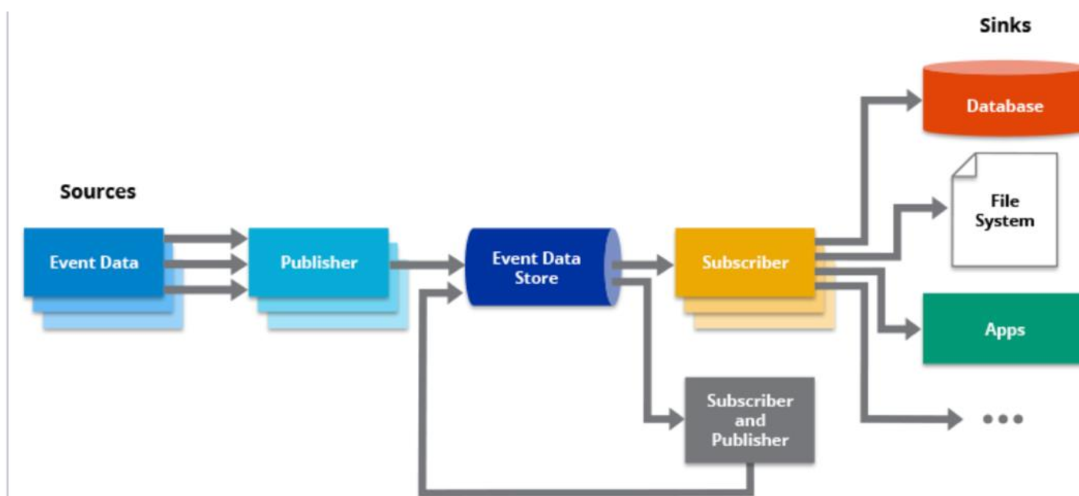

Рисунок 1.1.2 – Інтеграція Event Sourcing з EDA [7]

«Command Query Responsibility Segregation» є патерном, який розділяє операції читання та запису даних на дві окремі моделі, оптимізуючи їх для відповідних операцій. Комбінація «Event Sourcing» з CQRS призводить до потужної, хоч і простої, архітектури (рис.1.1.3), де події, що відображають зміни стану, можуть бути використані для асинхронного оновлення моделей читання, тим самим забезпечуючи високу продуктивність та консистентність даних. Такий підхід дозволяє ефективно масштабувати систему та оптимізувати відповідь на запити користувачів.

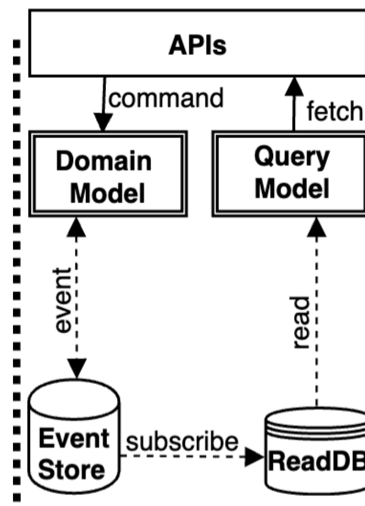

Рисунок 1.1.3 – Інтеграція Event Sourcing з CQRS [8]

1.2 Основи та принципи роботи

Застосування патерну «Event Sourcing» передбачає збереження всіх змін у станах об’єктів і системи як послідовність подій, що відрізняється від більш звичного збереження лише останнього стану сутності. Кожна подія описує зміну, яка сталася, і містить достатньо інформації для того, щоб можна було відтворити цю зміну.

Важливо зауважити, що набору подій не завжди достатньо, щоб відтворити кінцевий стан об’єкту, так як можуть існувати певні правила за якими ці події застосовуються до сутності в залежності від її поточного стану. Таким чином, для того, щоб відтворити сутність необхідно мати послідовність подій та правила застосування цих подій. На відміну від подій, правила їх застосування зберігаються не в сховищі даних, а в коді. Для керування цими правилами використовується агрегована сутність. Ці сутності відіграють критично важливу роль, оскільки вони допомагають керувати змінами стану системи в організований та консистентний спосіб. Така сутність складається з агрегатного кореня, який є єдиною точкою входу для змін усередині агрегату і гарантує його консистентність та набору правил, які визначають яким чином події впливають на сутність. Коли відбувається зміна стану, пов'язана з агрегатом, ця зміна створюється через агрегатний корінь. Агрегатний корінь забезпечує дотримання

всіх бізнес-правил і інваріантів перед тим, як застосувати зміни. Також, він відповідає за генерацію подій, які будуть записані у сховище даних.

Такий спосіб конструювання сутностей природньо забезпечує важливий механізм патерну, а саме відтворення стану сутностей. Для того, щоб отримати стан сутності у певний момент часу, достатньо застосувати не всі події, а лише ті, що відбулись до цього моменту, що призведе до точного відтворення стану сутності. При цьому, важливим є не допускати генерацію нових подій у не повністю відтворених сутностей, так як це призведе до появи альтернативного стану об’єкту і відповідних до цього конфліктів. Незважаючи на те, що під час роботи системи не повинно виникати невизначеної поведінки за якої може бути порушений послідовний порядок запису подій і існують методи, які дозволяють це забезпечити, такі як стратегія оптимістичного контролю одночасності або використання часових міток [9], іноді цю особливість можна використати для спрощення розробки, симулюючи альтернативні варіанти станів для перевірки певних гіпотез [10].

Але працюючи з цим патерном, не варто забувати, що робота з великою кількістю подій може призвести до суттєвого погіршення швидкодії застосунку. Для того, щоб запобігти цьому і пришвидшити критичні частини системи використовуються знімки станів (снепшоти). Снепшоти реалізуються шляхом створення знімків поточного стану доменної моделі на певні моменти часу, що дозволяє системі пропустити частину подій при наступному відновленні стану. Також, може бути створена окрема модель, яка буде сконструйована виключно зі снепшотів і містити обмежену інформацію про агреговану сутність, яка необхідна для виконання певних операцій.

Снепшоти можуть зберігатись як разом з іншими подіями, будучи реалізованими як окрема подія зі специфічним призначенням або в окремому місці з окремою структурою (рис. 1.2.1). Переваги першого способу полягають у тому, що він не вимагає окремої реалізації снепшотів, а дозволяє використати існуючу реалізацію збереження подій. Також, у цьому випадку, снепшоти можна сумістити з бізнес подіями, створюючи підсумовуючі події, які будуть не тільки

снепшотами, а і частиною бізнес логіки. Влучним прикладом таких подій є подія закриття зміни у супермаркеті, яка може містити агреговані данні про всі операції, що відбулись за цю зміну. Переваги іншого методу, який передбачає окрему імплементацію і збереження снепшотів окремо від подій полягає у ізоляції функціоналу снепшотів від подій, що ще сильніше прискорює відновлення станів. Однак, це ускладнює технічну реалізацію і призводить до явної дуплікації даних.

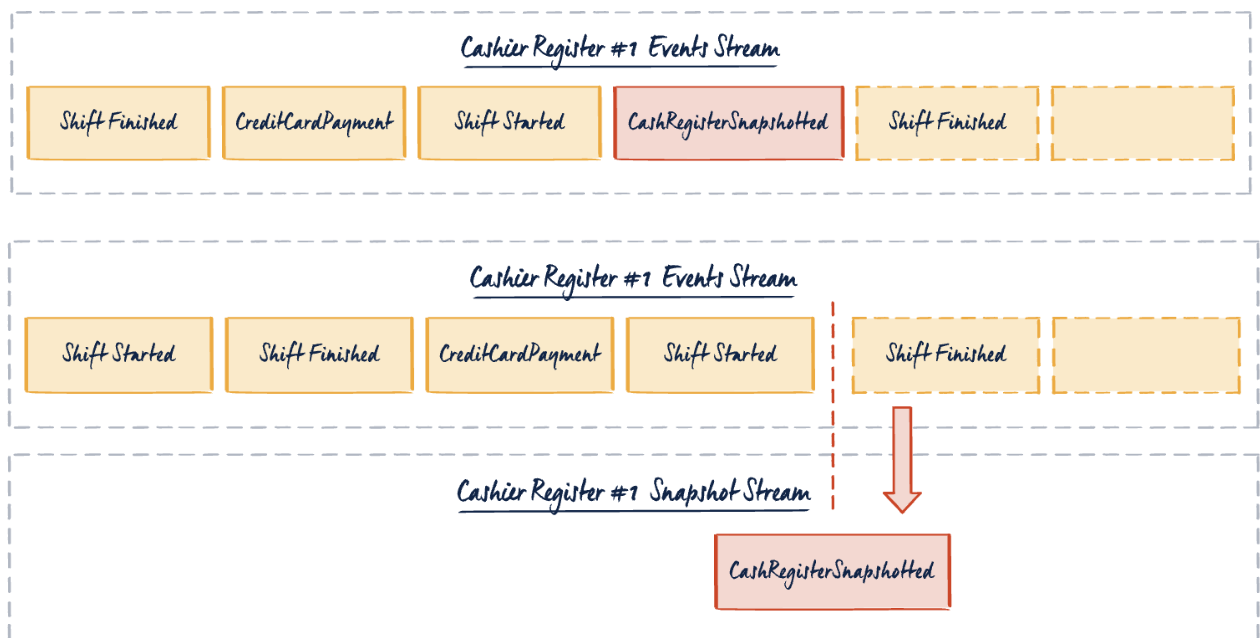

Рисунок 1.2.1 – Демонстрація різних методів зберігання снепшотів [11]

Крім методу зберігання, важливо правильно обрати умову для створення снепшотів: занадто часте їх створення може навантажувати систему, в той час як занадто рідкісне може знизити потенційні переваги від їх використання. Існують декілька основних підходів до частоти зберігання [12]:

- Після кожної події
- Після певної кількості подій
- Після того, як відбувається певна подія
- Після певного проміжку часу

Перший спосіб, хоча і зазначений, але майже не має практичного сенсу, бо майже нівелює усю перевагу патерну і прибирає сенс запису окремих подій. Другий спосіб використовується частіше і гарантує, що під час вичитки

необхідно буде прочитати додатково не більше ніж ту кількість подій, раз на яку робиться снепшот, але головним мінусом є те, що такий підхід ускладнює снепшот і зазвичай передбачає збереження всієї сутності, що часто не є оптимальним. Третій підхід найбільше підходить для снепшоту частини інформації про об’єкт, що з одного боку дозволяє не зберігати надлишкову інформацію, але з іншого надає можливість критично прискорити важливі операції в системі, які мають вимоги до швидкодії. Однак, разом з перевагами, він має недолік у тому, що для цього необхідно вибрати ключову подію, яка буде ініціювати запис створення снепшоту і це може ускладнити домену модель, а при обранні помилкової події сповільнити систему через створення великої кількості зайвих снепшотів. Четвертий підхід дуже схожий на метод резервного копіювання. Головна його перевага в тому, що він майже не призводить до ускладнень моделі і бізнес логіки і має легко зрозумілий принцип роботи, але разом з тим може бути вкрай неефективним для певних систем, в яких кількість створених подій не є регулярною і передбачуваною.

Таким чином, снепшоти хоч і є надзвичайно потужним механізмом, який дозволяє значно пришвидшити реконструювання об’єкту, але потребують від розробників чіткого розуміння, як і для чого вони збираються їх використати. Головні плюси снепшотів включають зменшення часу відновлення стану та оптимізацію використання ресурсів. Однак, використання снепшотів також має певні мінуси, як-от потенційна складність управління консистентністю даних між снепшотами та подіями. Найкраще їх використовувати в системах, де час відновлення стану критично важливий або коли кількість подій стає настільки великою, що відновлення всіх подій стає технічно затратним чи повільним.

Окрім вищезазначеного, агреговані сутності надають простір для модифікацій і версіонування подій. Більшість систем постійно еволюціонують і змінюються. Хоча події у цьому патерні зазначені як атомарні і незмінні, але це стосується переважно заборони на будь-яку модифікацію подій під час роботи системи у звичайний спосіб. Самі події і їх застосування можуть з часом змінюватись під час розробки через зміну вимог або оптимізацію певних

процесів. Це може бути виражено у додаванні певних даних до якоїсь події, або навпаки, певні дані події можуть втратити свій сенс і більше не використовуватись. Ці змінні можуть бути як сумісними і не потребувати окремих дій і втручання в збережені події, так і бути несумісними і вимагати певних дій для забезпечення сумісності даних після внесених змін.

Авжеж, найпростішими змінами є ті, які не потребують жодних змін у самих подіях і дозволяють використання старих даних без додаткових дій. До таких змін відноситься прибирання даних з подій, так як вони можуть бути просто проігноровані при їх наявності у старих версіях та додавання даних при умові, що вони необов'язкові, так як всі старі події будуть сприйматись як такі, що не містять цих даних. Набагато складнішим є змінні, які або потребують додавання нових обов'язкових атрибутів, або зміну типу даних. Найпростіше рішення у такому випадку – створення нової події, але з часом це може призвести до значного розростання класів подій і погіршити загальний стан коду. Крім цього, таке рішення не є правильним концептуально, адже якщо зі сторони бізнес логіки подія і її сенс не змінювався, то немає підстав створювати нову подію у застосунку. Інші підходи, хоч і є більш складними, але одночасно з цим принесуть переваги при подальшій розробці. Одне з можливих рішень – міграція даних. Це дозволить позбавитись від усіх старих даних і перетворити всі старі версії у сховищі даних на нові. Завдяки цьому не треба буде зберігати у коді логіку роботи з попередніми версіями. Однак, найбільший мінус такого рішення – сама міграція даних, яка порушує незмінність подій у системі, що є ключовим принципом патерну. При використанні цього способу рекомендується використовувати техніку «Міграції Без Простоїв» [13], яка полягає у здійснені міграцій в декілька етапів для забезпечення безперервної роботи системи. Інше рішення – перетворення старої версії у нову під час реконструювання подій, є найбільш правильним підходом з точки зору патерну. Це дозволяє зберігати події у сховищі даних незмінними, а логіку їх перетворення покласти на саму систему. Таким чином, в сховище будуть зберігатись події у їх початковому вигляді, але система буде сприймати їх як події останньої версії (рис. 1.2.2). Мінусом у

такому підході є розростання логіки системи, адже вона повинна знати не тільки як застосувати певну подію, але й як конвертувати її в останню версію. Також, така конвертація сповільнює роботу системи, що може бути скомпенсованим наявністю снепшотів, завдяки яким старі події будуть використовуватись дедалі менше.

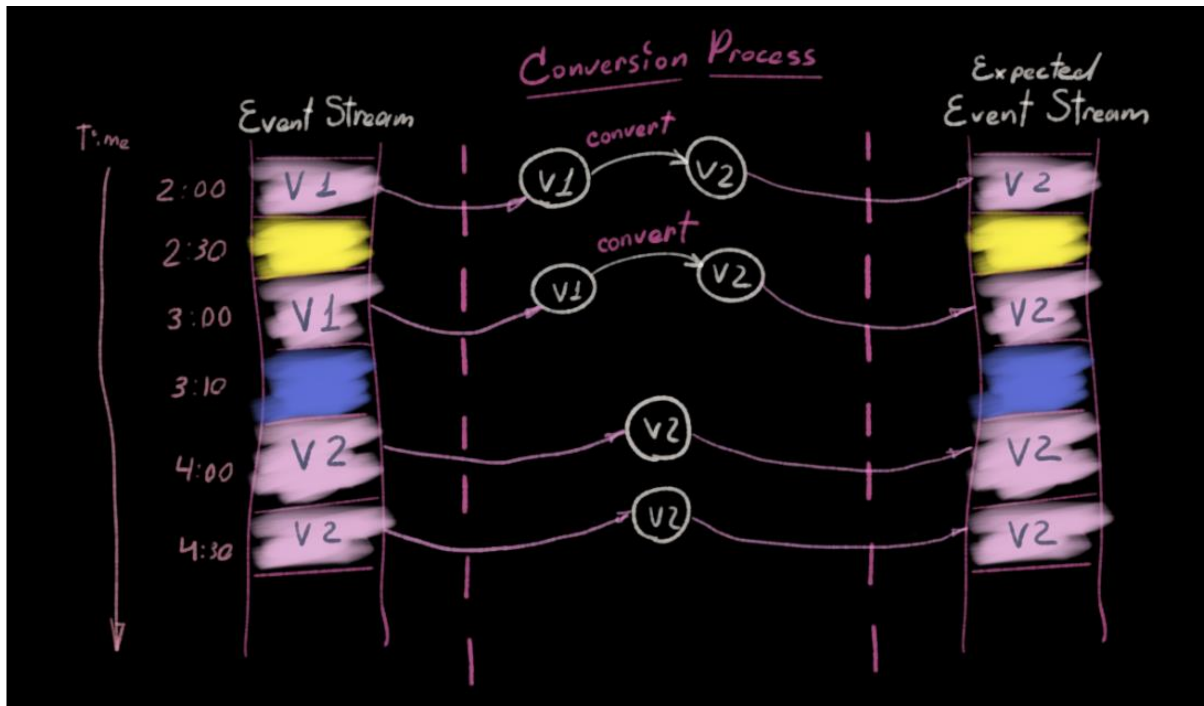

Рисунок 1.2.2 – Процес перетворення подій без змін у сховищі даних [14]

1.3 Переваги та недоліки

Як і усі інші технології, патерн «Event Sourcing» має ряд переваг і недоліків, на які треба зважати, приймаючі рішення про його застосування під час розробки. Оскільки він визначає спосіб зберігання даних, то приймати рішення треба зважено і обережно, так як під час розробки відмовитись від його використання, або навпаки – впровадити, буде вкрай важко і потребуватиме значних зусиль з боку розробників.

Основною перевагою є можливість докладного логування всіх змін, які відбуваються в системі. Оскільки кожна подія зберігається і записується, то в будь-який момент часу можна переглянути історію змін або відновити систему до будь-якого попереднього стану. Для систем з підвищеними вимогами до безпеки, або зі складним доменом, де треба розуміти, яка послідовність подій

привела систему в поточний стан, це може бути вирішальним фактором для використання цього патерну. Також, це підвищує відмовостійкість системи, завдяки можливості швидко відновити стани подій у випадку збоїв. Згідно дослідженню, половина розробників, що використовують цей патерн користувались цією причиною для його впровадження [5].

Окрім цього, існують переваги які пов'язані з більш зручною розробкою систем, що використовують цей патерн. По-перше, це його легка інтеграція з іншими архітектурними патернами такими як DDD, EDA, CQRS. Така інтеграція не тільки можлива і легка в реалізації, а й дозволяє досягти ефекту синергії, значно покращуючи кінцеві результати. По-друге, це можливість вносити зміни в систему шляхом додавання нових подій або модифікації існуючих. Завдяки тому, що існують чітко прописані кроки які потрібно зробити для внесення змін, це дозволяє розробникам знизити ймовірність проблем під час розробки і еволюції системи і працювати зі змінами будь-якої складності без зайвих ризиків. Цими перевагами керуються 65% розробників [5], що свідчить про те, що зручність розробки, яку забезпечує патерн є навіть більш важливою, ніж його основна функціональність у вигляді роботи з історичними даними.

Авжеж, є і низка недоліків, які можуть стати важливим фактором, чому не варто застосовувати цей патерн. Основним є необхідність зберігання великої кількості подій, що може призвести до значного зростання обсягу даних [5]. Це може створити виклики, пов'язані з управлінням даними, їхньою архівацією та процесом відновлення. Для певних систем, атрибути яких часто змінюються і перезаписуються, це може збільшити обсяг даних, що зберігаються, в рази. А через необхідність мати усю послідовність подій разом для реконструювання стану, архівувати частину або навіть видалити застарілі дані не є можливим.

Крім викликів, які пов'язані зі сховищем даних, є також інші виклики, які пов'язані з самими розробниками. Реалізація «Event Sourcing» вимагає більш складної архітектури порівняно з традиційними підходами. Розробники повинні враховувати як логіку застосунку, так і логіку збереження подій, що може збільшити складність проекту та потребувати більш висококваліфікованих

фахівців. А оскільки для ефективного використання «Event Sourcing» важливо глибоко розуміти доменну область, то розробники повинні ще й мати певну експертизу, що ще сильніше ускладнює поріг входу. Недостатнє розуміння і брак кваліфікації може призвести до неправильного моделювання подій, що ускладнить майбутнє внесення змін та розширення системи.

І окремо можна виділити незмінність подій, що є одною з ключових особливостей патерну. На відміну від інших особливостей, ця є одночасно, як і перевагою, так і недоліком. . З одного боку, незмінність гарантує, що історія подій зберігається без змін, що забезпечує високий рівень довіри до аудиту, дозволяючи системі точно відтворювати стан на будь-який момент часу. З іншого боку, сама ж незмінність може стати недоліком, коли виникає необхідність вносити зміни в уже зареєстровані події. Це може статися у випадку помилок або коли потрібно ввести корективи, що відповідають зміненим обставинам або законодавчим вимогам. В таких випадках система повинна включати механізми для обробки виправлень, що може складатися у введенні нових подій, які компенсують або анулюють ефект попередніх. Це збільшує складність системи та може вплинути на її продуктивність та простоту управління.

Підсумовуючи, патерн «Event Sourcing» пропонує ряд значущих переваг, які можуть бути вирішальними для систем з високими вимогами до документування змін і аудиту. Він дозволяє детально відслідковувати історію змін, відновлювати систему до будь-якого попереднього стану та інтегрується із сучасними архітектурними патернами, підвищуючи гнучкість та ефективність розробки. Однак, разом з перевагами, «Event Sourcing» несе в собі й певні виклики, такі як потреба у збереженні великої кількості подій, що може призвести до збільшення обсягу даних та вимагати від розробників глибшого розуміння домену та складніших технічних рішень. Незмінність подій, хоча й є ключовою особливістю, також може додати складності у випадках, коли потрібно коригувати минулі записи. Тому прийняття рішення про використання патерну повинно базуватися на глибокому аналізі специфіки проекту та

потенційних вигід і труднощів, які можуть виникнути. Важливо зважити всі переваги і можливі проблеми перед впровадженням цього патерну в розробку, щоб забезпечити максимальну ефективність і адаптацію до потреб системи.

1.4 Застосування

«Event Sourcing» є потужним патерном проектування, який використовується в різноманітних галузях для забезпечення детального логування змін, відновлення стану системи, та високої надійності. Він особливо корисний в галузях, де критично важливим є точне відстеження історії транзакцій або станів, наприклад у фінансових послугах, електронній комерції, ігровій індустрії, охороні здоров'я та високонавантажених системах обробки даних.

Патерн добре підходить для комплексних систем, де потрібно відновлювати попередні стани, аналізувати зміни або забезпечувати високий рівень надійності. Завдяки своїй сумісності з іншими архітектурними патернами, такими як CQRS та DDD, «Event Sourcing» може бути інтегрований у широкий спектр програмних рішень, збільшуючи їхню ефективність та оптимізуючи управління даними і бізнес-процесами.

Цей патерн може застосовуватись не до всієї системи, а лише до окремих її найбільш критичних частин, які потребують додаткового рівню контролю і перевірок, що ще більше розширює можливі галузі для його застосування.

На жаль, не так багато відомих компаній розповідають деталі про технічну реалізацію своїх додатків, тому важко відслідкувати, наскільки цей патерн дійсно є поширеним і дотичним доказом може бути лише список партнерів, які використовують EventStoreDB [15], яка є однією з найбільш відомих баз даних, що спеціалізуються саме на збережені подій.

На їх фоні виділяється Netflix, який у своєму технічному блозі ділиться деталями використання «Event Sourcing» [16]. Розробники Netflix застосовують патерн «Event Sourcing» у своїй системі для керування завантаженнями контенту, що дозволяє користувачам зберігати фільми та серіали на своїх пристроях для перегляду без мережі. Цей підхід вони обрали через його спроможність

забезпечувати високу масштабованість, відмовостійкість та надійність, що є критично важливим для сервісу з мільйонами користувачів по всьому світу.

Кожна активність користувача, така як запит на завантаження, пауза, відновлення або скасування завантаження, реєструється як окрема подія. Ці події надходять у систему, де вони агрегуються та обробляються. Завдяки цьому підходу, Netflix може точно відновлювати стан системи завантажень у будь-який момент, перевіряти коректність станів та аналізувати користувацьку поведінку.

Для оптимізації ними було впроваджено снепшоти для станів, які найчастіше запитуються, що дозволило їм значно знизити затримку і зменшило навантаження на базу даних. А для того, щоб керувати великим об'ємом подій, Netflix використовує розподілені системи та асинхронну обробку, що дозволяє системі ефективно масштабуватися і витримувати високі навантаження.

Найбільшим викликом для Netflix було забезпечення консистентності даних між різними компонентами системи, що було вирішено за допомогою ретельно спланованих механізмів синхронізації та впровадження надійних стратегій відновлення після збоїв.

Таким чином, завдяки імплементації патерну «Event Sourcing», Netflix зміг забезпечити високий рівень надійності, доступності та масштабованості для своєї глобальної платформи скачувань. Цей підхід дозволив детально логувати всі зміни в системі, відновлювати попередні стани в разі потреби та проводити глибокі аудити. Масштабування обробки подій, кешування стану сервісів, та оптимізація зберігання даних є ключовими аспектами, що сприяли зменшенню навантаження на систему і забезпеченню швидкої відповіді на запити користувачів.

1.5 Існуючі інструменти та технології

Незважаючи на те, що цей патерн було вперше описано у 2005 році [4], станом на 2023 рік у відкритому доступі існує не так багато відкритих рішень які би спрощували його імплементацію.

Більшість наявного на 2023 рік матеріалу пропонує не готові рішення, а лише невеликі керівництва по використанню цього патерну. Зазвичай вони

містять мінімальну реалізацію для демонстрації можливостей патерну, що є безумовно корисним для розробників, які хочуть вивчити цей патерн, але не демонструє і не показує як вирішити головні проблеми з якими стикаються розробники під час використання патерну. Такі рішення хоч і можуть бути хорошою точкою входу для тих, хто вирішив використати цей патерн, але не є готовими для використання у справжніх додатках, які оперують великою кількістю даних.

Тим не менш, у відкритому доступі можна знайти фреймворки для таких мов як:

- .Net
- Elixir
- Typescript
- Python
- Php
- Java

Незважаючи на велику кількість мов, для яких існують імплементації, що дозволяють використати патерн, станом на 2023 рік, лише .NET має декілька реалізацій, серед яких розробники можуть обирати. Всі інші мови мають лише одну реалізацію.

Фреймворки, які реалізують «Event Sourcing», часто є нішевими і рідко досягають значної популярності, якщо орієнтуватись на кількість зірок у відповідних проектах на GitHub, де міститься код цих фреймворків, що свідчить про специфічний коло застосування та аудиторію. Проте серед усіх виокремлюються рішення на .NET та Python маючи значно більшу популярність ніж їх аналоги іншими мовам. Завдяки своїй функціональності та спільноті, вони підтримують високий рівень інтересу та розвитку від своїх користувачів.

Більшість існуючих фреймворків дозволяє реалізувати ключові можливості патерну «Event Sourcing», однак вони роблять це з різним рівнем успіху та зручності для кінцевого користувача. Всі проаналізовані фреймворки обмежені підтримкою інтеграції тільки з певними сховищами даних, і не надають

можливості для реалізації власної інтеграції з іншими системами зберігання. Також, важливо зазначити, що у них відсутня реалізація часткових снепшотів, що може бути критичним для деяких застосувань. Лише декілька з них підтримують асинхронну обробку подій, що є важливим для систем з високими вимогами до продуктивності та відгуків.

Згідно з результатами аналізу існуючих фреймворків, можна дійти висновку, що вони не дозволяють повноцінно розкрити весь потенціал цього патерну та містять низку недоліків, що може обмежувати їх ефективність в певних сценаріях застосування. Ці обмеження підкреслюють необхідність у розробці нового фреймворку, який би виправив існуючі недоліки та розширив можливості розробників при використанні патерну «Event Sourcing». Створення такого рішення дозволило би забезпечити більш гнучке управління даними, оптимізувати відновлення станів системи, і забезпечити асинхронну підтримку, що робить можливим впровадження цього патерну в масштабних та вимогливих застосуваннях. Таким чином, розробка власного фреймворку, який враховує ці потреби, стає важливим кроком на шляху до покращення архітектурних практик та технологій в області програмування.

2 РОЗРОБКА АРХІТЕКТУРИ ФРЕЙМВОРКУ ДЛЯ МОВИ PYTHON

2.1 Вступ до архітектури фреймворку

Розробка нового фреймворку для мови Python, що використовує патерн «Event Sourcing», є актуальним завданням, що відповідає сучасним вимогам програмної інженерії. Основна мета цього проекту полягає у створенні фреймворку, який не тільки впроваджує цей патерн, але й усуває існуючі недоліки в застосуванні подібних рішень, забезпечуючи високу надійність та легкість у використанні.

Мотивація до розробки фреймворку виникла через виявлену потребу в більш ефективних інструментах для роботи з проаналізованим патерном. «Event Sourcing» дозволяє зберігати всю історію змін стану системи, що відкриває нові можливості для аудиту, відновлення даних та відстеження помилок у системі. Розроблюваний фреймворк покликаний максимально спростити використання цього патерну, надаючи розробникам потужний, але інтуїтивно зрозумілий інструмент.

У відповідь на обмеження існуючих рішень, новий фреймворк зосереджується на покращенні швидкості розробки і доступності інтеграцій. Це стає можливим за рахунок оптимізації процесів управління станами та подіями, а також наданням додаткових абстракцій для розширення фреймворку і інтеграції його у власні системи.

Вибір мови Python для розробки фреймворку обумовлений кількома важливими факторами, серед яких ключовим є її висока популярність серед розробників. Python займає провідні позиції серед мов програмування завдяки своїй читабельності, гнучкості та широкому спектру застосування в різних галузях — від веб-розробки до наукових досліджень [17]. Ця мова має значну кількість існуючих бібліотек та фреймворків, але лише одне рішення, яке реалізує патерн «Event Sourcing». Розробка спеціалізованого фреймворку для «Event Sourcing» на Python надасть розробникам зручний та ефективний

інструмент, що дозволить їм використовувати потенціал мови для створення надійних та масштабованих систем.

У цьому розділі буде представлено детальний опис архітектури розроблюваного фреймворку. Будуть досліджені технічні рішення, які були обрані для впровадження патерну «Event Sourcing», виокремлюючи основні компоненти архітектури та їх взаємодії. Розгляд цих аспектів є критично важливим, оскільки правильне проектування архітектури є ключем до забезпечення масштабованості, ефективності та легкості у використанні фреймворку. Отримане рішення повинно не тільки поліпшить технічну якість розроблюваних систем, але й зробить процес розробки більш зручним та адаптивним до різноманітних вимог проектів.

2.2 Вимоги до фреймворку

Визначення вимог є фундаментальним етапом у процесі розробки будь-якого програмного продукту, і фреймворк не є винятком. Цей процес є важливим, оскільки допомагає забезпечити, що кінцевий продукт відповідатиме очікуванням користувачів і технічним специфікаціям, а також задовольнить встановлені стандарти якості та продуктивності. Вимоги діють як визначальні параметри, що керують розробкою, впровадженням і тестуванням продукту, забезпечуючи необхідні орієнтири для кожного етапу проекту [18].

Для розробки даного фреймворку зосередимось на двох основних типах вимог: функціональних та нефункціональних. Користуючись порівняльною таблицею (табл. 1.2.1), можна побачити, що функціональні вимоги описують конкретні дії та операції, які система повинна виконувати. Ці вимоги чітко визначають, що система робить і як вона повинна реагувати на взаємодії з користувачами. З іншого боку, нефункціональні вимоги стосуються якості та характеристик системи, таких як надійність, продуктивність, масштабованість та безпека. Вони не описують прямі дії системи, але критично важливі для забезпечення ефективної та стабільної роботи фреймворку.

	Functional requirements	Nonfunctional requirements
Objective	Describe what the product does	Describe how the product works
End result	Define product features	Define product properties
Focus	Focus on user requirements	Focus on user expectations
Essentiality	They are mandatory	They are not mandatory but desirable
Origin type	Usually defined by the user	Usually defined by developers or other tech experts
Testing	Component, API, UI testing, etc. Tested before nonfunctional testing	Performance, usability, security testing, etc. Tested after functional testing
Types	Authentication, authorization levels, data processing, reporting, etc.	Usability, reliability, scalability, performance, etc.

Таблиця 2.2.1 – Порівняння функціональних та нефункціональних вимог [19]

З урахуванням вищезазначеного, сформуємо детальний список функціональних та нефункціональних вимог для розроблюваного фреймворку. Це дозволить ретельно спланувати розробку та забезпечити, що фреймворк буде відповідати всім вимогам ефективності, надійності та зручності використання.

Визначимо наступні функціональні вимоги для фреймворку та обґрунтуємо їх важливість для фреймворку:

- Фреймворк повинен забезпечувати здатність записувати події в системі, зберігаючи всі зміни стану об'єктів або системи у вигляді незмінних подій. Це необхідно для аудиту, відстеження змін і можливості відновлення системи до будь-якого попереднього стану.
- Фреймворк повинен надавати механізми для відновлення стану об'єктів системи за допомогою відтворення подій від початкового стану до поточного, що є важливим для відновлення після збоїв і перевірки історичних даних.
- Фреймворк повинен підтримувати можливість створення снепшотів. Ця функція дозволяє системі пропустити певну кількість подій при

відновленні станів, значно зменшуючи час, необхідний для реконструкції стану з великої історії подій.

- Фреймворк повинен забезпечувати консистентність даних при обробці транзакцій для зменшення помилок і гарантування цілісності даних.
- Фреймворк повинен надавати можливість асинхронної взаємодії зі сховищами даних, що дозволить обробляти високий об'єм запитів без блокування користувацьких інтерфейсів.
- Фреймворк повинен надавати API для його розширення та кастомізації, що дозволить адаптувати його під специфічні потреби проектів, надаючи розробникам необхідну гнучкість.
- Фреймворк повинен надавати можливості версіонування подій, що дозволить системі еволюціонувати, змінюючи формати подій без порушення роботи існуючих процесів, забезпечуючи гнучкість у розробці та можливість впровадження нововведень.
- Фреймворк повинен надавати інтерфейси для інтеграції зі сховищами даних, що забезпечить можливість використання існуючої інфраструктури, спрощуючи інтеграцію і масштабування.

Ці функціональні вимоги формують основу для створення робочого, надійного та ефективного фреймворку, що дозволяє втілити патерн «Event Sourcing» та який може бути адаптований до різних бізнес-потреб і технологічних середовищ.

Тепер, маючи сформовані функціональні вимоги, визначимо та обґрунтуємо нефункціональні вимоги для фреймворку:

- Фреймворк має бути легким у налаштуванні, розгортанні та управлінні, щоб мінімізувати витрати часу на технічну підтримку і оновлення.
- Фреймворк має бути гнучким у використанні, дозволяючи користувачам легко налаштовувати або модифікувати функціональність відповідно до змінних вимог бізнесу.

- Фреймворк має надавати користувачам повну, чітку і доступну документацію, що є необхідним для забезпечення легкого засвоєння та ефективного використання фреймворку.
- Фреймворк повинен дозволяти структурувати події зручним чином і надавати можливість легко додавати нові та змінювати існуючі події. Це важливо для адаптації системи під змінні бізнес-вимоги та технологічні умови, дозволяючи забезпечити максимальну гнучкість і масштабованість в обробці подій.

Завдяки таким нефункціональним вимогам розроблюваний фреймворк буде надавати користувачам широкий спектр можливостей і свободу використання у різних сценаріях та буде здатний задовольнити різні технічні та бізнесові потреби, забезпечуючи тривале та ефективне використання фреймворку в різних середовищах.

2.3 Принципи проектування фреймворку

Для створення ефективного та надійного фреймворку на основі «Event Sourcing», важливо дотримуватися декількох ключових принципів проектування. Це не тільки впливатиме на технічну реалізацію, але й гарантує, що фреймворк буде відповідати довгостроковим потребам користувачів у різних бізнес-сценаріях. Правильно обрані принципи допомагають створити систему, яка може ефективно масштабуватися, легко адаптуватися до нових вимог і технологічних змін, а також забезпечувати високий рівень безпеки та надійності.

Визначимо ключові принципи, яких буде дотримано під час розробки фреймворку:

- Модульність. Фреймворк має бути розроблений таким чином, щоб його компоненти були модульними. Це дозволяє легко замінювати або оновлювати окремі частини системи без впливу на решту функціональності.
- Розширюваність. Система повинна дозволяти розширення функціональності через плагіни або API. Це забезпечить можливість

додавання нових можливостей або інтеграцію з іншими системами, не переписуючи існуючий код.

- Розмежування зон відповідальності. Кожен компонент фреймворку має мати чітко визначену зону відповідальності, мінімізуючи залежності між різними частинами системи.
- Легкість використання. Інтерфейси фреймворку мають бути інтуїтивно зрозумілими та простими в застосуванні, щоб нові користувачі могли легко втілити його можливості в своїх проектах.

Обрані принципи сприяють створенню системи, яка не тільки відповідає поточним технічним вимогам, але й забезпечує легку адаптацію до майбутніх змін, забезпечуючи гнучкість у розвитку та підтримці. В результаті, фреймворк виявляється стійким до змін у технологічному середовищі, ефективним у роботі та зрозумілим для кінцевих користувачів, що робить його ідеальним рішенням для широкого спектра бізнес-застосувань.

Мова Python добре підходить для втілення цих принципів. Так, модульність досягається використанням пакетів та простору імен для організації коду в чітко визначені модулі, що можна легко імпортувати та використовувати у різних частинах програми. Розширюваність і розмежування зон відповідальності можуть бути втіленні за допомогою об'єктно-орієнтованого програмування. Визначення чітких інтерфейсів і абстракцій у класах сприяє створенню незалежних компонент, а наслідування і декоратори надають механізм для розширення існуючих класів та функцій без зміни їхнього первісного коду. Щодо легкості використання, Python відомий своїм чистим та інтуїтивно зрозумілим синтаксисом, що робить код легким для читання та написання. Це особливо корисно при проектуванні фреймворків, де зрозумілість коду може значно спростити вивчення і використання фреймворку для нових користувачів. Використання коментарів у коді та дотримання стандартів PEP8 [20] також сприяє підвищенню якості коду та його легкості використання.

2.4 Компоненти та структура фреймворку

Перш за все, визначимо структуру компонентів, які будуть відповідати за події і сутності, що з ними пов'язані, так як ця структура закладає основу для всієї системи. Для роботи з цим функціоналом визначимо такі сутності як «AggregateRoot» та «Event».

AggregateRoot є кореневим елементом агрегату — групи об'єктів, які спільно обробляються як одне ціле. У нашому випадку, групи подій, що відносяться до однієї сутності. Це означає, що всі зовнішні взаємодії з агрегатом відбуваються через AggregateRoot, який гарантує консистентність даних всередині агрегату та є єдиною точкою входу для виконання змін. У нашій розробці AggregateRoot буде реалізований як абстрактний клас. Така реалізація дозволить визначити універсальний інтерфейс для всіх агрегатів. Головна мета цього компоненту – забезпечити механізм для обробки подій і збереження поточного стану об'єктів під час роботи з ними.

Event є репрезентацією події у системі. Кожен екземпляр відображає конкретну подію або дію, яка впливає на стан додатку. Як і AggregateRoot цей клас буде реалізований як абстрактний. Завдяки цьому з'являється можливість визначити правила взаємодії подій і агрегатного кореня, стандартизувати структуру та поведінку всіх подій у системі. З абстрактним класом як базою, розробники можуть легко додавати нові типи подій, розширюючи базовий клас. Це полегшує масштабування системи і введення нової функціональності без порушення існуючої структури.

Не менш важливим ніж самі класи – є визначення того, як вони будуть взаємодіяти між собою. Один з механізмів, що часто використовується – визначення методів агрегату, що будуть створювати відповідні події. Це дозволяє працювати ізольовано від подій, що може спростити складність застосунку, але разом з цим це заважає використовувати події окремо від агрегату, що може бути корисним для інтеграції з DDD підходом або з CQRS. Тому, для подальшій розробки пропонується зосередитись на інших методах взаємодії. Так, можна використати декоратори - шаблон проектування, який

дозволяє модифікувати роботу функції, обернувши її в іншу функцію. Ці декоратори можуть бути застосовані до методів у класі AggregateRoot, які відповідають за обробку конкретних подій. Кожен метод, який обробляє подію, декорується спеціальним декоратором, який автоматично пов'язує декоруємий метод з подіями зазначеними у декораторі. Це означає, що коли подія цього типу застосовується, система автоматично викликає цей метод для обробки події. Також, такий декоратор забезпечує, що зміни стану, здійснені методом, атомарні та консистентні, що дуже важливо для забезпечення цілісності даних в «Event Sourcing». Завдяки такому підходу ми покладаємо правила застосування подій на агрегатний корінь, що дозволяє нам відокремити бізнес логіку, від самих подій. Однак, у випадку складних систем з різноманітними подіями, ми матимемо дуже великі класи агрегатів серед яких буде важко орієнтуватись. Інший спосіб полягає у тому, щоб зберігати у кожному класі події метод, що визначатиме, як ця подія впливатиме на агрегат. Ця інформація може бути використана декоратором для автоматичного виклику правильного методу обробки в AggregateRoot. Це дозволяє розподілити бізнес логіку по різним модулям, але порушує принцип єдиної відповідальності, оскільки у такому підході події стають не тільки носіями даних про зміни, але й беруть на себе відповідальність за логіку, яка регулює ці зміни. Це веде до того, що події стають перевантаженими та менш зосередженими на своїй первинній ролі.

Таким чином, найбільш коректно у розрізі патерну було би реалізувати лише декоратор, як такий, що найбільш точно дозволяє реалізувати основну ідею «Event Sourcing», але беручи до уваги можливу складність систем і керуючись визначеними попередньо принципами, буде реалізовано як можливість взаємодії за допомогою декораторів, так і визначення логікі у самої події, незважаючи на недоліки цього методу.

Для забезпечення ефективної роботи з великою кількість подій та підвищення продуктивності, у фреймворку буде реалізовано механізм снепшотів. Буде реалізовано як повний снепшот стану, так і частковий, що агрегує лише частину інформації. Для повного снепшоту будуть додані атрибути

у агрегатний корінь, які будуть визначати частоту снепшоту. Під час взаємодії зі сховищем даних, буде створена окрема подія, яка буде репрезентувати снепшот. Використовуючи останню таку подію і всі події, створені після неї, можна буде відтворити останній стан сутності. Головний недолік такого підходу до снепшотів полягає у тому, що це потребує багато додаткового місця для зберігання надлишкової інформації, хоч і значно прискорює роботу системи. Для того, щоб забезпечити можливість оптимізувати кількість надлишкової інформації, що зберігається, буде імплементована можливість запровадити часткові снепшоти. Це буде досягнуто дозволом перевизначити клас снепшоту і явно вказати, які атрибути повинні бути збережені, а також які події цей снепшот агрегує. Завдяки цьому, користувачі отримують гнучкі налаштування снепшотів, які дозволять оптимізувати найбільш затратні операції.

Також, з метою покращення ефективності роботи з даними та оптимізації високонавантажених операцій, буде надана можливість створювати проекції агрегату. Створення проекції буде відбуватись завдяки визначенню спеціального класу проекції, який буде містити в собі інформацію про те, на який агрегат зроблена ця проекція і які події вона репрезентує. Буде надана можливість як використати застосування подій агрегату, так і визначити власну логіку застосування подій. Це дозволить гнучко перетворювати дані в той вигляд, що необхідний для подальшого користування проекцією. Незважаючи на те, що проекція доволі схожа на агрегатний корінь, одна з ключових відмінностей є те, що проекція не дозволяє виконувати з нею операції, що призводять до запису нових подій до сховища даних, а надає доступ до даних лише для читання. Наявність проекцій дозволить системі залишатись швидкою і ефективною навіть при великому обсязі даних, а також позитивно впливатиме на модульність системи.

Для забезпечення довготривалої підтримки системи та адаптації до змін, необхідно реалізувати механізм версіонування. Для цього введемо абстрактний метод «upcast» в базовому класі подій. Цей метод буде відповідальний за перетворення подій зі старих версій у новіші перед їх обробкою в системі.

Описана реалізація забезпечує, що всі компоненти системи працюють тільки з актуальними версіями подій, не вимагаючи зміни даних безпосередньо в базі даних. У випадку відсутності нової версії, метод буде повертати саму подію. Додатково, щоб забезпечити можливість послідовності версій, метод після перетворення буде викликати «upcast» метод утвореної події, що дозволить або отримати цю подію без змін у випадку якщо вона є останньою версію або продовжить послідовність перетворень доки не буде досягнута остання версія. Завдяки цьому при додаванні нових версій буде відсутня необхідність переписувати старі версії, що забезпечить простий механізм оновлень навіть для великої кількості змін.

Для правильної інтеграції всіх вищезазначених можливостей необхідно ретельно продумати яким чином фреймворк буде взаємодіяти зі сховищами даних. Для реалізації цієї взаємодії пропонується створити абстрактний клас репозиторію, що слугуватиме як основа для визначення універсальних інтерфейсів для зберігання та витягування подій, а також для реалізації більш складних механізмів, таких як проекції, версіонування та снепшоти. Це дозволить легко інтегрувати різні сховища даних з мінімальними змінами до основного коду системи. Розробникам буде необхідно реалізувати інтерфейси, що відповідають за збереження та отримання подій, а на їх основі будуть створені методи які дозволяють відновлювати стан агрегату, зберігати нові події та керувати механізмами снепшотів, версій і проекцій. Для коректної роботи з цими механізмами клас буде взаємодіяти з інформацією про класи подій агрегату та їх зв’язками, що дозволить працювати з подіями найбільш оптимальним способом. Така реалізація дозволить ізолювати специфіку різних сховищ від загальної логіки системи.

Окремо варто зазначити, що буде створено інтерфейси, як для синхронної взаємодії зі сховищем даних, так і для асинхронної, що значно розширить спектр застосування фреймворку. Також будуть надані можливості визначити методи початку та закінчення транзакції, що дозволить забезпечити консистентність даних та атомарність операцій.

2.5 Тестування архітектури

Забезпечення ретельного тестування для фреймворку що реалізує патерн «Event Sourcing» є необхідним для створення надійної, стабільної і продуктивної системи, особливо з огляду на складність взаємодії між компонентами. Тестування допоможе переконатися, що система коректно обробляє та зберігає події, забезпечуючи точність відновлення стану системи з подій, а механізми проекцій, версіонування і снепшотів правильно оновлюють стан за допомогою подій. Крім того, наявність детально прописаних тестів може значно підвищити довіру користувачів до фреймворку і бажання його використовувати.

Складності для тестування додає той факт, що більшість класів, що будуть створені – абстрактні, що унеможливлює створення екземпляру такого класу для перевірки його поведінки. Це означає, що для тестування такого класу необхідно створити певну реалізацію, похідну від нього, яка буде реалізовувати усі абстрактні функції. Це може призводити до додаткової складності, оскільки поведінка похідних класів може впливати на результати тестів. Також, тестування базової реалізації повинно враховувати контексти, які можуть виникнути у похідних класах.

Для подолання цих складнощів використовують або моки - тестові об’єкти, які замінюють реальні компоненти програми в процесі тестування, що імітують похідні класи та інтерфейси, або створюють спрощену реалізацію, яка може бути використана для перевірки функціональності [21].

Для тестування нашого фреймворку ми будемо використовувати переважно підхід з моками, оскільки це дозволяє тестувати частини програми ізольовано від інших компонентів системи та імітувати різноманітні сценарії, які можуть бути важко або неможливо відтворити з реальними об'єктами. Це включає випадки з великими даними, крайові випадки або помилкові умови, що забезпечує повноту тестування.

Але, незважаючи на наявні переваги, ізоляція може призвести до того, що інтеграційні проблеми між компонентами не будуть виявлені на ранніх етапах тестування. Також, моки можуть приховувати проблеми в дизайні системи,

створюючи штучну середу, яка надмірно ідеалізована і не відображає реальних умов експлуатації. Тому, для тестування фреймворку будуть використані не тільки таке тестування, а й буде створено окрема реалізація та наведено приклад її використання.

Таке рішення не тільки покращить тестування і дозволить перевірити зручність використання фреймворку на практиці, але ще й буде використовуватись як приклад для розробників, демонструючи основні способи застосування фреймворку і його ключові можливості.

У якості такого прикладу пропонується створити інтеграцію з об'єктно-реляційною системою керування базами даних PostgreSQL [22]. Такий приклад не тільки буде демонструвати і перевіряти роботу фреймворку, а й може бути доданим в кодову базу фреймворку, дозволяючи розробникам використовувати вже готову реалізацію для цієї СКБД. У подальшому приклади можуть бути розширені інтеграціями з іншими СКБД, як реляційними, так і нереляційними, що ще сильніше зменшить поріг входження і дозволить розробникам одразу використовувати фреймворк у своїх системах без необхідності писати інтеграції зі сховищами даних.

2.5 Документація і розповсюдження

Забезпечення користувачів якісною документацією та керівництвами є критично важливим для успіху будь-якого програмного продукту, особливо коли йдеться про фреймворки та інструменти розробки. Якісна документація не лише сприяє легшому та швидшому освоєнню продукту новими користувачами, але й значно знижує кількість потенційних запитів на технічну підтримку, оскільки користувачі можуть самостійно знаходити відповіді на свої питання. Крім того, добре структуровані керівництва та посібники полегшують впровадження фреймворку в проекти, забезпечуючи пояснення ключових концептів та приклади реалізації типових завдань. Це не тільки підвищує задоволеність користувачів, але й сприяє розширенню спільноти розробників, які обирають цей фреймворк для своїх проектів. Інвестування часу та ресурсів у створення

зрозумілої, доступної та комплексної документації відіграє ключову роль у формуванні позитивного сприйняття продукту та його популярність.

Отже, для нашого фреймворку створення документації є важливою задачею. Ця документація надаватиме користувачам всю необхідну інформацію для швидкого старту, включаючи керівництва з встановлення, конфігурації та перших кроків використання фреймворку. Також буде докладно описано основні можливості фреймворку, включаючи його архітектурні особливості, варіанти застосування, і як ці можливості можуть бути використані для розробки програмного забезпечення. Подання чітких прикладів використання і найкращих практик дозволить користувачам максимально ефективно використовувати фреймворк у своїх проектах, забезпечуючи високу продуктивність та якість кінцевого продукту.

Для впровадження документації буде використано сервіс readthedocs [23], що є визнаним інструментом для генерації, хостингу та керування документацією. Він легко інтегрується з репозиторіями коду, автоматично генеруючи документацію при кожному оновленні коду. Це гарантує, що документація завжди актуальна і синхронізована з останніми змінами у коді. Крім того, він підтримує версіонування документації, дозволяючи користувачам переглядати документацію для різних версій фреймворку. Це особливо корисно для користувачів, які можуть використовувати старіші версії. Не менш важливою перевагою є те, що документація, розміщена на цьому сервісі, легко доступна з будь-якого місця та оптимізована для пошукових систем, що робить її зручною для пошуку та використання розробниками та користувачами по всьому світу [24]. Обрання readthedocs для нашого фреймворку забезпечить, що документація буде не тільки корисною та доступною, але й завжди актуальною та легкою у керуванні. Це важливий крок у створенні відкритої та доступної платформи, що сприяє ефективному використанню фреймворку спільнотою розробників.

Не менш важливим за документацію є ефективне та зручне розповсюдження фреймворку. Для цього фреймворк буде зібрано у вигляді окремої бібліотеки, що значно спростить процес його інтеграції в різні проекти

та платформи. Завдяки цьому, ми забезпечимо легкість встановлення фреймворку з використанням популярних менеджерів пакетів, таких як Poetry [25] або pip [26]. Це не тільки спрощує початкове встановлення, але й уніфікує процес оновлення фреймворку до новіших версій, гарантуючи, що користувачі завжди мають доступ до останніх функцій і виправлень помилок.

3 РЕАЛІЗАЦІЯ ФРЕЙМВОРКУ НА ОСНОВІ ЗАПРОПОНОВАНОЇ АРХІТЕКТУРИ

3.1 Опис реалізації та документації

Згідно із запропонованою архітектурою, було створено фреймворк під назвою «pyeventor», який не лише втілює всі необхідні механізми, такі як обробка подій, версіонування, снепшоти та проекції, але й забезпечує високий рівень гнучкості та можливість кастомізації. В процесі розробки особлива увага приділялась адаптації архітектури до особливостей мови Python, для того, щоб максимально спростити роботу розробників і зробити фреймворк більш інтуїтивним та доступним. Надалі будуть описані основні технічні рішення та продемонстровані основні можливості створеного фреймворку.

Ключовими компонентами архітектури є використання класу Protocol [27] для створення абстрактних інтерфейсів та використання декораторів для управління взаємодією між компонентами. Ці два підходи мають свої значні переваги та обрані на основі специфічних потреб та особливостей мови Python.

Protocol використовується замість абстрактних класів, так як він дозволяє використовувати структурну підтипізацію замість жорсткої ієрархії наслідування. Це означає, що клас вважається реалізацією інтерфейсу, якщо він відповідає структурі інтерфейсу, незалежно від того, чи наслідування прямо вказано. Завдяки цьому ми надаємо більшої гнучкості для подальшої кастомізації. Крім цього, згідно з принципами SOLID [28], замість створення одного інтерфейсу, вони були розбиті на дрібніші інтерфейси зі своєю зоною відповідальності, що дозволило полегшити подальше перевикористання коду і написання анотацій.

Декоратори дозволяють додавати додаткову функціональність до існуючих функцій без зміни їхнього основного коду. Завдяки цьому ми надаємо користувачам фреймворку можливість визначати взаємодію компонентів, не

вносячи змін до основного коду, розділяючи бізнес логіку і поведінку системи від поведінки і логіки фреймворку.

Використання Protocol та декораторів разом дозволяє створити фреймворк, який є не тільки масштабованим і гнучким, але й легким у використанні та інтеграції з різними системами та бібліотеками. Це робить його хорошим рішенням для сучасних розробок, які вимагають швидкої адаптації до змінних вимог і технологічних умов.

Основні механізми фреймворку реалізовані за допомогою класів Aggregate та Event. Перший визначає агрегатний корінь та реалізовує інтерфейси, що дозволяють застосувати події до кореня. Другий визначає базову сутність події, на основі якій повинні створюватись будь-які користувацькі події та механізм перетворення події в подію наступної версії. Їх взаємодія регулюється за допомогою декоратору register_handler, який дозволяє визначити метод, який буде викликаний для обробки події певного типу певним агрегатним коренем (додаток А).

Особливу увагу було приділено тому, як вирішувати конфлікти, які будуть з’являтись при використанні наслідування та перевизначені методів. Так, для того, щоб уникнути невизначеної поведінки при наявності більш ніж одного обробника для певної події певним агрегатним коренем, це було заборонено і при спробі зареєструвати більш ніж один обробник буде викинута помилка при запуску системи. Крім цього, для того щоб забезпечити гнучку роботу з наслідуванням і уникнути дуплікації коду, було реалізовано механізм, який дозволяє використати обробник базового класу у випадку, якщо обробник у поточного не було знайдено (додаток Б). Це стосується як агрегатного кореня, так і події. Це реалізовано за допомогою використання бібліотеки inspect [29], яка дозволяє втілити механізми рефлексії у мові Python та доступитися до порядку вирішення методів та анотацій методів для подальшого визначення поведінки. Завдяки цій бібліотеці також реалізована можливість визначати обробник як у корені, так і у самої події, підставляючи аргументи в правильному

порядку у обробник в залежності від того, якому базовому класу належить обробник (додаток В).

Механізм проекції було втілено за допомогою класу Projection, який відрізняється від агрегатного кореня тим, що він не зберігає події, що були до нього застосовані, тим самим не надаючи можливість в подальшому зберегти їх у сховище даних. Завдяки цьому досягається безпечна можливість створення і застосування подій до проекції для перевірки гіпотез у системи без можливості порушити консистентність даних системи. Для визначення даних, які проекція уособлює використовується декоратор projection, який визначає клас агрегатного кореня та класи подій, на які робиться проекція (додаток Г). Це дозволяє проігнорувати всі інші події у подальшій роботі з проекцією та автоматично визначити обробники подій, унаслідувавши їх з наявних, при цьому залишаючи можливість перевизначити їх.

Снепшоти реалізовані як наслідування від базового класу подій з додаванням окремого інтерфейсу, що забезпечує значні переваги у плані інтеграції та управління станом системи. Таке рішення дозволяє снепшотам використовувати всі існуючі механізми обробки подій, включно з розповсюдженням та збереженням. За використання снепшотів відповідає агрегатний корінь, вказуючи який клас снепшотів повинен використовуватись і раз на скільки подій він повинен бути створений (додаток Ґ). Створений снепшот зберігається разом з іншими подіями.

Початково розглядалася можливість впровадження часткових снепшотів, які б зберігали лише частину стану системи. Однак, цей підхід було визнано ризикованим, оскільки часткові снепшоти можуть призвести до проблем з незрозумілою синхронізацією даних і впливом на інші події. Відсутність чіткості у визначенні границь снепшотів може ускладнити відновлення системи або її частин після збоїв.

Щоб вирішити вищезазначені ризики, було впроваджено снепшоти проекцій. Ці снепшоти створюються для специфічних проекцій, що є безпечним, оскільки гарантує повне відтворення стану для сутності без ризику

неправильного застосування подій. Так як проекція не здатна записувати події, то за збереження снепшотів проекції відповідає агрегатний корінь, отримуючи данні про те, які снепшоти необхідні під час реєстрації проекції декоратором (додаток Д).

В архітектурі нашого фреймворку значна увага приділена забезпеченню гнучкості у зберіганні та управлінні даними, що є критичним аспектом для патерну, що розглядається. Через використання інтерфейсів для визначення способів збереження і відновлення даних, користувачі мають можливість налаштувати поведінку системи згідно зі своїми потребами. Так, користувачі повинні визначити лише методи для збереження та отримання подій, на основі яких їм будуть надані методи, які дозволяють працювати з агрегатами, проекціями та снепшотами.

Окремою важливою можливістю є визначення методів для зберігання і отримання снепшотів. Це дозволяє користувачам розмежувати місця зберігання снепшотів та звичайних подій, що може значно пришвидшити відновлення стану системи та зменшити вплив на загальну продуктивність при читанні даних

Агрегати та події в системі надають можливості для налаштування типів ідентифікаторів та механізмів їх впорядкування (додаток Е). Це надає користувачам додаткову свободу визначення того, як ідентифікувати та сортувати події, не обмежуючи певним типом, а дозволяючи впровадити той, що найбільш підходить для обраного сховища даних.

У процесі розробки фреймворку було впроваджено підтримку асинхронності, що є ключовим аспектом для забезпечення високої продуктивності та масштабованості сучасних застосунків. Це було реалізовано шляхом введення спеціалізованих асинхронних інтерфейсів, які дозволяють розробникам вибирати між синхронними та асинхронними варіантами роботи зі структурами даних і взаємодії з зовнішніми системами. Водночас, інтеграція асинхронних інтерфейсів вимагала перевикористання значної частини існуючого синхронного коду, адаптуючи його під асинхронні операції. Хоча багато компонентів вдалося адаптувати з мінімальними змінами, частина

функціональності все ж таки вимагала дублювання коду для забезпечення сумісності між синхронними та асинхронними версіями.

Для демонстрації можливостей фреймворку, були розроблені плагіни, які імплементують різні способи збереження даних. Це не тільки показує можливості фреймворку, але й забезпечує користувачів необхідними інструментами для початку роботи з фреймворком.

Одним із найпростіших плагінів є реалізація збереження даних у пам'яті. Цей плагін слугує як зразок того, як можна втілити інтерфейси, визначені в нашому фреймворку. Зберігання даних у пам'яті не призначене для використання в реальних застосунках через відсутність персистентності, але є надзвичайно корисним для розробки, тестування та демонстрації можливостей системи. Цей підхід дозволяє швидко розпочати роботу з фреймворком, експериментувати з його API та оцінити поведінку системи без необхідності конфігурації зовнішніх сховищ даних.

Для більш складних і реалістичних сценаріїв була забезпечена підтримка асинхронного збереження даних у PostgreSQL [22]. Цей плагін демонструє імплементацію асинхронних інтерфейсів та втілює більш складний кейс використання, що є придатним для розробників, які планують використовувати фреймворк. Завдяки асинхронній обробці запитів до бази даних, цей плагін забезпечує високу продуктивність і масштабованість, що є критично важливим для додатків з великим об'ємом даних або високою частотою транзакцій.

Обидва ці плагіни слугують як довідковий матеріал і приклад для розробників, які бажають реалізувати власні стратегії збереження даних з використанням створеного фреймворку. Таким чином, включення цих плагінів підвищує гнучкість, доступність та функціональність системи, сприяючи її широкому впровадженню та успіху.

Для забезпечення високої якості та стабільності роботи фреймворку, були створені юніт тести, які покривають всі аспекти його функціональності. Ці тести особливо важливі, оскільки дозволяють перевірити кожен компонент системи незалежно від інших, що дозволило виправити наявні помилки на ранніх етапах

розробки. У процесі тестування використовувались моки для абстракцій, що дозволило ізолювати тести від зовнішніх залежностей і зосередити увагу на тестуванні стандартних реалізацій. Такий підхід гарантує, що основний функціонал фреймворку працює належним чином і відповідає встановленим вимогам. Використання моків для абстрактних інтерфейсів у тестах також сприяє чіткому визначенню контрактів між різними частинами системи, що підвищує якість коду і зменшує ризик регресійних помилок при додаванні нових функцій або рефакторингу існуючих. Ці тести є не лише засобом перевірки правильності роботи коду, але й служать документацією, яка показує, як очікується, що система буде реагувати на різні вхідні дані та умови використання.

Так як створення всебічної та зрозумілої документації було одним із пріоритетних завдань у процесі розробки фреймворку, то вона не обмежується лише цим тестуванням, оскільки воно важке для розуміння і потрібно більше для підтримки фреймворку і його подальшого вдосконалення аніж для користувачів. Для майбутніх користувачів були створені приклади, які демонструють повний робочий процес для кожного з плагінів. Ці приклади допомагають користувачам швидко зрозуміти, як застосовувати фреймворк до власних проектів, ілюструючи інтеграцію з різними системами зберігання даних та використання асинхронних можливостей. Крім цього, була створена докладна документацію на платформі ReadTheDocs [23], де користувачі можуть знайти всебічні описи можливостей фреймворку, доповнені відповідними прикладами коду (рис. 3.2.1). Ця платформа забезпечує легкий доступ та навігацію по документації, що робить процес навчання та інтеграції фреймворку максимально зручним.

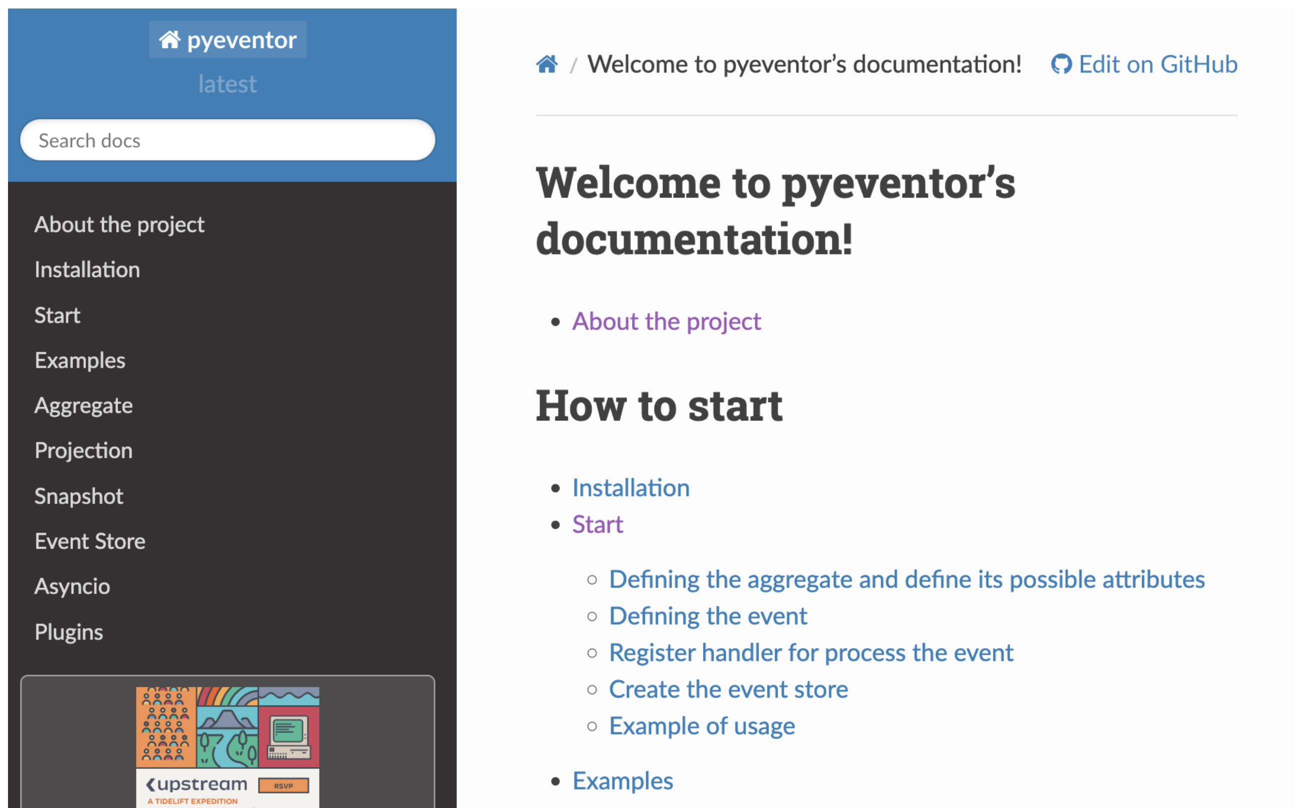

Рисунок 3.1.1 – Документація створеного фреймворку [30]

Завдяки цій комплексній документації, фреймворк не тільки доступний для широкого кола розробників, але й забезпечує усі необхідні ресурси для його ефективного впровадження та використання в реальних проектах. Таким чином, були не тільки реалізовані заплановані можливості, але й забезпечено їх прозорість та доступність для кінцевих користувачів

3.2 Порівняння з існуючими рішеннями

Для детального аналізу та оцінки фреймворку, важливо провести порівняльний аналіз з іншими популярними рішеннями. Цей підхід дозволить зрозуміти ключові переваги, недоліки фреймворку. Для порівняння були обрани популярні фреймворки на різних мовах програмування, а саме:

- eventsourcing на Python [31]
- EventFlow на .Net [32]
- Castore на TypeScript [33]

Під час порівняння будуть складені таблиці, які дозволять наочно оцінити різницю у функціонали між різними фреймворками, та для деяких сценаріїв

наведені приклади коду, що дозволить оцінити різницю між використанням фреймворків.

Оскільки швидкість виконання може значно варіюватися в залежності від конкретних імплементацій сховищ даних, було вирішено зосередити аналіз на функціональності фреймворків та на легкості їх застосування, що дозволить нам зрозуміти, як наш фреймворк відповідає потребам розробників в різних сценаріях використання і як він спрощує реалізацію типових задач. Основна увага буде приділятись таким аспектам, як:

- Створенні подій і агрегатного кореня
- Створенні снепшотів
- Створенні проекцій
- Асинхронності
- Кастомізації
- Наявності плагінів і додаткового функціоналу

Робота з агрегатним кореням і подіями містить схожий функціонал в усіх фреймворках, що було проаналізовано (таблиця 3.2.1). Головною відмінністю є підтримка нашим фреймворком наслідування сутностей, що значно спрощує імплементацію складних систем. Серед інших, EventFlow виділяється тим, що надає можливість працювати з версіями подій шляхом втручання у сховища даних, надаючи для цього функціонал, що є допустим рішенням, хоч і концептуально не відповідає патерну. Castore виділяється невдалим рішенням для застосування подій, пропонуючи єдиний метод для визначення обробників всіх подій одразу, що призводить до ускладнення кодової бази. Eventsourcing на відміну від них, навпаки, надає додатковий функціонал у вигляді версіонування не тільки подій, але і агрегатів, але реалізація цього версіонування передбачає створення методів у події, не створюючи для цього окремі класи, що може бути як недоліком, так і перевагою. На прикладі (додаток Є) можна побачити, що хоч імплементація eventsourcing є більш компактною, відбувається втрата знань про події попередньої версії, що створює незручності для довготривалої підтримки системи.

Фреймворк / Функціонал	pyeventor	eventsourcing	EventFlow	Castore
Створення агрегату	+	+	+	+
Створення події	+	+	+	+
Створення обробника у агрегаті	+	+	+	+/-
Створення обробника у події	+	+	+	+/-
Можливість версіонування подій	+	+	+/-	-
Можливість версіонування агрегату	-	+	-	-
Кастомізація айді агрегатів	+	+	+	-
Кастомізація впорядкованності подій	+	-	-	-
Підтримка наслідування агрегатів	+	-	-	-
Підтримка наслідування подій	+	-	-	-

Таблиця 3.2.1 – Порівняння роботи з агрегатним коренем і подіями у різних фреймворках

Створення снепшотів (таблиця 3.2.2) реалізовано в усіх фреймворках окрім Castore, але тільки в нашому наявна можливість визначити снепшоти для проекцій, що є важливою перевагою, так як дозволяє значно пришвидшити і оптимізувати роботу з критичними компонентами системи, що може стати ключовим приводом для обрання фреймворку. Також хочеться відмітити, що в усіх реалізаціях наявна можливість створення снепшоту раз на певну кількість подій, але в жодному не доступне створення снепшотів при настанні якоїсь події, що є можливістю для подальшого розвитку фреймворку.

Фреймворк / Функціонал	pyeventor	eventsourcing	EventFlow	Castore
Створення снепшоту агрегата	+	+	+	-
Створення снепшоту проекції	+	-	-	-
Створення снепшоту раз на певну кількість подій	+	+	+	-
Створення снепшоту по трігеру	-	-	-	-
Версіонування снепшоту	+	+	+	-

Таблиця 3.2.2 – Порівняння функціоналу снепшотів у різних фреймворках

Функціонал проекцій містить найменше можливостей, але реалізований лише в двох фреймворках (таблиця 3.2.3). Pyeventor та EventFlow надають цей функціонал в повній мірі, але якщо EventFlow вимагає перевизначити обробники

для кожної події, pyeventor здатен використовувати обробники агрегату на який робиться проекція, у випадку відсутності власних обробників, що зменшує обсяг коду необхідного для реалізації проекції.

Фреймворк / Функціонал	pyeventor	eventsourcing	EventFlow	Castore
Створення проекції	+	-	+	-
Обробники подій для проекції	+	-	+	-

Таблиця 3.2.3 – Порівняння функціоналу проекцій у різних фреймворках

Робота зі сховищами даних є найбільш критичною роботою фреймворку і саме тут ми можемо побачити найбільше відмінностей у реалізації (таблиця 3.2.4). Наявність готових рішень – єдиний функціонал, який спільний для всіх фреймворків. Найсильніше виділяється в цьому eventsourcing, який містить готову реалізацію для семи різних сховищ даних. У порівнянні з іншими, pyeventor лише наводить користувачам готові приклади передбачаючи власну імплементацію як більш правильний підхід. Також, хочеться відмітити особливість eventsourcing, який взаємодіє лише з подіями, пропонуючи реалізовувати методи по застосуванню подій самім користувачам, що потребує багато додаткових зусиль розробників. Можливість створення власної імплементації і розділене збереження сутностей є лише в окремих фреймворках, хоча і здатне значно покращити продуктивність системи. Таким чином, можна сказати, що більшість фреймворків роблять упор саме на функціональність роботи з подіями, оминаючи реалізації сховищ даних або надаючи їх в обмежено варіанті. Наявний функціонал pyeventor значно розширює ці можливості і є значною конкурентною перевагою фреймворку.

Фреймворк / Функціонал	pyeventor	eventsourcing	EventFlow	Castore
Робота з проекціями	+	-	+	-
Відокремлене збереження снепшотів	+	-	+	-
Відокремлене збереження проекцій	+/-	-	+	-
Доступ до стану у певний момент часу	+	+	+/-	-
Наявність готових імплементацій	+	+	+	+
Можливість створення своєї імплементації	+	-	-	+

Таблиця 3.2.4 – Порівняння функціоналу сховища даних у різних фреймворках

Порівнюючи додаткові можливості фреймворків (таблиця 3.2.5) можна побачити, що лише pyeventor втілює асинхронність та надає асинхронні інтерфейси для взаємодії зі сховищами даних. Однак, не можна не зауважити додатковий функціонал eventsourcing, який вміє шифрувати, стискати і кешувати дані, що може бути ефективним і корисним для певних ситуацій. Таким чином, хоч додаткові можливості і не є обов’язковими, їх впровадження могло би покращити репозиторій і надати розробників додаткових причин для його обрання.

Фреймворк / Функціонал	pyeventor	eventsourcing	EventFlow	Castore
Асинхронність	+	-	-	-
Шифрування	-	+	-	-
Компресія даних	-	+	-	-
Кеш	-	+	-	-

Таблиця 3.2.5 – Порівняння додаткового функціоналу у різних фреймворках

Відповідно до проведених порівнянь можна дійти висновку, що наш фреймворк виокремлюється через свою адаптивність і спрямованість на кастомізацію та інтеграцію з іншими системами. Основні можливості патерну реалізовані в усіх фреймворках, але деякі механізми, що дозволяють покращити ефективність роботи з патерном або не реалізовані, або реалізовані в обмеженому обсязі. Не менш важливим є зауважити, що інші фреймворки надають готові системи з якими важко інтегруватись, або які вимагають встановлення додаткових залежностей, тоді як наш фреймворк є більше інструментом, що може бути легко адаптований до різноманітних вимог і потреб проекту та містить мінімальний набір сторонніх залежностей. Тим не менш, деякі фреймворки містять більш різноманітний функціонал, що дозволяє втілити додаткові можливості та містять більше доступних інтеграцій, що може робити їх привабливими для розробників, які не планують використовувати кастомізацію, а хочуть отримати готове рішення. Таким чином, хоч наш фреймворк і задовольняє вимогам та має низку переваг над іншими, його може бути покращено шляхом додавання функціоналу та створення додаткових

реалізацій, що дозволить розробникам обирати між використанням вже створених імплементацій або написанням власної, що в перспективі ще сильніше знизить поріг входження та спростить розробку складних систем.

3.3 Майбутні удосконалення та дослідницькі напрямки

Незважаючи на вдалу реалізацію всіх запланованих можливостей, в процесі роботи над фреймворком було виявлено низку проблем, які варіюються від технічних викликів до більш фундаментальних питань. Ці проблеми не лише виявили деякі обмеження наших поточних рішень, але й відкрили широкий спектр можливостей для подальших досліджень і вдосконалення. Кожна з цих проблем відкриває двері до нових можливостей для удосконалення існуючих рішень і розробки нових інструментів, які можуть значно покращити якість та ефективність майбутніх версій нашого фреймворку.

Так, одним з недоліків поточної архітектури є необхідність дублювати код для підтримки асинхронних і сінхроних інтерфейсів. Це призводить до ускладнення коду і підвищує ризик виникнення неконсистентності. Одним із можливих шляхів вирішення проблеми може стати впровадження абстракцій, які дозволять уніфікувати логіку обробки для асинхронних та синхронних операцій.

Іншим недоліком є штучні обмеження на часткові снепшоти, які довелось впровадити в процесі розробки. Первісний план передбачав можливість робити часткові снепшоти агрегатного кореня, але від цього довелось відмовитись заради забезпечення цілісності даних. В контексті розробки систем на основі подій, безпека і надійність даних є вирішальними. Невдала реалізація часткових снепшотів може призвести до серйозних проблем з консистентністю даних, що в свою чергу може негативно вплинути на весь додаток. Забезпечення коректної реалізації цього функціонала потребує написання аналізаторів, які зможуть перевірити чи певний частковий снепшот не буде порушувати цілісність та окремих механізмів контролю, які би убезпечили юзерів від небезпечного використання цього функціоналу. Вирішення цих викликів не тільки значно покращить функціональність фреймворку, але й забезпечить користувачам ще більшу гнучкість та контроль над обробкою та зберіганням даних.

Також під час розробки була виявлена концептуальна проблема, що полягає в тому, що у подійно-орієнтованих системах не існує постійного стану сутності у традиційному розумінні; замість цього стан визначається послідовністю подій, які були застосовані до сутності. Ця особливість створює труднощі, коли справа доходить до фільтрації або вибірки даних на основі поточного стану сутності. Оскільки стан сутності не існує в явному вигляді і має бути відновлений із послідовності подій, звичайні методи фільтрації, які використовуються в традиційних базах даних, не можуть бути застосовані безпосередньо. Реалізація фільтрів, які могли б ефективно використовувати таку архітектуру для підтримки складної бізнес логіки, вимагає значних досліджень. Це не просто технічна проблема, а широке поле для аналізу можливостей і обмежень подійно-орієнтованих систем. Впровадження такого механізму або принципів його побудови в різних системах не тільки підвищить ефективність нашого фреймворку, але й відкриє нові можливості для розробки більш гнучких і масштабованих систем, здатних ефективно обробляти складні бізнес процеси.

Незважаючи на те, що ці проблеми не були вирішенні під час реалізації, вони самі по собі стають важливим результатом роботи, так як відкривають великий простір для подальших досліджень і розвитку, підкреслюючи напрямки, в яких може розвиватися фреймворк. Подальші дослідження в цих напрямках не лише підвищать якість і надійність нашого фреймворку, а й можуть вплинути на сам патерн та способи його застосування в цілому.

ВИСНОВКИ

Відповідно до задачі у даній роботі було проаналізовано патерн «Event Sourcing» та розглянуто теоретичний матеріал, що до нього відноситься. Були зроблені висновки стосовно переваг, недоліків патерну та способів його застосування. Крім того, були оглянуті існуючі рішення та їх наявні обмеження. Цей аналіз дозволив краще зрозуміти патерн та наявну проблематику у його застосуванні.

На основі даного аналізу була розроблена архітектура фреймворку з акцентом на гнучкість та реалізацію. Ця архітектура була спроектована з урахуванням потреб реальних проектів та здатності забезпечити ефективну підтримку патерну «Event Sourcing». Були визначені основні принципи, на яких буде побудовано фреймворк, а також чіткий список функціональних та нефункціональних вимог, які використовувались під час розробки.

Згідно з визначеною архітектурою було розроблено фреймворк [34], який відповідає всім визначеним вимогам та створена документація до нього. Створена реалізація була порівняна з існуючими рішеннями для валідації результату і визначенні переваг та недоліків рішення. Порівняння показало, що розроблений фреймворк містить низку переваг порівняно з існуючими рішеннями та містить функціонал, який відсутній у інших, що повинно сприяти ефективному використанню паттерну та підвищити його ефективність. Разом з цим були виявлені недоліки та можливі покращення фреймворку.

Крім технічних недоліків, під час розробки також було виявлено ряд концептуальних проблем патерну які потребують подальшого вдосконалення та дослідження. Ці проблеми можуть стати об'єктом подальших досліджень та розробок у майбутніх роботах.

Основним досягненням є створений фреймворк, який відзначається низкою переваг порівняно з аналогами. Це відкриває перспективи для ефективного застосування паттерну у розробці програмного забезпечення. Паралельно з цим, визначені напрямки майбутніх досліджень у даній області, що

можуть сприяти подальшому вдосконаленню фреймворку та створенню нових підходів у роботі з патерном «Event Sourcing»

Список використаної літератури

1. Рейтинг мов програмування 2023 [Електронний ресурс] // Режим доступу: https://dou.ua/lenta/articles/language-rating-2023/
2. Breaking Down The Numbers: How Much Data Does The World Create Daily in 2024? [Електронний ресурс] // Режим доступу: https://edgedelta.com/company/blog/how-much-data-is-created-per-day/
3. Contemporary Audit Regulation – Going Global! / *Handbook of Key Global Financial Markets, Institutions, and Infrastructure*. / Humphrey C., Loft A. // 2013. – ст. 333–343.
4. Event Sourcing [Електронний ресурс] / Martin Fowler / 2005 // Режим доступу: https://martinfowler.com/eaaDev/EventSourcing.html
5. An Empirical Characterization of Event Sourced Systems and Their Schema Evolution - Lessons from Industry/ Michiel Overeem, Marten Spoor, Slinger Jansen, Sjaak Brinkkemper // 2021 – ст. 5-7 – Режим доступу: https://arxiv.org/pdf/2104.01146
6. Domain-driven design practice — Domain Events [Електронний ресурс] / Chaojie Xiao / 2022 // Режим доступу: https://medium.com/@chaojie.xiao/domain-driven-design-practice-domain-events-15b38f3c58fc/
7. What is Event-Driven Architecture? [Електронний ресурс] // Режим доступу: https://hazelcast.com/glossary/event-driven-architecture/
8. Improving observability in Event Sourcing systems / Stanley Lima, Jaime Correia, Filipe Araujo, Jorge Cardoso // 2021 – ст. 2 – Режим доступу: https://www.researchgate.net/publication/352267210_Improving_observability_in_Event_Sourcing_systems
9. Event sourcing pattern [Електронний ресурс] // Режим доступу: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/event-sourcing.html

10. Improving observability in Event Sourcing systems / Stanley Lima, Jaime Correia, Filipe Araujo, Jorge Cardoso // 2021 – ст. 5 – Режим доступу: https://www.researchgate.net/publication/352267210_Improving_observability_in_Event_Sourcing_systems
11. Snapshotting Strategies [Електронний ресурс] / Oskar Dudycz / 2021 // Режим доступу: https://www.eventstore.com/blog/snapshotting-strategies
12. When To Take A Snapshot [Електронний ресурс] / Oskar Dudycz / 2021 // Режим доступу: https://www.eventstore.com/blog/snapshots-in-event-sourcing#when-to-take-a-snapshot
13. Migrating a production database without any downtime [Електронний ресурс] / Denis Stebunov / 2021 // Режим доступу: https://teamplify.com/blog/zero-downtime-DB-migrations
14. Event Sourcing: Events Evolution, Versioning, and Migration [Електронний ресурс] / 2021 // Режим доступу: https://valerii-udodov.com/posts/event-sourcing/events-versioning
15. Event Store [Електронний ресурс] // Режим доступу: https://www.eventstore.com/
16. Scaling Event Sourcing for Netflix Downloads, Episode 1 [Електронний ресурс] / 2017 // Режим доступу: https://netflixtechblog.com/scaling-event-sourcing-for-netflix-downloads-episode-1-6bc1595c5595
17. The Reasons for Python’s Popularity in 2024 [Електронний ресурс] / Matheus Jacques/ 2024 // Режим доступу: https://www.scalablepath.com/python/python-popularity
18. An insight into the importance of RequirementsEngineering / Nikhil T. More, Bhushan S. Sapre, Pramila M. Chawan // 2011 – ст. 3 – Режим доступу: https://www.researchgate.net/publication/316475303_An_Insight_into_the_Importance_of_Requirements_Engineering

19. Functional and Nonfunctional Requirements: Specification and Types [Електронний ресурс] / 2023 // Режим доступу: https://www.altexsoft.com/blog/functional-and-non-functional-requirements-specification-and-types
20. PEP 8 – Style Guide for Python Code [Електронний ресурс] / 2013 // Режим доступу: https://peps.python.org/pep-0008
21. Intra-Class Testing of Abstract Class Features/ Peter J. Clarke, Djuradj Babich, Tariq M. King, James F. Power // 2007 – ст. 2 – Режим доступу: https://www.researchgate.net/publication/4298344_Intra-Class_Testing_of_Abstract_Class_Features
22. Офіційна сторінка PostgreSQL [Електронний ресурс] // Режим доступу: https://www.postgresql.org
23. Офіційна сторінка readthedocs [Електронний ресурс] // Режим доступу: https://about.readthedocs.com/
24. How to do search engine optimization (SEO) for documentation projects [Електронний ресурс] // Режим доступу: https://docs.readthedocs.io/en/stable/guides/technical-docs-seo-guide.html
25. Офіційна сторінка Poetry [Електронний ресурс] // Режим доступу: https://python-poetry.org/
26. Офіційна сторінка pip [Електронний ресурс] // Режим доступу: https://pip.pypa.io/en/stable
27. Офіційна документація Protocol [Електронний ресурс] // Режим доступу: https://peps.python.org/pep-0544/
28. Design Principles and Design Patterns / Robert C. Martin // 2000.
29. Офіційна сторінка inspect [Електронний ресурс] // Режим доступу: https://docs.python.org/uk/3/library/inspect.html#module-inspect
30. Сторінка pyeventor на сайті readthedocs [Електронний ресурс] // Режим доступу: https://pyeventor.readthedocs.io/en/latest/index.html

31. Офіційна сторінка eventsourcing [Електронний ресурс] // Режим доступу: https://eventsourcing.readthedocs.io/en/stable/index.html
32. Офіційна сторінка EventFlow [Електронний ресурс] // Режим доступу: https://docs.geteventflow.net/FAQ.html
33. Офіційна сторінка Castore [Електронний ресурс] // Режим доступу: https://castore-dev.github.io/castore/
34. Репозиторій pyeventor [Електронний ресурс] // Режим доступу: https://github.com/darkllen/pyeventor

Додаток А
(довідниковий)

```
from pyeventor.aggregate import Aggregate
from pyeventor.event import Event
from pyeventor.decorator import register_handler

class CustomEventA(Event[int, None]):
    ...

class CustomAggregate(Aggregate[int]):

    @register_handler(CustomEventA) # register from aggregate
    def custom_handler(self, event: CustomEventA):
        ...

class CustomEventC(Event[int, None]):

    @register_handler(CustomAggregate) # register from event
    def custom_handler(self, aggregate: CustomAggregate):
        ...

aggregate = CustomAggregate()
event_a = CustomEventA()
event_b = CustomEventB()

aggregate.apply(event_a)
aagregate.apply(event_b)
```

Додаток Б

(довідниковий)

```
@classmethod
def get_handler(
    cls, aggregate_class: Type[Aggregate], event_class: Type[Event]
) -> Optional[Callable]:
    for next_class_base in inspect.getmro(aggregate_class):
        class_handlers = cls.get_aggregate_handlers(next_class_base)
        for next_event_base in inspect.getmro(event_class):
            if handler := class_handlers.get(next_event_base):
                return handler
    return None
```

Додаток В

(довідниковий)

```
class ApplyI(Protocol):
   def _apply_without_saving(self, event: Event) -> "ApplyI":
      if handler := EventHandler.get_handler(type(self), type(event)):
         for _, v in inspect.signature(handler).parameters.items():
            if issubclass(v.annotation, self.__class__):
               handler(event, self)
               return self
            if issubclass(v.annotation, Event):
               handler(self, event)
               return self
      raise HandlerException(f"handler for {event.__class__.__name__} not found")
```

Додаток Г

(довідниковий)

```python
from pyeventor.aggregate import Aggregate, Projection
from pyeventor.event import Event
from pyeventor.decorator import register_handler, projection

class CustomEventA(Event[int, None]):
    ...

class CustomEventB(Event[int, None]):
    ...

class CustomAggregate(Aggregate[int]):

    @register_handler(CustomEventB)
    def custom_handler(self, event: CustomEventB):
        ...

    @register_handler(CustomEventA)
    def special_custom_handler(self, event: CustomEventA):
        ...

@projection(CustomAggregate, [CustomEventA])
class CustomProjection(Projection):

    def _init_empty_attributes(self):
        ...

    @register_handler(CustomEventA)
    def redefine_apply(self, event: CustomEventA):
        # in case of absence of redefined,
        # CustomAggregate will be used for resolving the handling
```

Додаток Ґ

(довідниковий)

```
from pyeventor.aggregate import Aggregate, Projection
from pyeventor.event import Event
from pyeventor.decorator import register_handler, projection

class CustomEventA(Event[int, None]):
    ...

class CustomEventB(Event[int, None]):
    ...

class CustomAggregate(Aggregate[int]):
    SnapshotClass = JsonSnapshot # define the snapshot class, JsonSnapshot is a default one

    @register_handler(CustomEventB)
    def custom_handler(self, event: CustomEventB):
        ...

    @register_handler(CustomEventA)
    def special_custom_handler(self, event: CustomEventA):
        ...

aagregate = CustomAggregate(auto_snapshot_each_n=2) # each two events, snapshots will be created
event_a = CustomEventA()
event_b = CustomEventB()

aggregate.apply(event_a)
aagregate.apply(event_b) # after that event, snapshot will be created
aggregate.apply(event_a)

storage = CustomEventStore()
storage.save(aggregate) # snpahot is saved among with other events
storage.load(aggregate.id) # loaded with snapshot (only the last event will be applied as it was done after the snpahot)
storage.load(aggregate.id, from_snapshots=False) # loaded without snapshots
```

Додаток Д
(довідниковий)

```
def projection(aggregate_class: Type[Aggregate], events: list[Type[Event]]):
    def wrapper(projection_class: Type[Projection]):
```

```
if (
    projection_class.SnapshotClass
    not in aggregate_class.projection_snapshot_classes
):
    aggregate_class.projection_snapshot_classes.append(
        projection_class.SnapshotClass
    )
```

Додаток Е
(довідниковий)

```
class CustomEvent(Event[int, datetime]):
```

```
    def _sequence_generate(self) -> datetime:
        # define the generator for the event sequence
        # according to that it will define the order of events later
        # it should be consistent and provide an order for all the events of the Aggregate
        return datetime.now()


event = CustomEvent(2) # _sequence_generate is ued for sequence_order
assert event.data == 2
```

Додаток Є

(довідниковий)

```
# pyeventor
class CustomEvent(Event):
    ...
```

```
    def upcast(self):
        return CustomEventV2()

class CustomEventV2(Event):
    ...


# eventsourcing
class CustomEvent(DomainEvent):
    class_version = 2

    @staticmethod
    def upcast_v1_v2(state):
        ...
```