МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет
імені Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних
інформаційних систем, д–р техн. наук, проф.
_____Ю. П. Кондратенко
«____»_____2022 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

КОМП'ЮТЕРНА ГРА У ЖАНРІ STORY GAMES ІЗ ПІДТРИМКОЮ ПРОГРАМНОГО ІНТЕРФЕЙСУ

Спеціальність 122 «Комп'ютерні наук»

122 – БКР – 402.21810216

Виконав студент 4–го курсу, групи 402
_____ ***О. В. Кравець***
«___» _____ 2022 р.

Керівник: канд. техн. наук, доцент (б.в.з)
_____***М. Л. Дворецький***
«___» _____ 2022 р.

Миколаїв – 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет ім. Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

Освітньо-кваліфікаційний рівень **бакалавр**

Спеціальність **122 «Комп'ютерні науки»**
(шифр і назва)

Галузь знань **12 «Інформаційні технології»**
(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф. _____Ю. П. Кондратенко
«____»_____2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

Видано студенту групи 402 факультету комп'ютерних наук

_____Кравцю Олександру Валерійовичу_____.
(прізвище, ім'я, по батькові студента)

1. Тема кваліфікаційної роботи «Комп'ютерна гра у жанрі Story games із підтримкою програмного інтерфейсу».
Керівник роботи Дворецький Михайло Леонідович, канд. техн. наук, доцент (б.в.з.)

Затв. наказом Ректора ЧНУ ім. Петра Могили від «07» 12 2021 р. № 318

2. Строк представлення кваліфікаційної роботи студентом «__» _____ 20__ р.

3. Очікуваний результат роботи: back-end сервіс з підтримкою REST API та Telegram бот.

4. Перелік питань, що підлягають розробці (зміст пояснювальної записки):

- створення back-end сервісу на мові програмування Go;
- створення бота на основі Telegram API;

5. Перелік графічних матеріалів: сторінок – 69, таблиць – 0, рисунків – 24, посилань – 26, додатків – 2, презентація.

6. Завдання до спеціальної частини: «Охорона праці при користуванні комп'ютерами».

7. Консультанти:

Розділ	Прізвище, ініціал та посада консультанта	Підпис
Спеціальна частина з охорони праці	Алексеева А.А., старший викладач	

Керівник роботи ___ канд. техн. наук, доцент (б.в.з.) Дворецький М._____
(наук. ступінь, вчене звання, прізвище та ініціали)

_____
(підпис)

Завдання прийнято до виконання _Кравець О. В._____
(прізвище та ініціали)

_____
(підпис)

Дата видачі завдання «_23_» ___листопада_____ 2021 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: «Комп'ютерна гра у жанрі Story games із підтрімкою програмного інтерфейсу»

№	Найменування роботи	Початок	Закінчення	Примітки
1	Подання заяви на затвердження теми та керівників БКР	02.11.2021	02.11.2022	виконано
2	Отримання завдання на виконання БКР	25.11.2021	25.11.2022	виконано
3	Складання календарного плану роботи на весь період виконання БКР	05.12.2021	10.12.2021	виконано
4	Отримання завдання на переддипломну практику	23.05.2022	23.05.2022	виконано
5	Проходження переддипломної практики, збір та аналіз матеріалів до БКР	24.05.2022	02.06.2022	виконано
6	Розробка звіту з переддипломної практики	03.06.2022	04.06.2022	виконано
7	Виконання БКР: аналіз сфери, розробка мобільного застосунка, тестування	25.02.2022	28.05.2022	виконано
8	Попередній захист БКР на засіданні комісії кафедри	31.05.2022	31.05.2022	виконано
9	Доробка та остаточне оформлення БКР	01.06.2022	11.06.2022	виконано
10	Подання БКР рецензенту	17.06.2022	17.06.2022	виконано
11	Подання БКР, її електронної копії та інших документів (відгуку, рецензії) до захисту	24.06.2022	24.06.2022	
12	Захист БКР перед екзаменаційною комісією (ЕК)	29.06.2022	29.06.2022	

Розробив студент _____Кравець О. В._____ _____
(прізвище та ініціали) (підпис)

Керівник роботи канд. техн. наук, доцент (б.в.з.) Дворецький М. Л _____
(наук. ступінь, вчене звання, прізвище та ініціали) (підпис)

«____» _____ 202__ р.

АНОТАЦІЯ

бакалаврської кваліфікаційної роботи студента групи 402

ЧНУ ім. Петра Могили

Кравця Олександра Валерійовича

Тема: «Комп'ютерна гра у жанрі Story games із підтримкою програмного інтерфейсу»

Об’єкт роботи – процес створення комп’ютерної гри в жанрі Story games.

Предмет роботи – засоби та інструменти для створення back-end сервісів та чат ботів, архітектура сучасних додатків.

Метою бакалаврської кваліфікаційної роботи є підвищення зручності процесу читання історій за рахунок надання користувачу можливості впливати на сюжет історій шляхом створення гри у жанрі Story games із підтримкою програмного інтерфейсу.

Робота складається з фахового розділу і спеціальної частини з охорони праці. Пояснювальна записка складається зі вступу, трьох розділів, висновків та додатків.

У першому розділі проаналізуємо та оберемо стек технологій для створення додатку. Розглянемо плюси та мінуси обраних технологій.

У другому розділі створимо архітектуру для back-end сервісу та Telegram бота, розглянемо доцільність використання гексагональної архітектури та Domain Driven Design в системі, оптимізуємо систему шляхом використання бази даних Redis.

У третьому розділі розглянемо користувацький інтерфейс Telegram бота та програмний інтерфейс back-end сервісу.

В останньому спеціальному розділі було розглянуто нормативну базу охорони праці при користуванні комп’ютерами.

Сторінок – 69, таблиць – 0, рисунків – 24, посилань – 26, додатків – 2.

Ключові слова: API, REST, Go, архітектура, історія, гра, сервіс, DDD.

ABSTRACT

for bachelor's qualification work of a student of 402 group

at Petro Mohyla Black Sea National University

Kravets Oleksandr Valerievich

Topic: «Computer game in the genre of Story games with software interface support»

The object of research is the process of creating a computer game in the genre of Story games.

The subject of research is tools for creating backend services and chat bots, architecture of modern applications.

The goal is to increase the convenience of the process of reading stories by giving the user the opportunity to influence the plot of stories by creating a game in the genre of Story games with support for the software interface.

The research work consists of a professional section and a special section on labor protection. The explanatory note consists of an introduction, three chapters, conclusions and appendices.

In the first section, we will analyze and select the technology stack to create an application. Consider the pros and cons of selected technologies.

In the second section we will create an architecture for the back-end service and Telegram bot, consider the feasibility of using hexagonal architecture and Domain Driven Design in the system, optimize the system by using the Redis database.

In the third section we will consider the user interface of the Telegram bot and the software interface of the back-end service.

In the last section the normative base of labor protection at use of computers was considered.

Pages – 69, tables – 0, figures – 24, links – 26, appendices – 2.

Keywords: API, REST, Go, architecture, story, game, service, DDD.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

СУБД – Система Управління Базами Даних

API – Application Programming Interface
DDD – Domain Driven Design
HTTP – Hypertext Transfer Protocol
REST – Representational State Transfer
UML – Unified Modeling Language
VO – Value Object

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи

на тему:

«КОМП'ЮТЕРНА ГРА У ЖАНРІ STORY GAMES ІЗ ВИКОРИСТАННЯМ ПРОГРАМНОГО ІНТЕРФЕЙСУ»

Спеціальність 122 «Комп'ютерні науки»

122 – БКР – 402.1810216

Виконав студент 4–го курсу, групи 402
_____ *О. В. Кравець*
(підпис, ініціали та прізвище)
«___» _____ 2022 р.

Керівник: *канд. техн. наук, доцент (б.в.з)*
(наук. ступінь, вчене звання)
_____*М. Л. Дворецький*
(підпис, ініціали та прізвище)
«___» _____ 2022 р.

Миколаїв – 2022

ВСТУП

За період своєї історії книги пройшли довгий шлях розвитку: від глиняних табличок Вавилона і Шумеру, та єгипетських папірусів, через пергаментні книги середніх віків і такі звичні нам книги з паперу до сучасних електронних книг. Читання книг має велику кількість переваг.

За даними наукових досліджень, читання справді захищає від захворювань мозку. Коли людина читає, активність мозку збільшується і мозок постійно перебуває в тонусі, що покращує його стан і допомагає уповільнити або запобігти хворобі Альцгеймера і деменції.

За рахунок читання підвищується не тільки грамотність, а й мовні навички – здатність чітко, ясно та красиво формулювати свої думки.

У світі позбавлення стресу – основна турбота багатьох людей. Багатство та ритміка книжкового тексту має властивість заспокоювати психіку та звільняти організм від стресу. Але книжки мають один великий недолік – неможливість вибору альтернативного сценарію, читач змушений слідувати, закладеному автором, шляху.

Комп'ютерні ігри являються альтернативою ігор. Часто вони допомагають вирішити проблему із чітко закладеним сценарієм.

Комп’ютерні ігри сьогодні, стали значною частиною нашого життя. За різними оцінками, у світі налічується понад 2,3 млрд. геймерів.

Деякі типи відеоігор можуть зняти стрес після важного робочого дня або покращити роботу мозку під час виконання вузького набору завдань. Ряд досліджень також показав їхню ефективність при освоєнні другої мови, вивченні математики та природничих наук.

Але ігри мають деякі недоліки, такі як викликання залежності. У 2019 році ВООЗ ухвалила вважати залежність від ігор хворобою. Також, не всі люди можуть дозволити собі грати в ігри через те, що для них потрібен потужний комп'ютер.

«Storiks» вирішує цю проблему тим, що поєднує світ книг зі світом ігор, надаючи читачеві можливість вибирати альтернативні варіанти розвитку історій, що робить читання більш цікавим.

Сторігейми – це короткі інтерактивні історії – оповідання з вибором варіантів дій, які можна читати та створювати. Захоплюючі пригоди, в яких Вам належить визначати і хід сюжету і те, яким буде закінчення історії. Читач може зануритися в захоплюючу пригоду, в якій він має визначати і хід сюжету і те, яким буде закінчення історії. Історія оживає новими фарбами залежно від прийнятих рішень. Читач більше не скований ланцюгами лінійного сюжету.

Об’єкт дослідження: процес створення комп’ютерної гри у жанрі Story games.

Предмет дослідження: засоби та інструменти для створення back-end сервісів та чат ботів, архітектура сучасних додатків.

Мета роботи: підвищення зручності процесу читання історій за рахунок надання користувачу можливості впливати на сюжет історій шляхом створення гри у жанрі Story games із підтримкою програмного інтерфейсу.

Відповідно до мети виділені наступні **завдання дипломної роботи**:

1. Аналіз та вибір стеку технологій.
2. Проектування моделювання та оптимізація системи.
3. Огляд користувацького інтерфейсу.

1 АНАЛІЗ ВИМОГ ТА ТА ВИБІР СТЕКУ ТЕХНОЛОГІЙ

1.1 Мова програмування Go. Переваги та недоліки мови Go в сучасній розробці

Go – це багатопотокова мова програмування, розроблена компанією Google. Широкій публіці було надано доступ у 2009 році.

На даний момент Go є однією з тих мов, які стрімко набирають популярність. Ця мова відносно молода, але, незважаючи на це, вона дуже хороша. Вона чудово поєднує в собі лаконічність та хорошу продуктивність, що дозволяє створювати високонавантажені програми у менші терміни.

Переваги мови Go:

- Вихідний код компілюється в бінарний файл, що значно підвищує продуктивність вашої програми. Також це дуже зручно при деплої програми – немає потреби попередньо встановлювати необхідні залежності, досить просто перенести виконуваний файл і запустити його.
- Go досить простий для вивчення – він має зрозумілий синтаксис і хорошу документацію, яка читається без особливих зусиль.
- Особливо хочеться виділити багатопоточність у Go. Горутини (goroutines) і канали (channels), інструменти Go до роботи з потоками несуть у собі одночасно легкість і практичність.
- Спочатку мова розроблялася як заміна C++: їх продуктивність знаходиться майже на одному рівні, але Go при цьому відрізняється більш простим синтаксисом, що дозволяє вести розробку додатків швидше.
- Go це мова, що компілюється, на якій можна швидко написати потрібну програму. Типізація мови строга, як у Paskal, наприклад, але при цьому відрізняється великим прикладним характером, що ріднить мову з Python.

Притаманна Go і характерна для компільованих мов висока продуктивність та полегшена кроссплатформенність.

Недоліки:

– Синтаксис занадто спрощений. Не вистачає сучасних ООП–конструкцій, як у більш «дорослих» язиках типу Java та C#.

Для створення REST API був обраний фреймворк gin–gonic/gin.

Gin це веб–фреймворк, написаний на Go. Він має API, подібний до Martini, з набагато кращою продуктивністю, до 40 разів швидше завдяки httprouter.

Особливості фреймворка:

– Маршрутизація на основі дерева Radix, невеликий обсяг пам'яті.
– Вхідний HTTP–запит може оброблятися ланцюгом проміжних програм і кінцевою дією.
– Gin може вловити паніку, що виникла під час HTTP–запиту, і відновити його. Таким чином, сервер буде завжди доступним.
– Gin може аналізувати та перевіряти JSON запит, наприклад, перевіряти наявність необхідних значень.
– Gin надає зручний спосіб збору всіх помилок, що виникли під час HTTP–запиту. Зрештою, проміжне програмне забезпечення може записати їх у файл журналу, в базу даних і відправити їх через мережу.
– Gin надає простий у використанні API для рендерінгу JSON, XML і HTML

1.2 База даних та міграції back-end сервісу Storiks API

В якості основної бази даних використовується PostgreSQL. PostgreSQL – СУБД, яка використовує реляційну модель для баз даних і підтримує стандартну мову запитів SQL.

PostgreSQL надає безліч різних можливостей, досить надійна і має хороші характеристики щодо продуктивності. Вона працює практично на всіх UNIX–

платформах, включаючи UNIX–подібні системи, такі як FreeBSD та Linux. Її можна використовувати на Windows NT Server і Windows 2000 Server, а розробки підходять навіть такі системи Microsoft для робочих станцій, як ME. Крім того, PostgreSQL вільно поширюється та має відкритий вихідний код.

PostgreSQL вигідно відрізняється від багатьох інших СУБД. Вона має практично всі можливості, які є в інших базах даних (комерційних або Open Source), а також деякими додатковими.

Наведемо перелік функціональних можливостей PostgreSQL:

- Транзакції
- Вкладені запити
- Посилальна цілісність – зовнішні ключі
- Складні блокування
- Типи, що визначаються користувачем
- Перевірка сумісності версій

Починаючи з версії 6.5 PostgreSQL є дуже стійкою системою, кожна наступна версія проходить процедуру регресивного тестування, що забезпечує стабільність.

У ході експлуатації PostgreSQL проявила себе як заслуговуюча довіри СУБД. Кожна версія перевіряється дуже ретельно, бета–версії проходять щонайменше місячне тестування. Завдяки численній спільноті користувачів та відкритому доступу до вихідного коду помилки виправляються дуже швидко.

Продуктивність цієї СУБД також зростає від версії до версії, і останні атестації показують, що за певних умов вона поступається комерційним продуктам. Деякі системи, що мають не такий повний набір можливостей, перевершують PostgreSQL у продуктивності, але за рахунок втрати функціональності.

Однією із сильних сторін PostgreSQL є її архітектура. Як і багато комерційних СУБД, PostgreSQL може застосовуватися в середовищі клієнт–сервер, що дає масу переваг як користувачам, так і розробникам.

Основа PostgreSQL є серверним процесом бази даних. Він виконується одному сервері. У цій СУБД ще не реалізована технологія високої готовності, як у деяких інших комерційних системах рівня підприємства, які можуть розподіляти навантаження між декількома серверами, домагаючись таким чином додаткової масштабованості та стійкості до зовнішніх впливів.

Доступ із додатків до даних бази здійснюється за допомогою процесу бази даних. Клієнтські програми не можуть отримати доступ до даних самостійно, навіть якщо вони працюють на тому комп'ютері, на якому виконується серверний процес.

PostgreSQL орієнтований на TCP/IP – це може бути локальна мережа або Інтернет.

Завдяки тому, що маніпулювання даними зосереджено на сервері, СУБД не доводиться контролювати численних клієнтів, які отримують доступ до спільно використовуваного каталогу сервера, і PostgreSQL може підтримувати цілісність даних навіть при одночасному доступі великої кількості користувачів.

Open Source–ліцензія дає право на використання, модифікацію та розповсюдження програмного забезпечення без ліцензійних виплат.

Під час роботи з базою даних міграція схеми є одним із важливих завдань, які нам часто доводиться виконувати протягом усього життя програми, щоб адаптуватися до нових вимог бізнесу.

Для роботи з міграціями було обрано бібліотеку golang–migrate/migrate. Ця бібліотека є однією з найпопулярніших бібліотек для роботи з міграціями в Go.

Go–Migrate зчитує міграції з джерел та застосовує їх до бази даних у правильному порядку.

1.3 Використання бази даних Redis для кешування

Для кешування в Storiks API був обран зовнішній кеш у вигляді бази даних Redis. Це один з найпопулярніших варіантів для кешування. Зовнішній кеш дозволить в майбутньому дуже добре масштабувати систему.

Redis (REmote DIctionary Server) – це нереляційна високопродуктивна СУБД. Redis зберігає всі дані у пам'яті, доступ до даних здійснюється за ключем. Опційно копія даних може зберігатись на диску. Цей підхід забезпечує продуктивність, що в десятки разів перевершує продуктивність реляційних СУБД, а також спрощує секціонування (шардинг) даних.

Насамперед, Redis вміє зберігати дані на диск. Можна налаштувати Redis так, щоб дані взагалі не зберігалися, зберігалися періодично за принципом copy–on–write, або зберігалися періодично і писалися в журнал (binlog). Таким чином, завжди можна досягти необхідного балансу між продуктивністю та надійністю.

Redis дозволяє зберігати не тільки рядки, але й масиви (які можуть використовуватися як черги або стеки), словники, множини без повторів, великі масиви біт (bitmaps), а також множини, відсортовані за якоюсь величиною. Зрозуміло, можна працювати з окремими елементами списків, словників та множин. Redis дозволяє вказати час життя даних (двома способами – «видалити» тоді» і «видалити через …»). За замовчуванням усі дані зберігаються вічно.

Цікава особливість Redis полягає в тому, що це однопоточний сервер. Таке рішення спрощує підтримку коду, забезпечує атомарність операцій і дозволяє запустити по одному процесу Redis на кожне ядро процесора. Зрозуміло, кожен процес прослуховуватиме свій порт. Рішення нетипове, але цілком виправдане, тому що на виконання однієї операції Redis витрачає дуже невелику кількість часу (порядку однієї стотисячної секунди).

У Redis є реплікація. Реплікація з кількома основними серверами не підтримується. Кожен підлеглий сервер може у ролі головного іншим. Реплікація в Redis не призводить до блокування ні головному сервері, ні підлеглих. На репліках дозволено операцію запису. Коли головний та підлеглий сервер відновлюють з'єднання після розриву, відбувається повна синхронізація (resync).

Також Redis підтримує транзакції (будуть послідовно виконані або всі операції, або жодної) і пакетну обробку команд (виконуємо пачку команд, потім отримуємо пачку результатів). До того ж ніщо не заважає використовувати їх разом.

Ще одна особливість Redis – підтримка механізму publish/subscribe. З його допомогою програми можуть створювати канали, підписуватись на них та поміщати у канали повідомлення, які будуть отримані всіма передплатниками.

Також хотілося б зазначити таке:

- Redis дуже простий і чудово документований;
- На даний момент довжина ключа в Redis може становити до 231 байт, довжина рядка – до 512 Мб, списки та множини можуть містити до 232 елементів, один екземпляр Redis може зберігати до 232 ключів;
- На одному сервері можна тримати кілька пронумерованих баз даних, за замовчуванням їх число дорівнює 16–и.
- Програми, що використовують Redis, зручно профілювати (команда slowlog) та налагоджувати (команда monitor);
- Redis написаний таким чином, що резервну копію бази даних можна зробити простим копіюванням файлу дампа, навіть під час роботи сервера;
- Доступ до сервера можна захистити паролем;

Найсерйозніше обмеження Redis у тому, що обсяг даних, який може зберігатися одному фізичному сервері, обмежений обсягом оперативної пам'яті цьому сервері. Була спроба обійти це обмеження за рахунок використання

віртуальної пам'яті, але ця ідея була визнана невдалою. Таким чином, зберігати в Redis багато даних варто недешево.

Варіанти використання Redis:

- Сховище сесій та профілів користувачів;
- Сервер черг, плюс тримаємо в умі механізм|publish/subscribe;
- Повноцінна заміна Memcached, при цьому у випадку з Redis ми отримаємо реплікацію, довші ключі та значення, можливість відновлення кешу з диска тощо;
- Місце для зберігання кількості користувачів онлайн, кодів капч, різних прапорів, саджестів пошукових запитів;
- СУБД для невеликих додатків – скорочувань посилань, іміджбордів, можливо навіть блогів;
- Роль «словника» у шардингу, тобто сервер, який знає, які шарди на яких серверах шукати;
- Сховище проміжних результатів обчислень під час обробки великих обсягів даних;

1.4 Використання MongoDB в якості бази даних для Telegram бота

Як ми знаємо, будь–яка реляційна БД має стандартну схему, яка показує кількість таблиць та зв'язку між ними. У MongoDB такої схеми зв'язку між таблицями немає.

MongoDB – документоорієнтована система управління базами даних з відкритим вихідним кодом. Для зберігання даних використовується JSON–подібний формат. Ця СУБД відрізняється високою доступністю, масштабованістю та безпекою.

Головні особливості MongoDB:

- Це кроссплатформенна документоорієнтована база даних NoSQL з відкритим вихідним кодом.

– Вона не вимагає опису схеми таблиць, як у реляційних БД. Дані зберігаються у вигляді колекцій та документів.
– Між колекціями немає складних з'єднань типу JOIN, як між таблицями реляційних БД. Зазвичай з'єднання здійснюється за збереження даних шляхом об'єднання документів.
– Дані зберігаються у форматі BSON (бінарні JSON–подібні документи).
– У колекцій не обов'язково має бути подібна структура. Один документ може мати один набір полів, тоді як інший документ – зовсім інший (як тип, і кількість полів).
– В одному документі можуть бути поля різних типів даних, дані не потрібно наводити до одного типу. Основна перевага MongoDB полягає в тому, що вона може зберігати будь–які дані, але ці дані мають бути у форматі JSON.

Нижче наведено кілька причин, з яких варто використовувати MongoDB:

– Документоорієнтована база – збереження даних у форматі документів замість формату реляційного типу, це робить MongoDB дуже гнучкою та адаптованою до бізнес–вимог. Можливість зберігання різних типів даних особливо важлива під час роботи з великими даними, які збираються з різних джерел і не лягають в одну структуру.
– Спеціальні запити – MongoDB підтримує пошук за полями, діапазонні запити та пошук за регулярними виразами. Можуть бути зроблені запити повернення певних полів у документах.
– Індексація – Ви можете створити індекси для покращення продуктивності пошуку в MongoDB. Будь–яке поле в документі може бути проіндексовано. Це забезпечує високу швидкість роботи СУБД.
– Реплікація – ця СУБД може забезпечити високу доступність за допомогою наборів реплік. Набір реплік складається з двох або більше

примірників MongoDB. Кожна репліка набору може бути ролі первинної чи вторинної. Первинна репліка – головний сервер, який взаємодіє з клієнтом та виконує всі операції читання/запису. Вторинні репліки зберігають копію даних первинної репліки за допомогою вбудованої реплікації. Якщо з первинною реплікою щось трапилося, відбувається автоматичне перемикання на вторинну репліку, потім стає основним сервером.

– Балансування навантаження – MongoDB використовує концепцію шардингу для горизонтального масштабування за допомогою поділу даних між кількома екземплярами БД. Вона може працювати на декількох серверах, балансуючи навантаження та/або дублюючи дані, щоб підтримувати працездатність системи у разі апаратного збою.
– Можливість розгорнути у хмарі – ви отримуєте готову до роботи, оптимально налаштовану, масштабовану та керовану базу даних на запит за дві хвилини.
– Доступність – MongoDB підтримує всі популярні мови програмування, її можна використовувати безкоштовно як Open Source рішення.

Виходячи с цих переваг можна сказати, що MongoDB є досить хорошим рішенням для нашого телеграм бота.

1.6 Особливості та переваги месенджеру Telegram

Раніше в глобальній мережі користувачі, в основному, спілкувалися через соціальні мережі. Тепер з'явилися і інші можливості. Досить мати під рукою ноутбук, телефон або планшет. З цих пристроїв можна відправляти документи, файли і повідомлення без використання платних СМС, а також здійснювати дзвінки. Це стало можливим разом з появою месенджерів.

Месенджер – це спеціальний додаток або програма, яку завантажують і встановлюють на смартфон або комп'ютер. Його основна мета – це миттєвий

обмін текстовими повідомленнями, фото, картинками, відео, документами з друзями, родичами, знайомими, колегами по роботі або по навчанню. Також можна здійснювати дзвінки за допомогою аудіо або відеозв'язку.

WhatsApp, Viber, Telegram, WeChat, Line, Facebook Messenger та інші. Ці месенджери зручні і стали звичним засобом для комунікацій. Щодня вони дедалі більше заполонюють наше повсякденне життя. Ми спілкуємося з друзями та родичами, колегами у чатах. І це стало звичною справою.

Telegram – найбільш технологічно просунутий та безпечний месенджер.

На даний момент, нова соціальна мережа телеграм користується великим попитом, особливо з боку людей, які вважають, що їхнє листування повинно залишатися лише між ними. Як заявляють багато сучасних видань, Telegram здатний нав'язати конкуренцію будь–кому, у тому числі ВК і facebook.

До основних переваг відносяться:

- Телеграм піклується про безпеку даних. Захист від несанкціонованого читання здійснюється за допомогою протоколу зв'язку MTProto.
- Наявність секретних чатів. Саме вони гарантують повну конфіденційність листування, оскільки інформація ніде не зберігається.
- Велика кількість способів зробити листування максимально непомітним. Наприклад, певний код або спеціальний таймер, за допомогою якого діалоги автоматично видаляються.
- Є як мобільні, і комп'ютерні версії.
- Є і супер чати, які здатні вмістити до 5000 осіб.
- Можливість передачі файлів великого розміру (до 1 Гб).
- Висока швидкість роботи, зокрема. доставки файлів адресату.
- Синхронізація між користувачами.

Для розробки робота був обраний месенджер телеграм. Оскільки телеграм найпоширеніший месенджер це дозволить максимально великій

кількості людей отримати доступ до нашої програми. Також користувачам не потрібно буде нічого завантажувати додатково, достатньо мати встановлений месенджер. За рахунок зручного, звичного та інтуїтивно зрозумілого інтерфейсу користувач не буде відчувати дискомфорту при знайомстві з додатком та проходженням історій.

Чат–боти – це спеціальні акаунти, які не закріплені за людьми, а повідомлення, що відправляються від них або їм, обробляються зовнішньою системою. При цьому для користувача спілкування з ботом виглядає як звичайне листування з реальною людиною.

Чат–бот це ще й альтернатива мобільному додатку. Так як створити чат–бот месенджера простіше та дешевше, ніж мобільний додаток. Адже месенджер–база для нього вже є, тому не потрібно буде підлаштовуватись під різні операційні системи смартфонів, наймати спеціалістів, продумувати дизайн, а ще проходити довгу та дорогу модерацію, щоб розмістити додаток у AppStore, GooglePlay та інших офіційних магазинах. До того ж, програми займають пам'ять, не вчасно оновлюються та не вантажаться з повільним інтернетом.

Месенджери та їх чат–боти дуже зручні, тому що переносять частину рутинних операцій у звичне для користувача середовище – месенджери, де ми проводимо великий період часу, починаючи з керівництва до рядових співробітників; функціонують за принципом «не ускладнювати» або «кращий інтерфейс – це відсутність інтерфейсу», тому позбавляють нас зайвого перемикання на інші додатки, яких у нашому смартфоні і так достатньо; мінімізують вплив «людського фактора» при передачі інформації та відповідно:

- знижують ймовірність помилок;
- додають часу для виконання більш значущих завдань, позбавляючи рутинних операцій;
- підвищують рівень керованості процесами завдяки функції «справедливого» контролера;

Типи чат–ботів

Залежно від того, як були запрограмовані конкретні чат–боти, ми можемо розділити їх на дві великі групи: ті, які працюють відповідно до заздалегідь створених команд (простий чат–бот) та навчені (інтелектуальний або розширений чат–бот).

Прості чат–боти працюють за раніше написаними ключовими словами, які вони розуміють. Кожна з цих команд має бути написана розробником окремо з використанням регулярних виразів чи інших форм аналізу рядків.

Якщо користувач поставив запитання без одного ключового слова, робот не зможе його зрозуміти і, як правило, відповідає такими повідомленнями, як «вибачте, я не зрозумів».

Розумні чат–боти покладаються на штучний інтелект під час спілкування з користувачами.

Замість підготовлених відповідей робот відповідає відповідними пропозиціями з цього питання. Крім того, всі слова, сказані клієнтами, записуються для подальшої обробки.

Тим не менш, штучний інтелект не диво і ще не готовий виробляти ідеальний досвід для користувачів. Навпаки, це потребує багато роботи.

Основними факторами, що спонукають людей використовувати чат–ботів, є:

Продуктивність. Чат-боти надають допомогу або доступ до інформації швидко та ефективно.

Розваги. Чат–боти розважають людей, даючи їм забавні поради, вони також допомагають вбивати час, коли користувачам нічого робити.

Соціальні та реляційні фактори . Чат–боти забезпечують конверсію та покращують соціальний досвід. Спілкування з ботами також допомагає

уникнути самотності, дає можливість говорити без суджень і покращує навички розмови.

Telegram має найголовніший бот для створення – @BotFather, який видає інструкцію для створення свого бота. Створивши нового робота і отримавши ключ доступу, потрібно буде приєднати його до системи.

Висновки до розділу 1

У даному розділі було проаналізовано та обрано стек технологій для розробки додатку. В якості мови програмування було обрано мову програмування Go. В якості основної бази даних для back-end сервісу було обрано PostgreSQL, а для Telegram бота використовується база даних MongoDB. Для кешування даних було обрано базу даних Redis. Для комунікації між ботом та сервісом використовується архітектурний стиль REST API.

Також були розглянуті особливості месенджера Telegram та його переваги.

За рахунок використання сучасних технологій та практик додаток можна легко масштабувати, що дає можливість витримувати високі навантаження.

2 ПРОЕКТУВАННЯ ТА МОДЕЛЮВАННЯ СИСТЕМИ

2.1 Архітектура back-end сервісу Storiks API. Гексагональна архітектура

Для розробки системи було обрано гексагональну архітектуру.

Будучи одним із варіантів листкової архітектури, гексагональна має на увазі поділ додатка на окремі концептуальні шари, що мають різну зону відповідальності, а також регламентує те, яким чином вони пов'язані один з одним.

Гексагональна архітектура не новий підхід до розробки із застосуванням фреймворків. Навпаки, це лише узагальнення «найкращих практик» – практик нових і старих.

Цей тип архітектури дотримується класичних ідей, до якої приходять розробники при проектуванні програм: відокремлення коду програми від фреймворку. Програма формується сама по собі, а не на базі фреймворку, використовуючи останній лише як інструмент для вирішення якихось завдань нашої програми.

Незважаючи на те, що архітектура називається гексагональною, що має вказувати на фігуру з певною кількістю граней, основною думкою все ж таки є те, що граней у цієї фігури багато. Кожна грань є «портом» доступу до нашого додатку або його зв'язок із зовнішнім світом.

Порт може бути представлений як будь–який провідник вхідних запитів (або даних) до нашої програми. Наприклад, через порт HTTP (запити браузера, API) надходять запити HTTP, які конвертуються в команди для програми. Подібним чином можуть діяти різні менеджери черг або будь–що, що взаємодіє з додатком за протоколом пересилання повідомлень (наприклад AMQP). Все це є лише портами, через які ми можемо попросити програму зробити якісь дії. Ці грані становлять безліч портів, що «входять». Інші порти можуть бути використані для доступу до даних із боку програми, наприклад порт бази даних.

Архітектура дозволяє взаємодіяти з додатком як користувачеві, так і програмам, автоматичних тестів, скриптів пакетної обробки. Також дозволяє розробляти та тестувати програму без будь–яких додаткових пристроїв або баз даних.

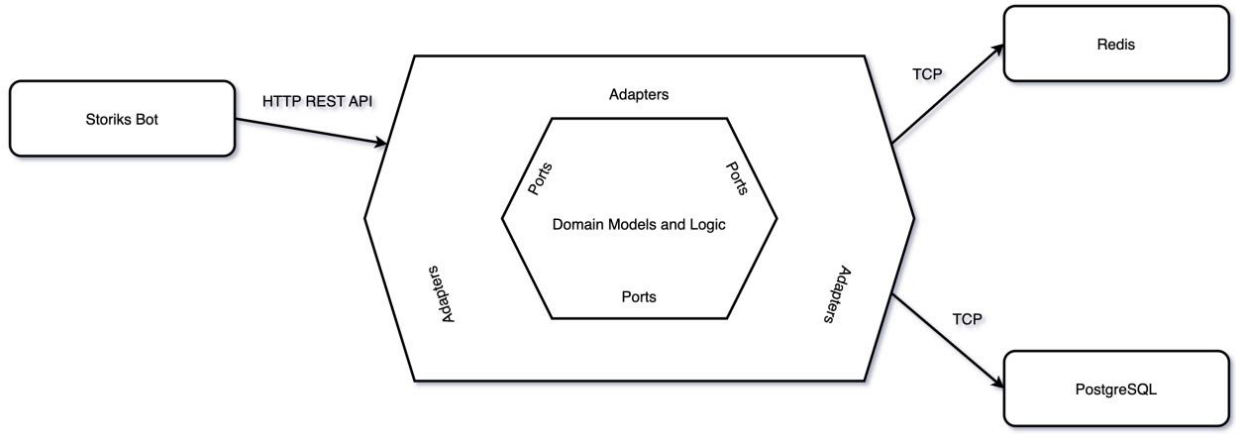

Рисунок 2.1 – Архітектура сервісу Storiks API

Пакет storiks/internal/domain створений з використанням DDD підходу і містить у собі основні структури додатку та бізнес-логіку. Також пакет storiks/internal/domain містить в собі абстракції для роботи зі сховищами даних. Такий підхід дозволяє зібрати в одному місті всю бізнес логіку додатку і в подальшому легко її розширювати.

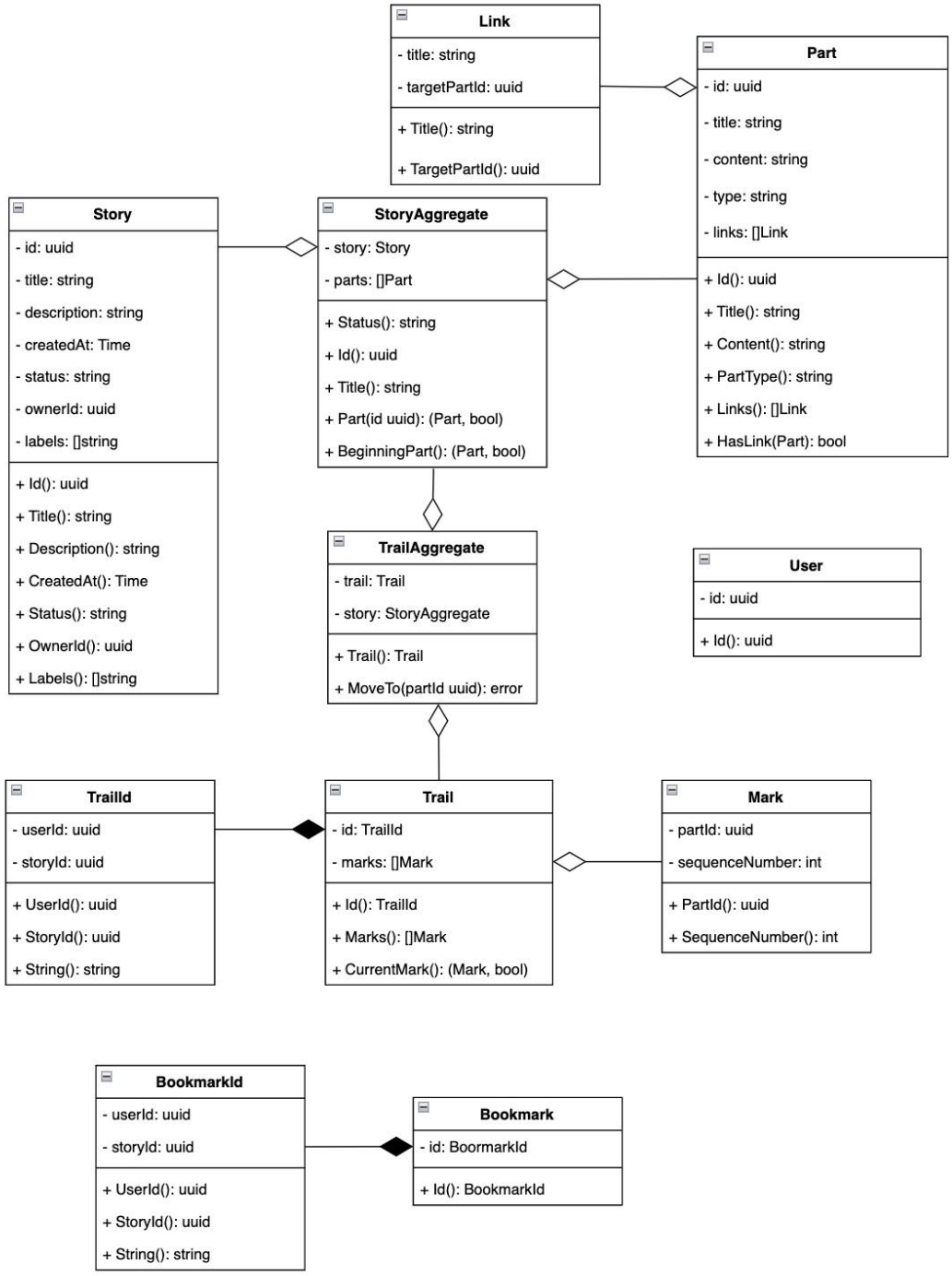

Рисунок 2.2 – UML діаграма пакету storiks/internal/domain

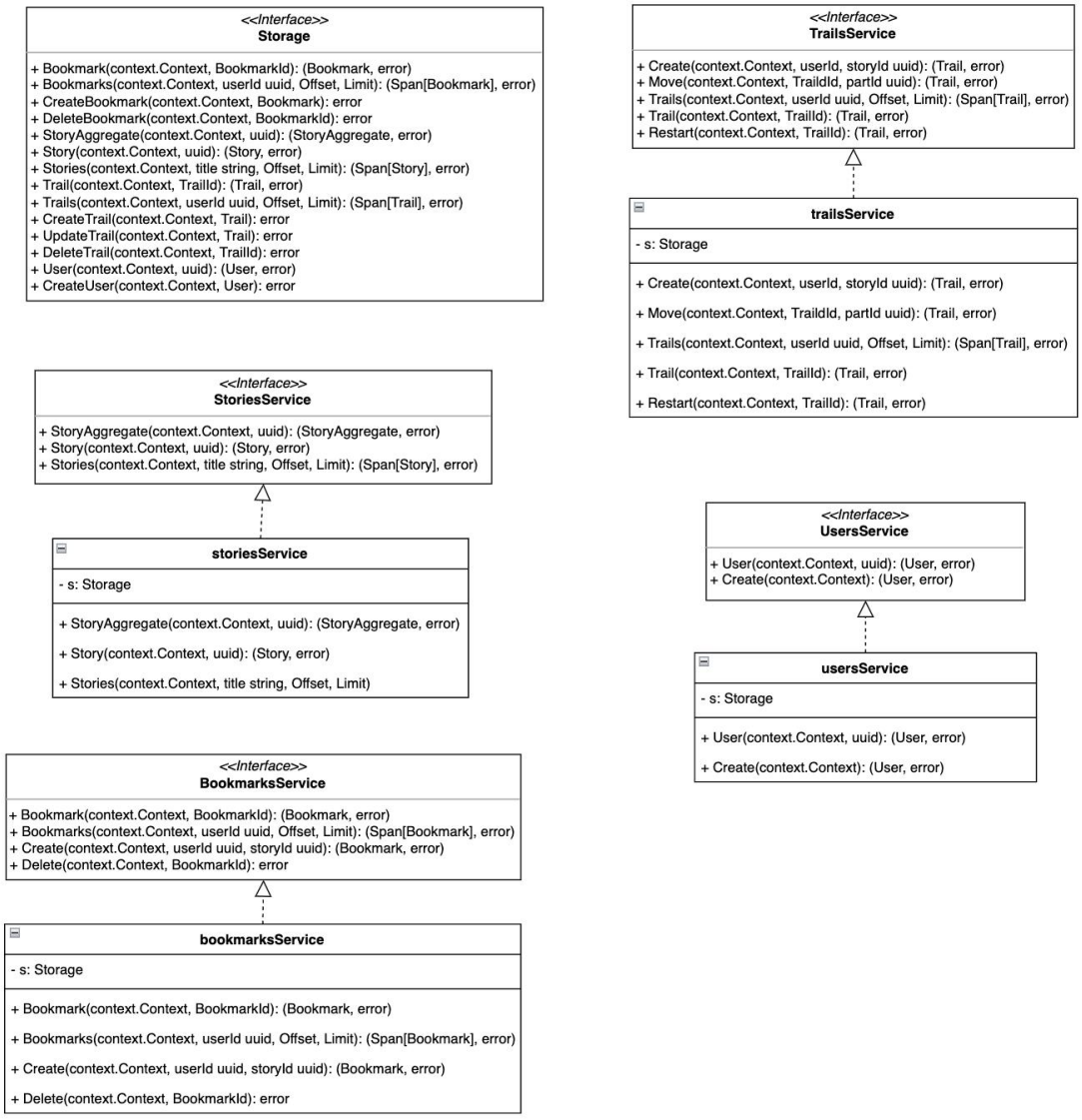

Рисунок 2.3 – UML діаграма пакету storiks/internal/domain

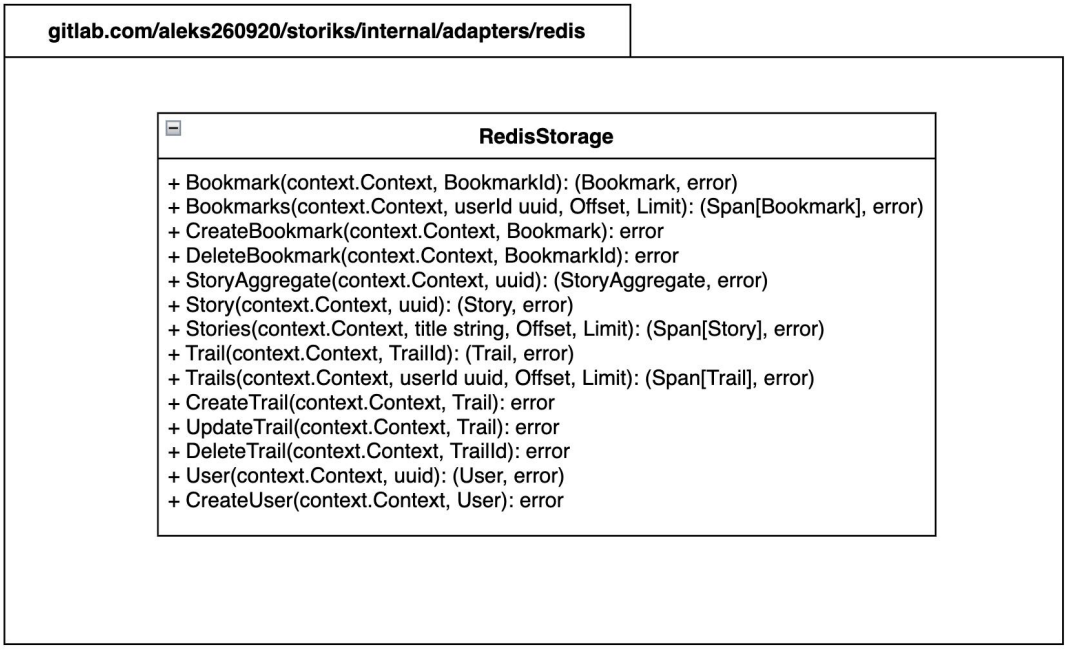

Рисунок 2.4 – UML діаграма пакету storiks/internal/adapters/redis

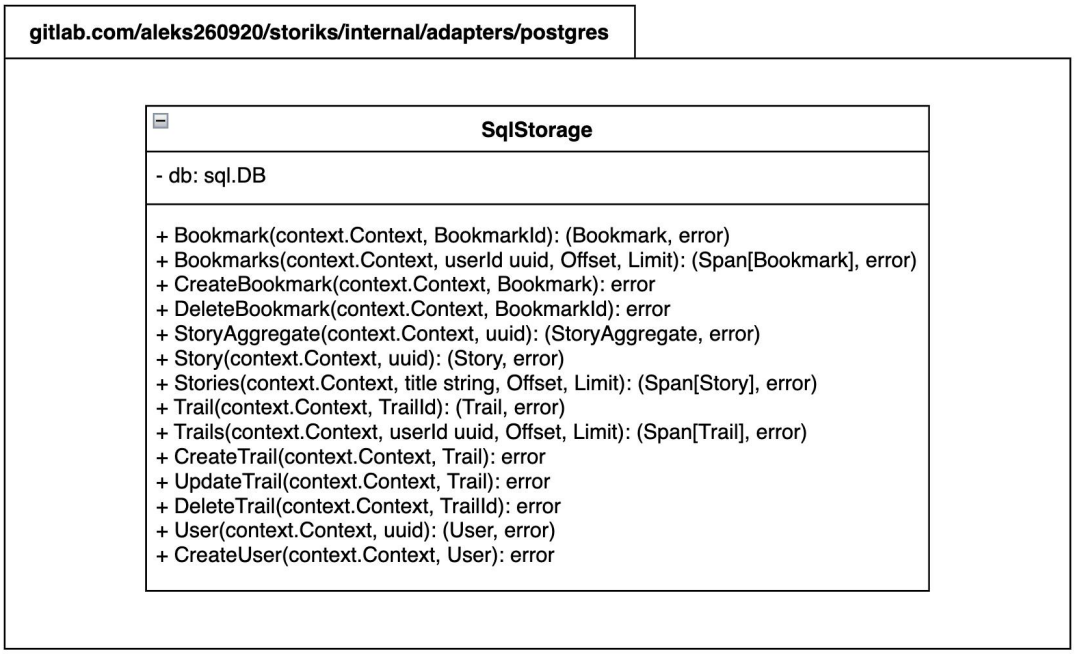

Рисунок 2.5 – UML діаграма пакету storiks/internal/adapters/postgres

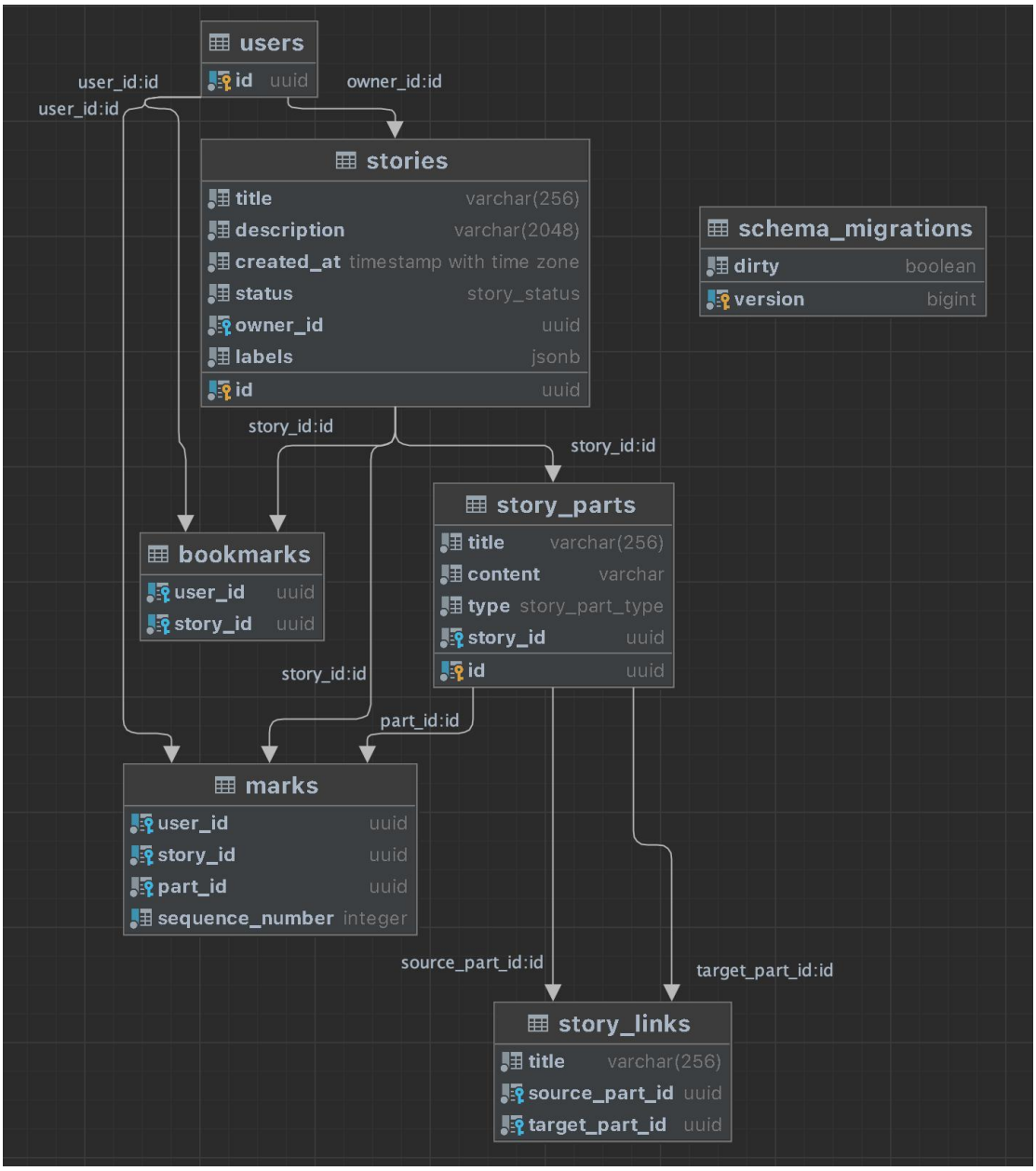

Рисунок 2.6 – Схема бази даних

Таблиця stories містить в собі історії, які доступні для читання користувачам.

Таблиця parts містить в собі частини історій.

Таблиця story_links містить в собі посилання між частинами історій.

Таблиця bookmarks містить в собі закладки користувачів.

Таблиця marks містить в собі відмітки, які залишає користувач під час проходження історій.

Таблиця users містить в собі користувачів системи.

Таблиця schema_migrations містить в собі список міграцій бази даних.

2.2 Архітектура бота

Архітектура бота дуже схожа на архітектуру Storiks API за винятком того, що в якості бази даних використовується MongoDB.

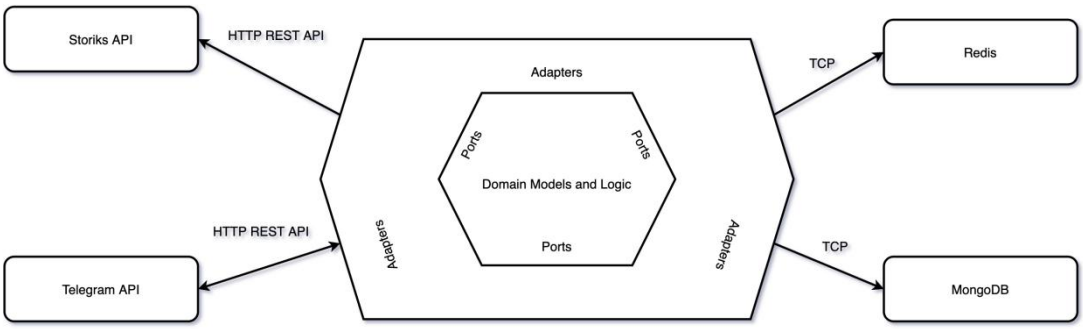

Рисунок 2.7 – Архітектура бота

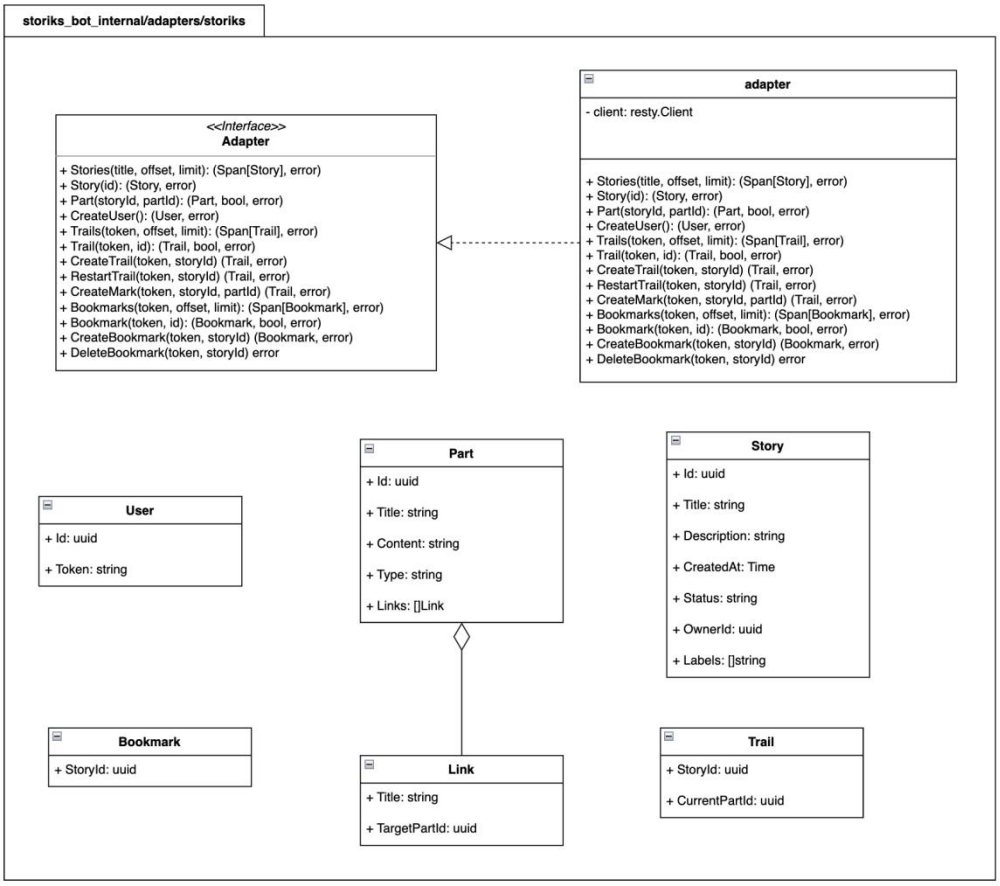

Рисунок 2.8 – UML діаграма пакету storiks_bot_internal/adapters/storiks

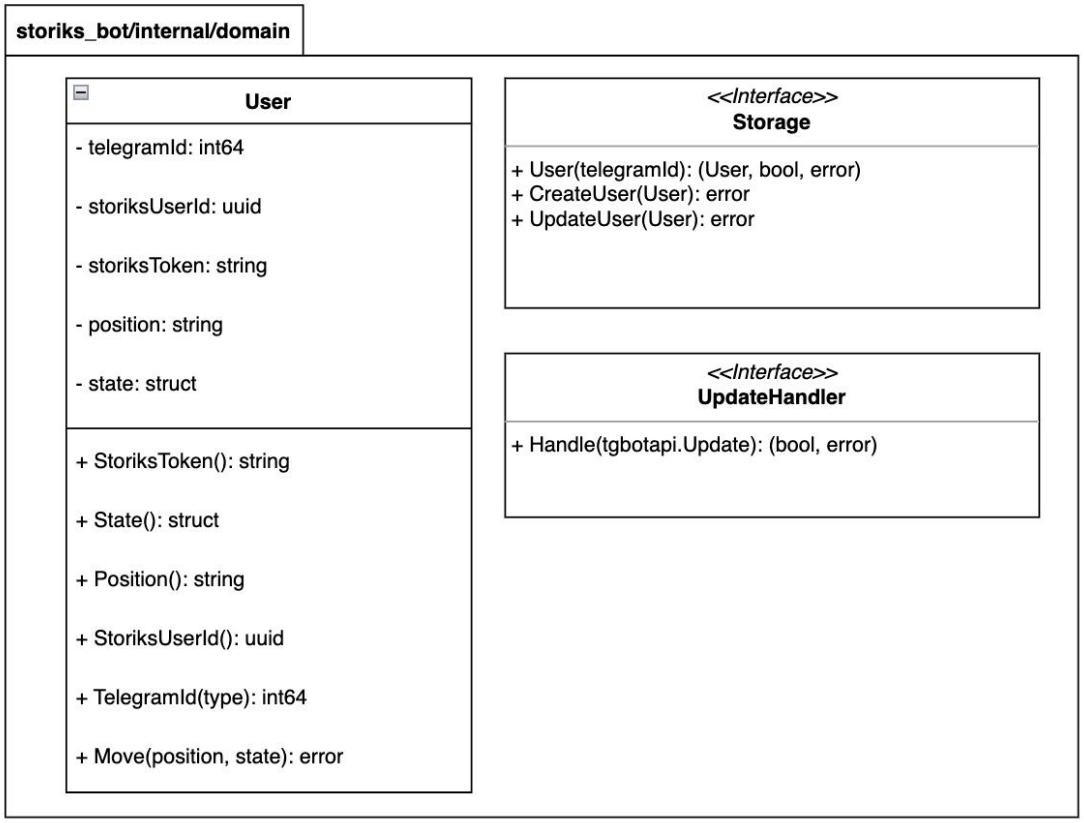

Рисунок 2.9 – UML діаграма пакету storiks_bot/internal/domain

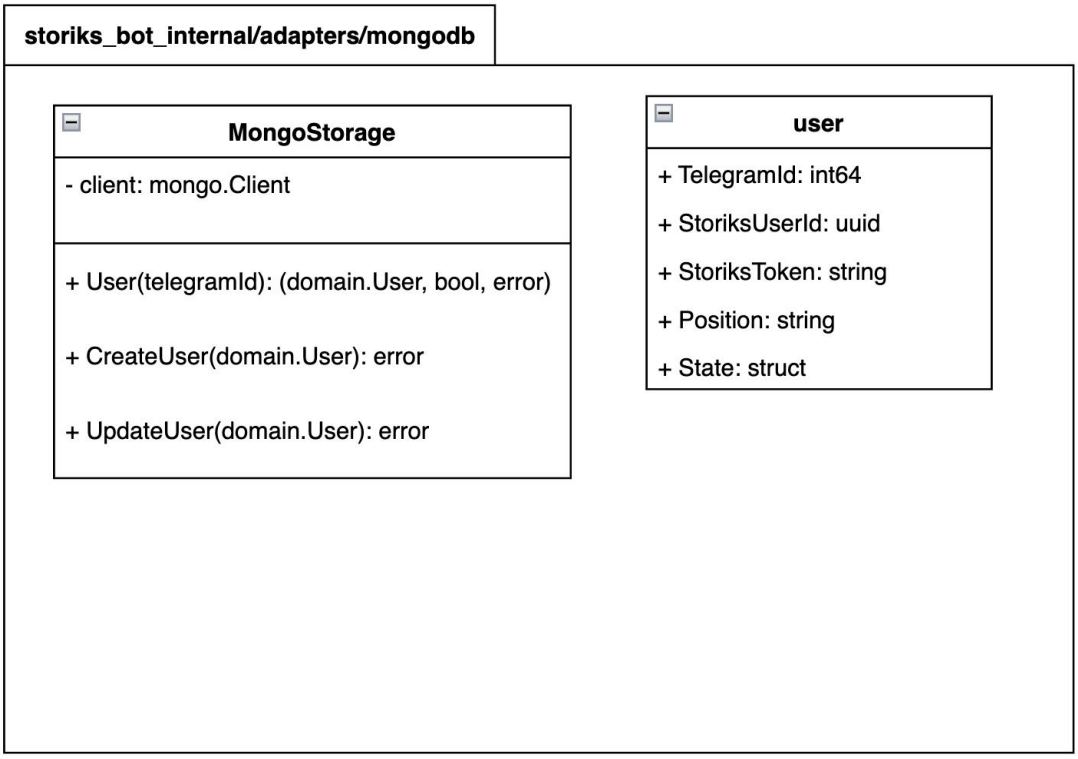

Рисунок 2.10 – UML діаграма пакету storiks_bot_internal/adapters/mongodb

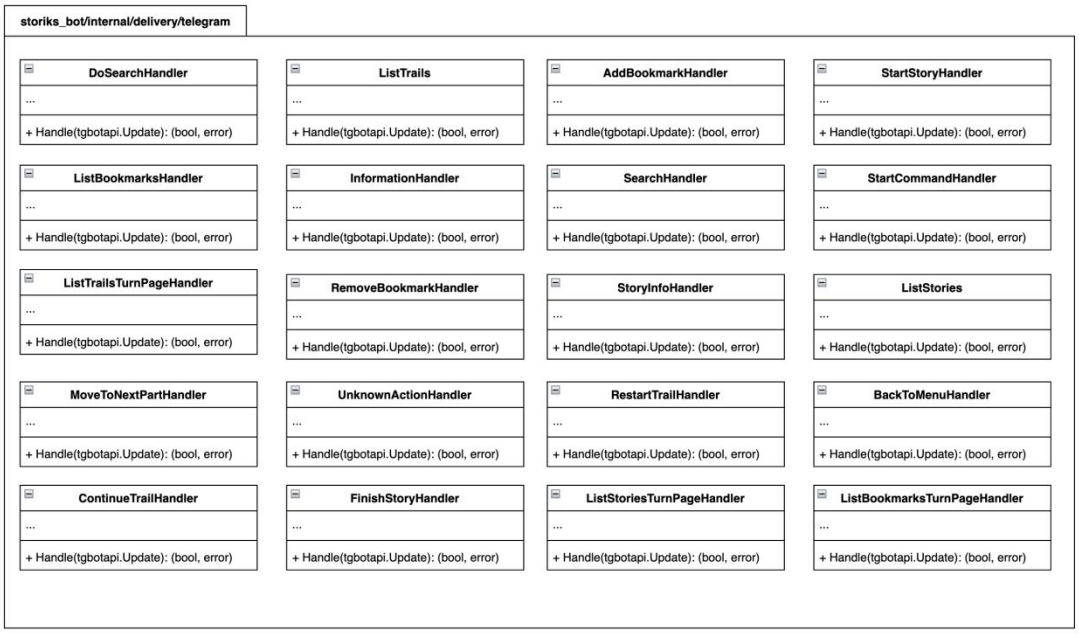

Рисунок 2.11 – UML діаграма пакету storiks_bot/internal/delivery/telegram

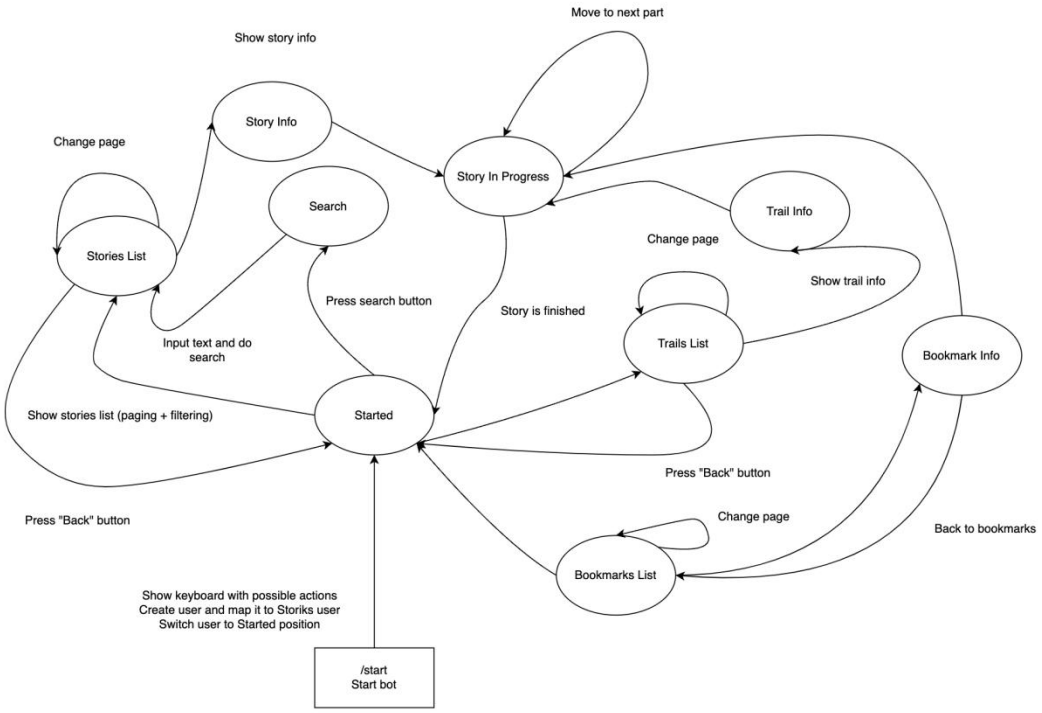

Рисунок 2.12 – Діаграма переходу станів бота

База даних бота складається з однієї колекції users. Колекція users містить дані про користувачів. Кожен документ складається з _id, position, state, storiksToken, storiksUserId, telegramId.

_id	– унікальний ідентифікатор користувача у базі даних
telegramId	– унікальний ідентифікатор користувача у Telegram
position	– поточна позиція користувача
state	– стан користувача
storiksToken	– токен, для доступу до Storiks API
storiksUserId	– унікальний айді користувача в Storiks API

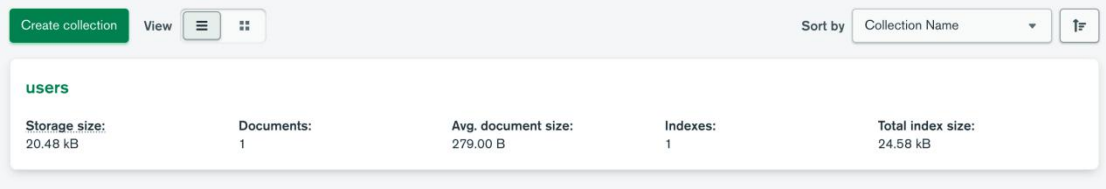

Рисунок 2.13 – Колекція users у базі даних

2.3 Особливості DDD та його використання в системі

Для опису логіки сервісу використовується Domain Driven Design.

Domain–driven design (Предметно–орієнтоване проектування, рідше проблемно–орієнтоване) – це набір принципів та схем, спрямованих на створення оптимальних систем об'єктів. Зводиться створення програмних абстракцій, які називаються моделями предметних областей. У ці моделі входить бізнес–логіка, що встановлює зв'язок між реальними умовами галузі застосування продукту та кодом.

Предметно–орієнтоване проектування не є якоюсь конкретною технологією або методологією. DDD – це набір правил, що дозволяють приймати правильні проектні рішення. Цей підхід дозволяє значно прискорити процес проектування програмного забезпечення у незнайомій предметній галузі.

Стратегічне проектування

Проектування на високому рівні абстракції, без технічних нюансів, що здійснює вся команда – як менеджерами/замовниками, так і технічними спеціалістами.

Основною метою застосування DDD є отримання високоякісної моделі програмного забезпечення, яка максимально точно відображатиме поставлені бізнес–мети. Для цього потрібно об'єднання зусиль розробників і експертів у предметної області. Створення дружної та згуртованої команди дозволяє отримати велику кількість переваг для бізнесу. Обмін знаннями між членами команди знижує шанси появи «таємного знання» про модель, досягається консенсус між експертами предметної галузі щодо різних понять та термінології, розробляється більш точне визначення та опис самого бізнесу.

Ubiquitous Language

Для того, щоб зрівняти розробників та експертів предметної області, щоб було набагато простіше обмінюватися корисними знаннями про предметну область, підхід DDD пропонує застосовувати загальний набір термінів, понять та фраз, який використовуватиметься у спілкуванні між членами команди, і який пізніше позначиться у вихідному коді результуючої програми.

Ця колективна мова термінів називається – єдина мова. (Ubiquitous Language). Це один з основних та найважливіших шаблонів предметно–орієнтованого проектування. Це не бізнес жаргон, нав'язаний розробникам, а справжня мова, створена цілісною командою – експертами у предметній галузі, розробниками, бізнес–аналітиками та всіма, хто залучений до створення системи. Роль у команді не така істотна, оскільки кожен член команди використовує для опису проекту єдину мову.

Bounded Context

Дуже важливо розуміти, що в рамках предметної області сенс певного терміна чи фрази може відрізнятися. Існує певна межа, у межах якої поняття єдиної мови мають цілком конкретне контекстне значення – обмежений

контекст (Bounded context). Це другий за значимістю термін DDD після єдиної мови. Обидва ці поняття взаємопов'язані і неспроможні існувати один без одного.

Отже, обмежений контекст – це явна межа, всередині якої існує модель предметної області, що відображає єдину мову моделі програмного забезпечення.

У кожному обмеженому контексті існує лише одна єдина мова. Обмежені контексти є відносно невеликими, менше, ніж може здатися на перший погляд. Єдиний означає «всюдисущий» або «повсюдний», тобто мова, якою розмовляють члени команди і якою виражається окрема модель предметної області, яку розробляє команда. Мова є єдиною лише в рамках команди, що працює над проектом у єдиному обмеженому контексті. Спроба застосувати єдину мову в рамках всього підприємства або що гірше серед кількох підприємств закінчиться провалом.

Domain

Це те, що робить організація, і середовище, в якому вона це робить. Розробник програмного забезпечення для організації обов'язково працює у її предметній галузі. Слід розуміти, що з розробці моделі предметної області необхідно зосередитися у певній підобласті, оскільки неможливо створити єдину, всеосяжну модель бізнесу навіть помірковано складної організації. Дуже важливо розділяти моделі на логічні розділені предметні підобласті (Subdomain) всієї предметної області відповідно до їх фактичної функціональності. Підобласті дозволяють швидше визначити різні частини предметної області, необхідні для вирішення конкретного завдання

Core domain

Дуже важливим є аспект підходу DDD. Змістове ядро – це підобласть, має першорядне значення організації. Зі стратегічної точки зору бізнес повинен виділятися своїм змістовим ядром. Більшість DDD проектів зосереджено саме

на змістовому ядрі. Найкращі розробники та експерти мають бути задіяні саме у цій підобласті. Більшість інвестицій має бути спрямована саме сюди для досягнення переваги для бізнесу та отримання найбільшого прибутку.

Простір завдань та простір рішень

Предметні області з простору завдань та простору розв'язків. Простір завдань дозволяє думати про стратегічну бізнес проблему, яка повинна бути вирішена, а простір рішень, зосередиться на тому, як реалізується програмне забезпечення, щоб вирішити бізнес проблему.

Простір завдань – частини предметної області, які необхідні, щоб створити смислове ядро. Це комбінація смислового ядра та підобластей, яке це ядро має використовувати.

Простір рішень – один чи кілька обмежених контекстів, набір конкретних моделей програмного забезпечення. Розроблений обмежений контекст – це конкретне рішення, уявлення реалізації.

Ідеальним варіантом є забезпечення однозначної відповідності між підобластями та обмеженими контекстами. Таким чином, об'єднуються простір завдань та простір рішень, виділяються моделі предметної області у чітко визначені області залежно від поставлених цілей. Якщо система не розробляється з нуля, вона часто є великою грудою бруду, де підобласті перетинаються з обмеженими контекстами.

Тактичне проектування

Використання технічних структурних патернів у коді для відображення результатів стратегічного проектування безпосередньо в коді програми.

Entity

Якщо якесь поняття предметної області є унікальним і відмінним від інших об'єктів у системі, то його моделювання використовується сутність. Такі об'єкти–сутності можуть сильно відрізнятися своєю формою за весь цикл існування, проте їх завжди можна однозначно ідентифікувати та знайти за

запитом. Для цього використовуються унікальні ідентифікатори, створення яких необхідно продумати насамперед під час проектування сутності.

Value Object

Якщо для об'єкта не важлива індивідуальність, якщо вона повністю визначається своїми атрибутами, його слід вважати об'єктом–значенням. Щоб з'ясувати, чи є якесь поняття значенням, необхідно з'ясувати, чи має воно більшість з наступних характеристик:

- Воно вимірює, оцінює чи описує об'єкт предметної області;
- Повинний бути незмінним: при спробі оновлення значення властивостей ми маємо створити та повернути новий екземпляр VO;
- Воно моделює щось концептуально цілісне, поєднуючи пов'язані атрибути в одне ціле;
- При зміні способу виміру або опису його можна повністю замінити;
- Його можна порівнювати з іншими об'єктами за допомогою відношення рівності значень. Два VO вважаються однаковими тоді і лише тоді, коли всі поля VO рівні;
- Воно надає пов'язаним із ним об'єктам функцію без побічних ефектів;
- Складається тільки з інших VO та примітивів (не може містити сутність чи сервіс);
- Повинен утримувати у собі логіку самовалідації;

Domain Service

Використовуючи єдину мову, іменники цієї мови відбиваються в об'єкти, а дієслова відбиваються у поведінки цих об'єктів. Дуже часто існують дієслова або якісь дії, які не можна віднести до якоїсь сутності або якогось об'єкта–значення. Якщо існує така операція в предметній області, її оголошують як Domain Service (вона відрізняється від прикладної служби, яка є клієнтом).

Є три характеристики служб:

- Операція, що виконується службою, належить до концепції предметної області, яка належить жодної з існуючих сутностей;
- Операція виконується над різноманітними об'єктами моделі предметної області;
- Операція немає стану.

Domain Event

Event це те, що сталося у минулому. Логічно, подія предметної області це те, що сталося у конкретній предметної області, і те, що мають знати і що мають реагувати інші частини тієї ж предметної області. Повинні бути іммутабельними (оскільки не можна міняти минуле). Назва повинна описувати подію, що відбулася в минулому.

Module

Модулі всередині моделі є іменованими контейнерами для певної групи предметних об'єктів, тісно пов'язаних один з одним. Їхня мета – ослаблення зв'язків між класами, які знаходяться у різних модулях. Оскільки модулі підходу DDD – це неформальні чи узагальнені розділи, їх слід правильно називати. Вибір їх імен є функцією єдиної мови.

Factory

Деякі агрегати чи сутності можуть бути досить складними. Складний об'єкт неспроможен створювати себе за допомогою конструктора. Ще гірше, коли передають створення складного об'єкта на клієнт. Так, клієнт повинен знати про внутрішню структуру та взаємозв'язки всередині об'єкта. Це порушує інкапсуляцію і прив'язує клієнта до певної реалізації (отже, при зміні об'єкта доведеться змінювати та реалізацію клієнта).

Краще виконувати створення складних агрегатів чи інших об'єктів окремо. І тому використовуються фабрики. Фабрики – елементи програми, обов'язки якого створювати інші об'єкти.

Aggregate

Агрегатом називається кластер із об'єктів сутностей чи значень. Тобто ці об'єкти розглядаються як єдине ціле з погляду зміни даних. У кожного агрегату є корінь Aggregate Root та межа, всередині якої завжди мають бути задоволені інваріанти.

Всі звернення до агрегату повинні здійснюватися через його корінь, який є сутністю з глобально унікальним ідентифікатором. Всі внутрішні об'єкти агрегату мають лише локальну ідентичність, вони можуть посилатися один на одного як завгодно. Зовнішні об'єкти можуть зберігати посилання на корінь, а не на внутрішні об'єкти.

Repository

Репозиторій – область пам'яті, яка призначена для безпечного зберігання вміщених до неї елементів. Репозиторій використовують для агрегатів. Поміщаючи агрегат у відповідний Репозиторій, а потім витягуючи його звідти, ви отримуєте цілісний об'єкт. Якщо агрегат буде змінено, зміни будуть збережені. Якщо агрегат буде видалено, його вже не можна буде витягти.

Кожен агрегат, що передбачає постійне зберігання, має власний Репозиторій. Найчастіше в репозиторії реалізуються методи для вибірки повністю згенерованих агрегатів за якимись критеріями.

Є два типи Repository:

- Орієнтовані на імітацію колекцій;
- Орієнтовані механізм постійного зберігання.

2.4 Використання кешування для оптимізації системи

Кешування – це найбільш часто використовуваний варіант «обміну» пам'яті на швидкість обчислень. Ідея в основі лежить найпростіша – замість того, щоб виконувати ті самі дії багато разів, результат їх виконання можна зберегти в швидку пам'ять, а коли він буде потрібен наступного разу, просто взяти готовий. Це звичайно дає виграш у швидкості, тому що виконується

менше дій, але й веде до збільшення пам'яті, тому що обчисленний результат займає місце.

Є три основних типи кешування з механіки роботи:

- Lazy cache – найпростіший у реалізації тип кешування, часто вбудований у фреймворки. Кеш просто зберігає дані і віддає їх доки не застаріє. Такий тип кешування можна використовувати для даних, які майже не змінюються. Інший варіант використання – робити лінивий кеш з невеликим часом старіння для стабільної роботи під час сплесків навантаження.
- Synchronized cache, синхронізований кеш – клієнт разом з даними виходить мітку останньої зміни і може запитати у постачальника чи не змінилися дані, щоб повторно не запитувати. Такий тип кешування дозволяє завжди мати нові дані, але дуже складний у реалізації. Цей тип кешування не рятує від накладних витрат на спілкування між системами. Тому часто доповнюється іншими типами кешування, щоб пришвидшити роботу.
- Write–through cache, або кеш наскрізного запису – будь–яка зміна даних виконується відразу в сховищі та в кеші. Цей тип кешу може ніколи не старіти, але виникають проблеми з так званою "когерентністю".

Об'єм кеша завжди обмежений. Найчастіше він менший за обсяг даних, які в цей кеш можна покласти. Тому елементи, вміщені в кеш, рано чи пізно будуть витіснені. Сучасні фреймворки для кешування дозволяють дуже гнучко керувати старінням з огляду на пріоритети, час старіння, обсяги даних і т.д.

Простий спосіб підтримки когерентності – примусове старіння (скидання) кеша при зміні даних. Тому збільшення пам'яті для кеша, щоб він менше застарів, не завжди гарна ідея.

Основний параметр, який характеризує систему кешування – це відсоток попадань запитів у кеш.

Часті скидання кешу, кешування даних, що рідко запитуються, недостатній обсяг кеша – все це веде до порожньої витрати оперативної (зазвичай) пам'яті, не підвищуючи ефективність роботи.

Іноді дані змінюються настільки часто і непередбачувано, що кешування не дасть ефекту, відсоток влучень буде близьким до нуля. Але зазвичай дані зчитуються набагато частіше, ніж записуються, тому кеш ефективний.

У нашому випадку Redis використовується для кешування агрегатів. Таке рішення має значно пришвидшити швидкість запитів до системи за рахунок того, що не потрібно буде робити багато запитів до бази даних.

Висновки до розділу 2

У даному розділі було розглянуто створення архітектури back-end сервісу Storiks API та Telegram бота. У якості архітектури було обрано гексагональну архітектуру. Таке рішення дозволяє легко розширювати систему та додавати нові модулі. Також було розглянуто використання Domain Driven Design в системі. Для оптимізації деяких частин систему було обрано стратегію кешування агрегатів в базі даних Redis.

3 ОГЛЯД КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

3.1 Інтерфейс Telegram бота

За допомогою спеціального API Telegram було створено меню у вигляді клавіатури.

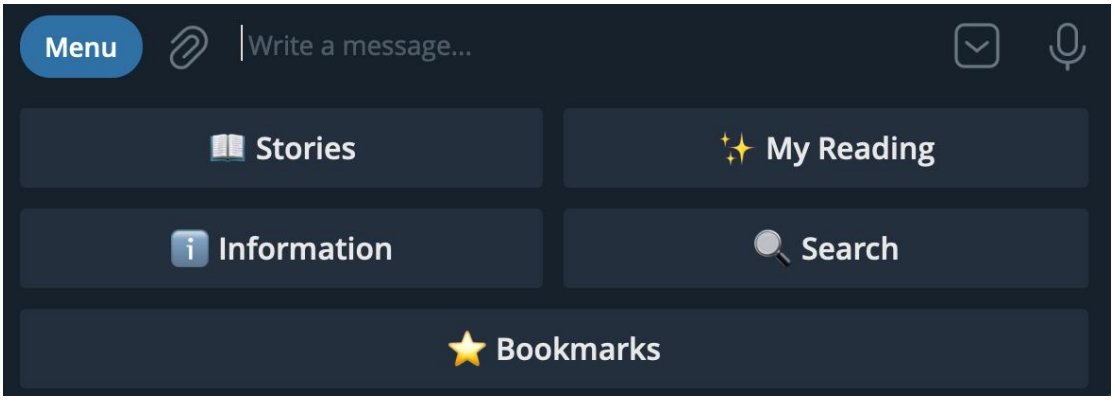

Рисунок 3.1 – Головне меню бота у вигляді клавіатури

Користувач може натиснути на будь–яку кнопку та перейти у відповідний розділ. Натиснувши на кнопку “Stories” користувач переходить до списку історій, доступних для читання. Користувач може обрати будь–яку історію або перейти на іншу сторінку. Також була створена спеціальна кнопка “Back” для того, щоб була можливість повернутися до меню.

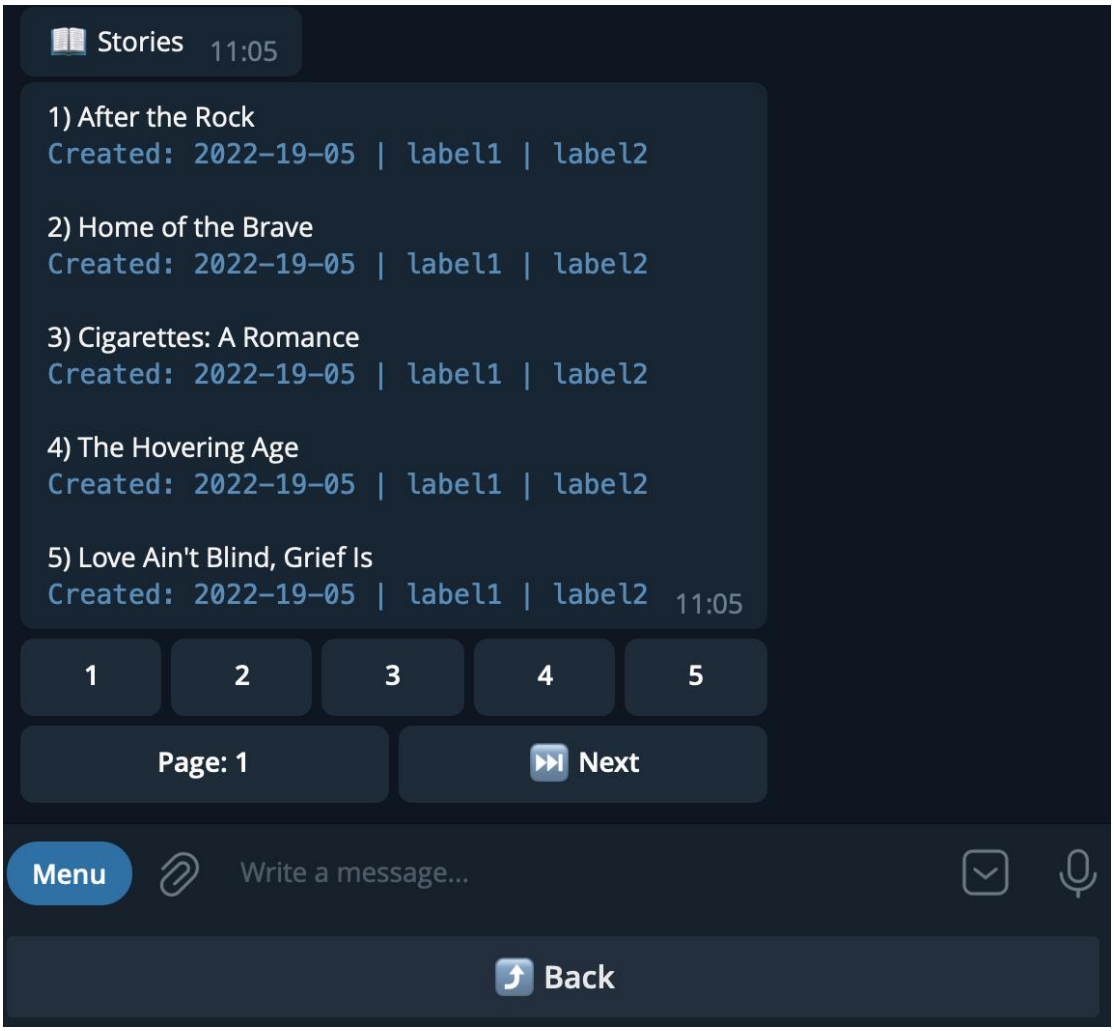

Рисунок 3.2 – Сторінка з доступними користувачам історіями

Вибравши історію, користувачу відображається інформація про історію. На цій сторінці користувач може прочитати опис історії та вирішити, чи хоче він її прочитати. Якщо користувач раніше вже читав цю історію, він може продовжити або почати читати історію спочатку. Також є можливість додати історію до закладок або видалити її із закладок, якщо вона була додана раніше.

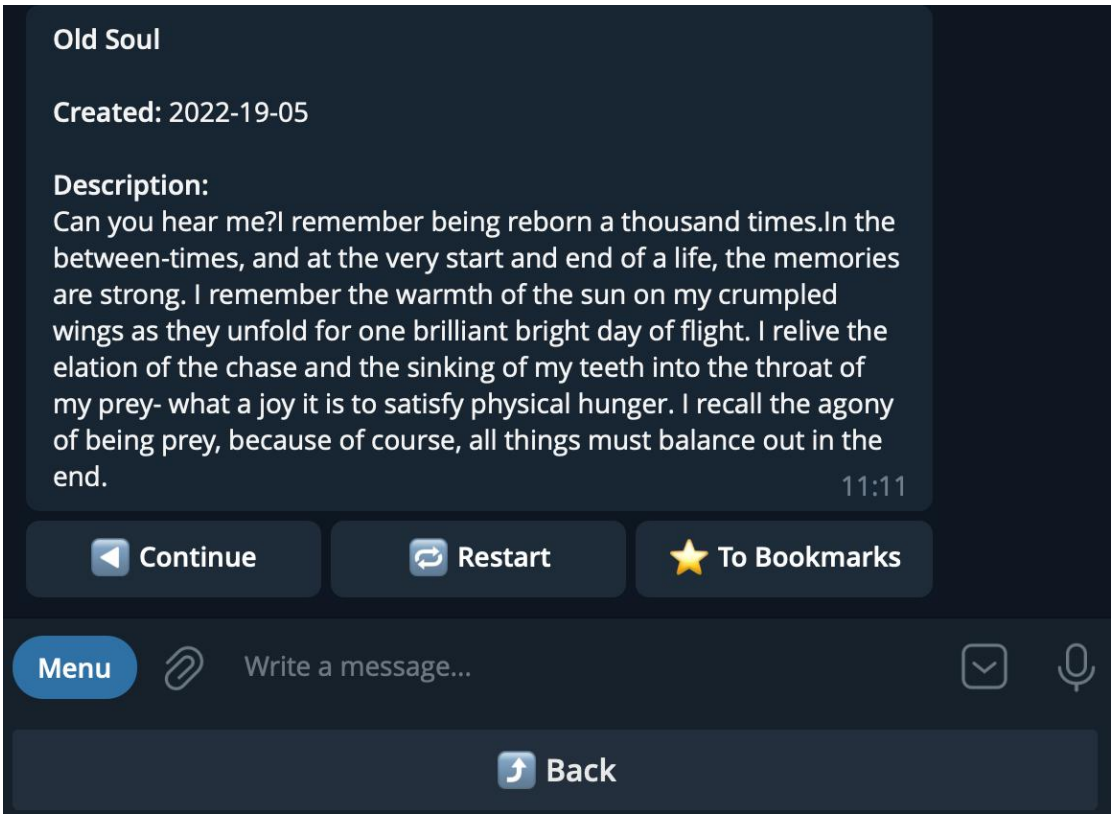

Рисунок 3.3 – Сторінка інформації про обрану історію

При додаванні історії до закладок користувачеві відображається відповідне повідомлення, кнопка “To Bookmarks” змінюється на “Remove Bookmark”.

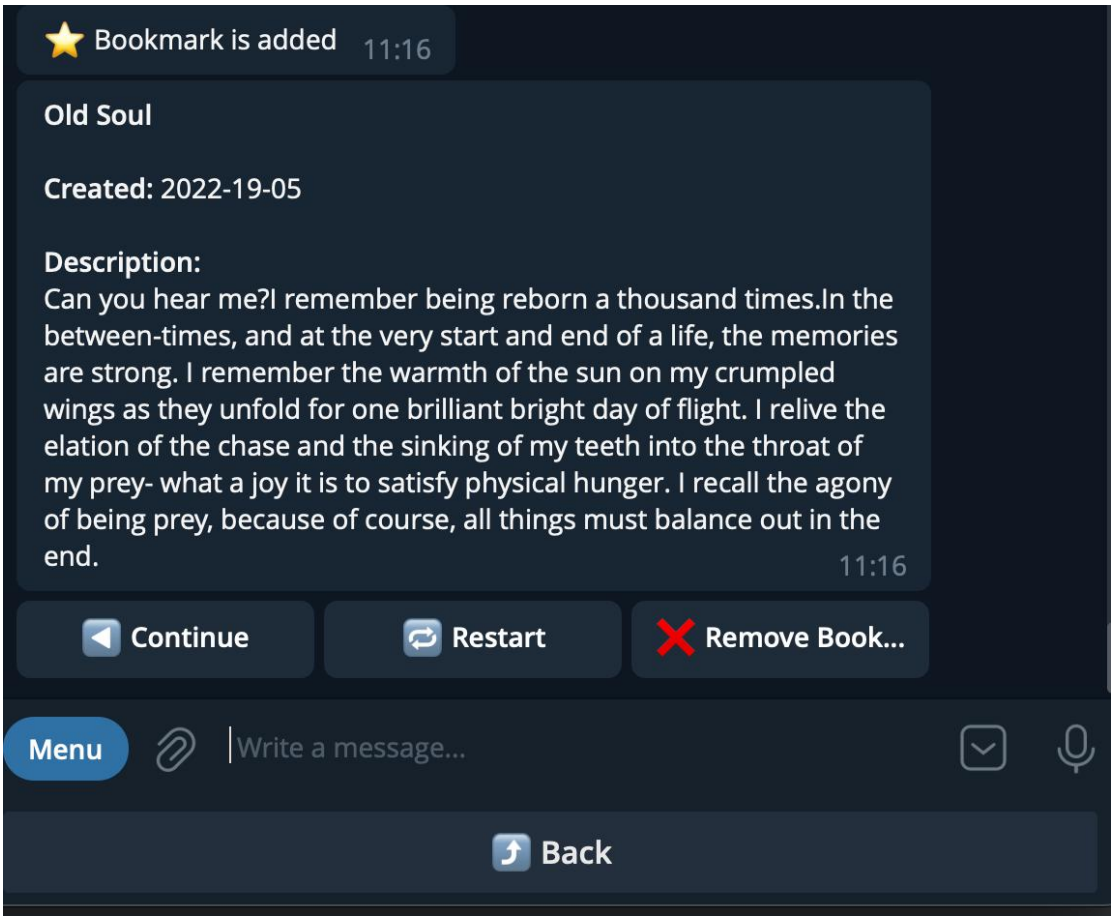

Рисунок 3.4 – Сторінка інформації про обрану історію, яка додана до закладок

При переході в режим читання користувач перенаправляється на місце, де він зупинився або на початок історії. Під час читання кожної частини історії користувач може вибирати альтернативний сценарій шляхом натискання на один із запропонованих варіантів.

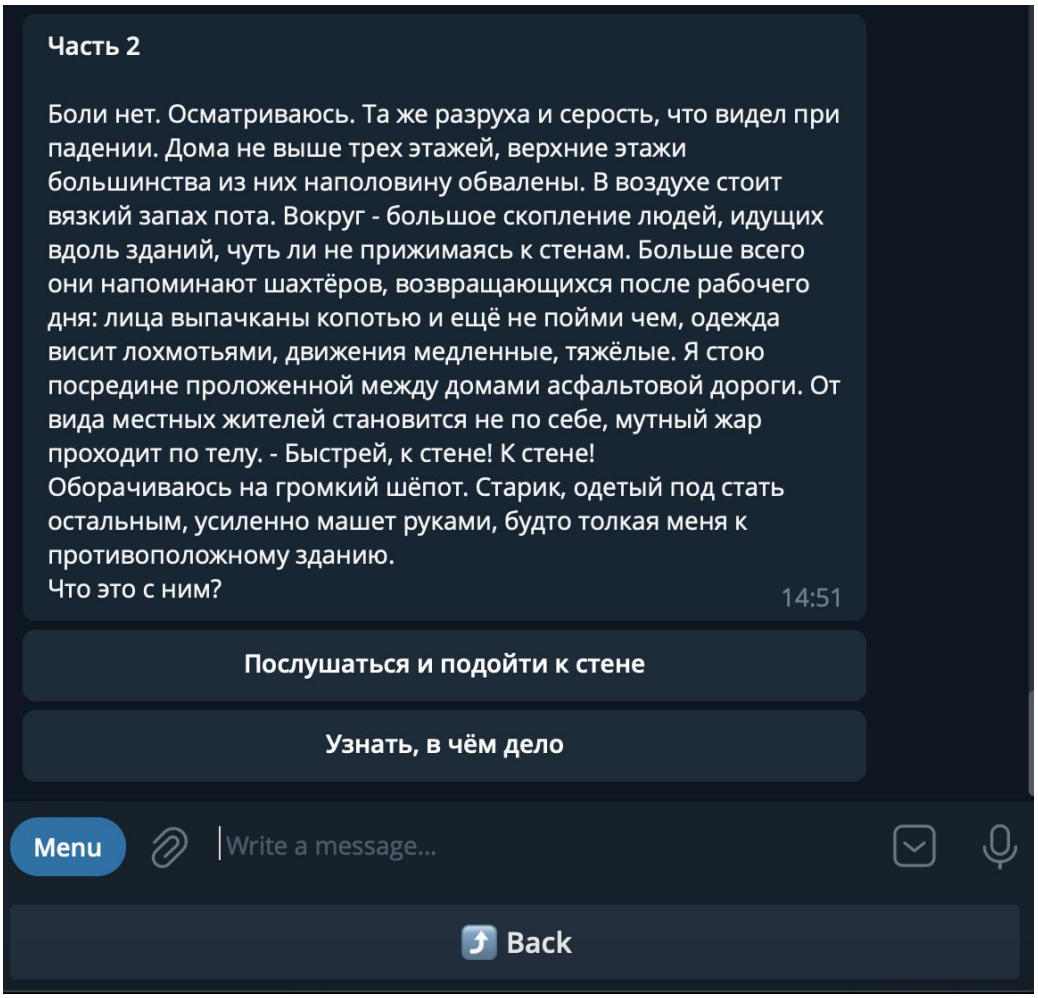

Рисунок 3.5 – Відображення частини історії с декількома можливими варіантами вибору

При досягненні одного з можливих кінців історії у користувача з'являється кнопка "Finish", натиснувши на яку він може закінчити читання історії і перейти в головне меню.

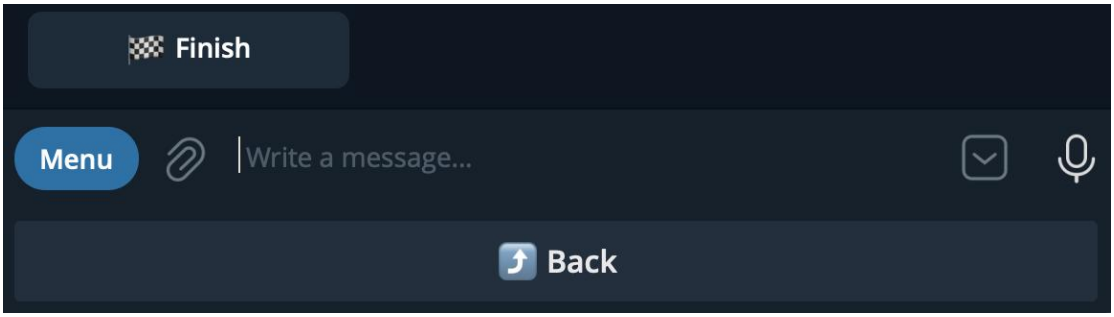

Рисунок 3.6 – Відображення кнопки "фініш" для завершення історії

При натисканні на кнопку "Information" у користувача відображається інформація про бота.

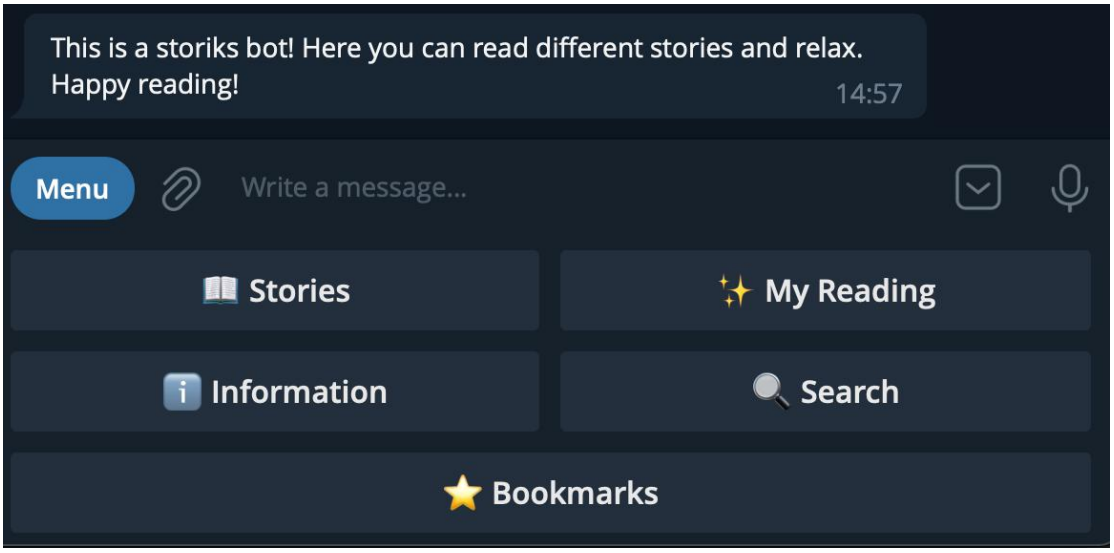

Рисунок 3.7 – Відображення інформації про бота

Також користувач має можливість пошуку історій при натисканні на кнопку “Search”.

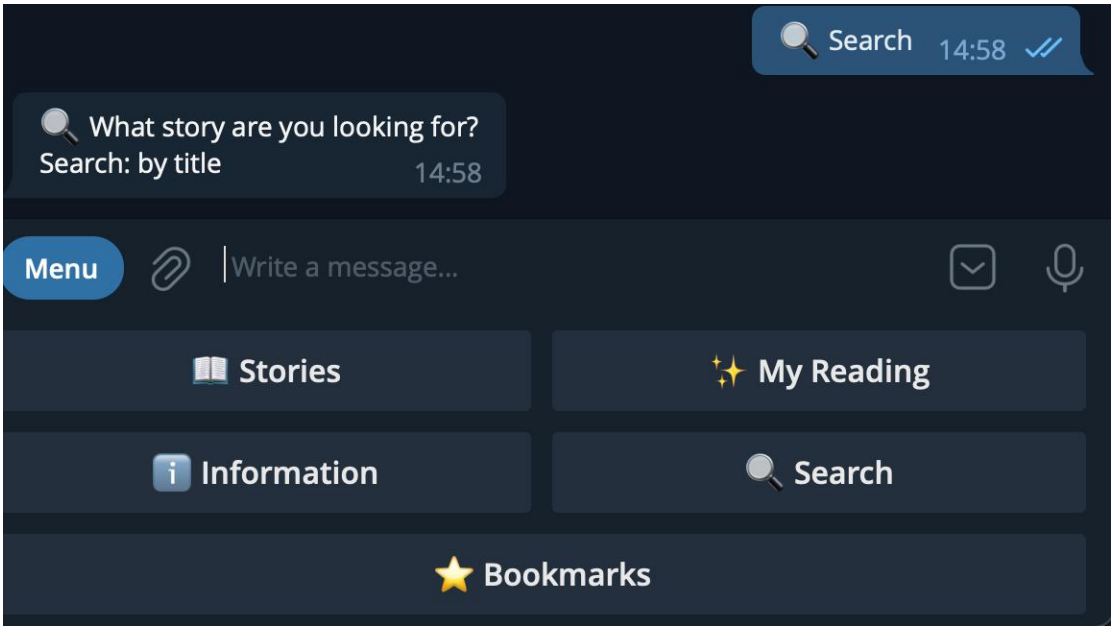

Рисунок 3.8 – Перехід користувача у стан пошуку історій

Якщо ввести ім'я історії та надіслати повідомлення, то у користувача відображається список зі знайденими історіями.

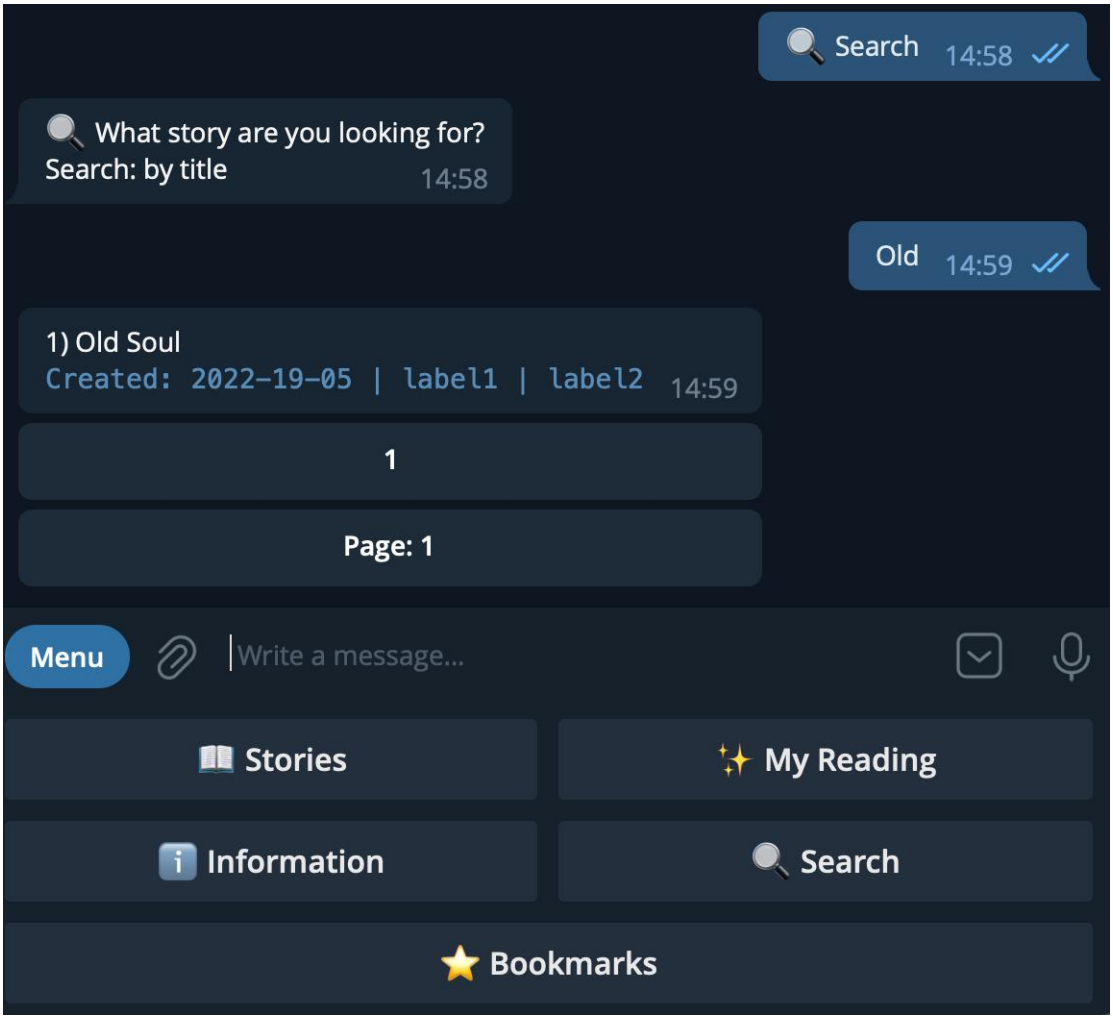

Рисунок 3.9 – Результати пошуку історій

При переході на сторінку “My Reading” користувач може переглянути список розпочатих історій.

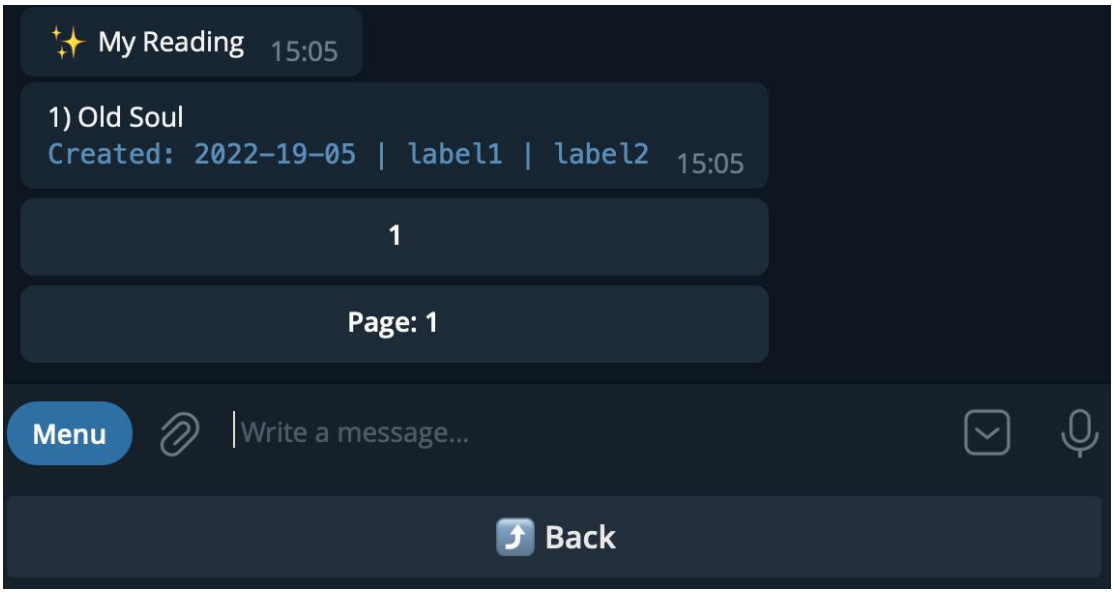

Рисунок 3.10 – Відображення активних історій користувача

Також користувач може переглянути список закладок, натиснувши кнопку“Bookmarks”.

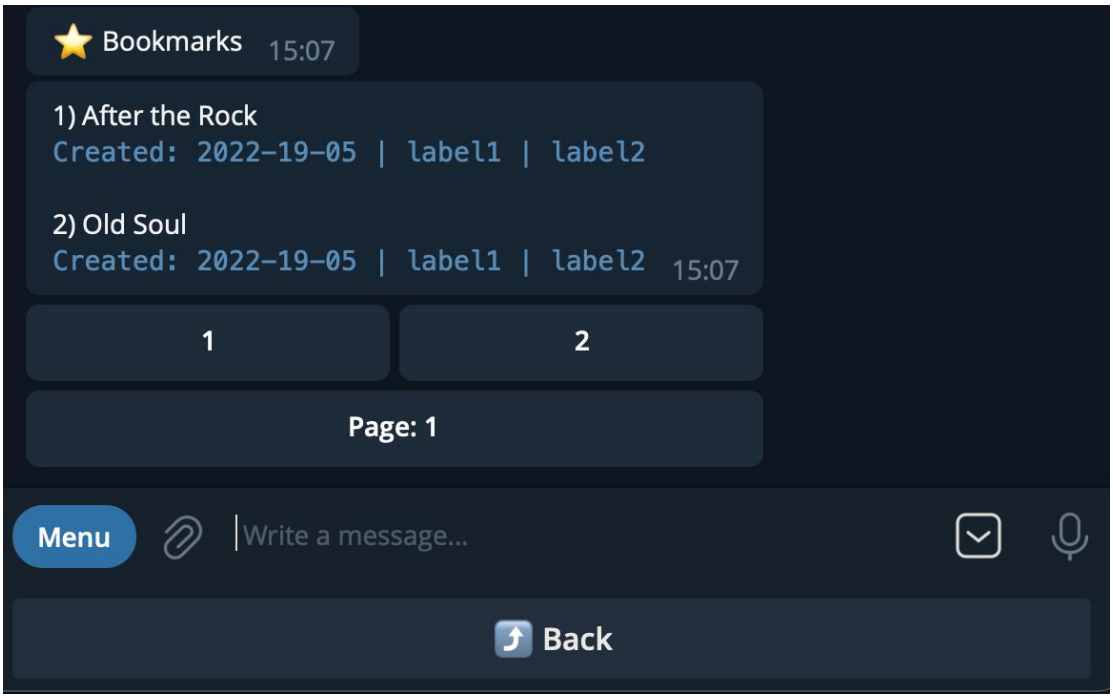

Рисунок 3.11 – Відображення закладок користувача

3.1 Програмний інтерфейс back-end сервісу

Сервіс підтримує програмний інтерфейс у вигляді REST API.

REST API – це спосіб взаємодії сайтів і веб–прикладів із сервером. Його також називають RESTful.

Термін складається з двох абревіатур, які розшифровуються наступним чином. API (Application Programming Interface) – це код, який дозволяє двом додаткам обмінюватися даними із сервером. REST (Representational State Transfer) – це спосіб створення API за допомогою протоколу HTTP.

Технологію REST API застосовують в'їзд, де користується сайтом або веб–приложеніями потрібно надати дані з сервера. В даний час це найбільш поширений спосіб організації API. Він витеснив раніше популярні способи SOAP і WSDL.

У RESTful немає єдиного стандарту роботи: його називають «архітектурним стилем» для операцій із серверами. Такий підхід у 2000 році у своїй дисертації в програміст і дослідник Рой Філдинг, один із розробників протоколу HTTP.

Кожен об’єкт на сервері в HTTP має свій унікальний URL–адрес у строго послідовному форматі.

Принципи REST API

Client–Server. Клиент – це інтерфейс користувача або додатки. У REST API код запитів залишається на стороні клієнта, а код для доступу до даних – на стороні сервера. Це забезпечує організацію API, дає змогу легко переносити користувальницький інтерфейс на платформі та дає можливість краще масштабувати серверне зберігання даних.

Stateless. Сервер не повинен зберігати інформацію про стан клієнта. Кожен запит від клієнта повинен містити тільки ту інформацію, яка потрібна для отримання даних із сервера.

Casheable. В даних запита повинно бути вказано, потрібно чи кешувати дані. Якщо таке вказівка є, клієнт отримає право звертатися до цього буфера при необхідності.

Uniform Interface. Всі дані повинні запитуватися через один URL–адрес стандартними протоколами, наприклад, HTTP. Це спрощує архітектуру сайту чи додатків і робить взаємодію із сервером більш зрозумілою.

Layered System. Сервер RESTful може працювати на різних рівнях, при цьому кожен сервер взаємодіє лише з найближчими рівнями та не пов’язаний із іншими.

Code on Demand. Сервери можуть відправляти код клієнта. Такий загальний код додатків або сайт стає складним тільки при необхідності.

Starting with the Null Style. Користувач знає тільки одну точку входу на сервер. Дальніші можливості по взаємодії забезпечуються сервером.

Незважаючи на відсутність стандартів, при створенні REST API є загальноприняті найкращі практики, наприклад:

- використання захищеного протоколу HTTPS
- використання інструментів для розробки API Blueprint і Swagger
- застосування додатків для тестування Get Postman
- застосування як можна більшої кількості HTTP–кодів (список)
- архівування великих блоків даних

У REST API є 4 методу HTTP, які використовують для дій з об’єктами на серверах:

- GET
- DELETE
- POST
- PUT

Сервіс містить в собі 13 ендпоінтів для роботи.

POST /users використовується для створення нового користувача в системі.

GET /stories/:story_id використовується для отримання історії по унікальному ідентифікатору.

GET /stories використовується для отримання списку історій із підтримкою пагінації.

GET /stories/:story_id/parts/:part_id використовується для отримання частини історії по унікальному ідентифікатору історії та частини.

POST /stories/:story_id/trail/marks використовується для отримання всіх відміток шляху користувача по унікальному ідентифікатору історії.

POST /stories/:story_id/trail використовується для створення шляху по унікальному ідентифікатору історії.

POST /stories/:story_id/trail/position/started використовується для переведення користувача в стартову позицію історії.

GET /stories/:story_id/trail використовується для отримання шляху користувача по унікальному ідентифікатору історії.

GET /trails використовується для отримання всіх шляхів користувача користувача.

POST /stories/:story_id/bookmark використовується для створення закладки.

GET /stories/:story_id/bookmark використовується для отримання закладки користувача по унікальному ідентифікатору історії.

GET /bookmarks використовується для отримання списку закладок із підтримкою пагінації.

DELETE /stories/:story_id/bookmark використовується для видалення закладки користувача по унікальному ідентифікатору історії.

Висновки до розділу 3

У даному розділі було розглянуто користувацький інтерфейс Telegram бота та програмний інтерфейс back-end сервісу Storiks API.

Через те, що інтерфейс був розроблений у вигляді Telegram бота, додаток являється дуже зручним у використанні та доступним майже для кожного. За рахунок цього читач стає більш залученим та зацікавленим під час читання.

За допомогою програмного інтерфейсу у вигляді REST API можна легко додавати нові види клієнтів, такі як веб-додатки, десктоп додатки, мобільні додатки або інтегрувати сервіс з іншими системами.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет імені Петра Могили
Факультет комп’ютерних наук
Кафедра інтелектуальних інформаційних систем

Спеціальний розділ

ОХОРОНА ПРАЦІ

до кваліфікаційної роботи

на тему:

«КОМП’ЮТЕРНА ГРА У ЖАНРІ STORY GAMES ІЗ ПІДТРИМКОЮ ПРОГРАМНОГО ІНТЕРФЕЙСУ»

Спеціальність 122 «Комп’ютерні науки»

122 – БКР – 402.21810216

Виконав студент 4-го курсу, групи 402

_____О. В. Кравець

«___» _____ 2022 р.

Консультант к.т.н., доцент

_____А. О. Алексєєва

«___» _____ 2022 р.

Миколаїв – 2022

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

ВДТ	–	Візуальний дисплейний термінал
ДСанПіН	–	Державні санітарні норми і правила
ДСН	–	Державні санітарні норми
ЕОМ	–	Електронна обчислювальна машина
КПО	–	Коефіцієнт природної освітленості
ПК	–	Персональний комп’ютер
ПЕОМ	–	Персональні електронні обчислювальні машини
ССБП	–	Система стандартів безпеки праці

4 ОХОРОНА ПРАЦІ

Охорона праці при користуванні комп’ютерами

ВСТУП

Сучасний розвиток технічного та технологічного стану виробництва передбачає постійну автоматизацію та оптимізацію виробничих процесів. Сьогодні, напевно, важко уявити компанію, господарська діяльність в якій здійснювалась би без використання комп’ютерної техніки. Через масовий характер робіт, що виконуються працівниками за допомогою комп’ютера, законодавством України чітко врегульовано норми та вимоги до використання комп’ютерної техніки на підприємстві, безпосередньо й охорона праці при роботі з комп’ютером.

Охорона праці – це система правових, соціально–економічних, організаційно–технічних, санітарно-гігієнічних і лікувально-профілактичних заходів та засобів, спрямованих на збереження життя, здоров'я і працездатності людини у процесі трудової діяльності[2]. Норми передбачають, що опромінюється весь організм людини, тоді як на ділі впливу піддається лише верхня частина тулуба. Згадані норми встановлені з розрахунку на кожен вид опромінення в окремо, хоча реально всі поля діють одночасно, а їх комплексний вплив досі не досліджено. Крім того, зображення на екрані динамічно оновлюється, а низька частота оновлення викликає його мерехтіння. При цьому очні і внутрішньоочні м'язи, фокусують погляд, втомлюються від надмірного навантаження. Розвивається зорове стомлення, що сприяє виникненню короткозорості. Тривала робота з комп'ютером вимагає також підвищеної зосередженості, що призводить до появи головного болю, дратівливості, нервової напруги і стресу. Саме тому треба приділити увагу нормам користування екранними пристроями.

4.1. Загальні обов’язки роботодавців

1. Роботодавець повинен поінформувати працівників під розписку про умови праці та наявність на їх робочих місцях небезпечних та шкідливих виробничих факторів (фізичних, хімічних, біологічних, психофізіологічних), які виникають під час роботи з екранними пристроями та ще не усунуто, а також про можливі наслідки їх впливу на здоров’я працівників відповідно до вимог статті 5 Закону України “Про охорону праці”.

2. Роботодавець повинен забезпечити навчання і перевірку знань працівників з питань охорони праці та безпечного використання екранних пристроїв до початку роботи з ними, а також у випадках модифікації та організації роботи обладнання.

3. Роботодавець повинен вжити відповідних заходів, щоб забезпечити відповідність робочого місця працівника до цих Вимог.

4. Під час облаштування робочого місця працівника з екранними пристроями необхідно обирати таке устаткування, яке не створює зайвого шуму та не виділяє надлишкового тепла. Рівні шуму на робочих місцях осіб, які працюють з екранними пристроями, мають відповідати вимогам Санітарних норм виробничого шуму, ультразвуку та інфразвуку ДСН 3.3.6.037–99, затверджених постановою Головного державного санітарного лікаря України від 01 грудня 1999 року № 37.

5. Роботодавець повинен за рахунок тривалості робочої зміни організувати внутрішні регламентовані перерви для відпочинку відповідно до Державних санітарних правил і норм роботи з візуальними дисплейними терміналами електронно–обчислювальних машин ДСанПІН 3.3.2.007–98, затверджених постановою Головного державного санітарного лікаря України від 10 грудня 1998 року № 7 (далі – ДСанПІН 3.3.2.007–98).

6. Роботодавець має забезпечити за свій рахунок проведення медичних оглядів працівників відповідно до вимог Порядку проведення медичних оглядів

працівників певних категорій, затвердженого наказом Міністерства охорони здоров'я України від 21 травня 2007 року № 246, зареєстрованого в Міністерстві юстиції України 23 липня 2007 року за № 846/14113.

За результатами цих оглядів роботодавець за потреби повинен забезпечити виконання відповідних оздоровчих заходів.

7. Роботодавець зобов’язаний за необхідності проводити лабораторні дослідження умов праці працівників з метою виявлення шкідливих і небезпечних факторів виробничого середовища, важкості та напруженості трудового процесу (зокрема щодо виявлення ризиків, пов’язаних із погіршенням зору, порушенням фізичного стану, стресом) та вживати заходів щодо усунення виявлених ризиків відповідно до статті 13 Закону України „Про охорону праці”.

4.2. Вимоги безпеки до робочих місць працівників з екранними пристроями

1. Робочі місця працівників з екранними пристроями мають бути спроектовані так і мати такі розміри, щоб працівники мали простір для зміни робочого положення та рухів.

2. Для забезпечення безпеки та захисту здоров’я працівників усе випромінювання від екранних пристроїв має бути зведене до гранично допустимого рівня (вплив на людину факторів довкілля – шуму, вібрації, забруднювачів, температури тощо, який не спричиняє соматичних або психічних розладів, а також змін стану здоров’я, працездатності, поведінки, що виходять за межі пристосувальних реакцій) з погляду безпеки та охорони здоров’я працівників.

3. Організація робочого місця працівника з екранними пристроями має забезпечувати відповідність усіх елементів робочого місця та їх розташування

ергономічним, антропологічним, психофізіологічним вимогам, а також характеру виконуваних робіт.

4. Штучне освітлення приміщення з робочими місцями, обладнаними відеотерміналами ЕОМ загального та персонального користування, має бути обладнане системою загального рівномірного освітлення. У виробничих та адміністративно-громадських приміщеннях, де переважають роботи з документами, допускається вживати систему комбінованого освітлення (додатково до загального освітлення встановлюються світильники місцевого освітлення)[3].

5. Мікроклімат виробничих приміщень з робочими місцями працівників з екранними пристроями має підтримуватись на постійному рівні та відповідати вимогам Санітарних норм мікроклімату виробничих приміщень ДСН 3.3.6.042–99, затверджених постановою Головного державного санітарного лікаря України від 01 грудня 1999 року № 42 (далі – ДСН 3.3.6.042–99)[6].

6. Робочий стіл або робоча поверхня повинні бути достатнього розміру та мати поверхню з низькою відбивною здатністю, допускати гнучкість під час розміщення екрана, клавіатури, документів і відповідного устаткування.

7. Робоче крісло має бути стійким і дозволяти працівнику з екранними пристроями легко рухатися та займати зручне положення.

Сидіння має регулюватися по висоті, спинка сидіння – як по висоті, так і по нахилу. Слід передбачати підніжку для тих, кому це необхідно для зручності.

4.3. Мінімальні вимоги безпеки під час роботи з екранними пристроями

1. Щодня перед початком роботи необхідно очищати екранні пристрої від пилу та інших забруднень.

2. Після закінчення роботи екранні пристрої слід відключати від електричної мережі.

3. У разі виникнення аварійної ситуації необхідно негайно відключити екранний пристрій від електричної мережі.

4. Не допускається:

– виконувати технічне обслуговування, ремонт і налагодження екранних пристроїв безпосередньо на робочому місці працівника під час роботи з екранними пристроями;

– відключати захисні пристрої, самочинно проводити зміни у конструкції та складі екранних пристроїв або їх технічне налагодження; працювати з екранними пристроями, у яких під час роботи виникають нехарактерні сигнали, нестабільне зображення на екрані та інші несправності.

5. Під час виконання робіт операторського типу, пов’язаних з нервово–емоційним напруженням, у приміщеннях під час роботи з екранними пристроями, на пультах і постах керування технологічними процесами та в інших приміщеннях мають дотримуватися оптимальні умови мікроклімату відповідно до вимог ДСН 3.3.6.042–99.

4.4. Вимоги щодо режиму відпочинку та праці на підприємствах з екранними пристроями

Правила встановлюють внутрішньозмінні режим праці та відпочинку при роботі з електронними пристроями при 8–годинній денній робочій зміні від характеру праці. Для розробників програмного забезпечення при використання електронними пристроями слід призначити регламентовану перерву для відпочинку тривалістю 15 хвилин через кожну годину роботи за персональним комп’ютером. Робітникам, які працюють за комп’ютером, необхідно робити перерву для відпочинку кожну годину або дві під час робочого дня тривалістю 10–15 хвилин. У всіх випадках, коли виробничі обставини не дозволяють використовувати регламентовану перерву, тривалість безперервної роботи з електронними пристроями не повинна бути більше, ніж 4 години[4].

При організації праці, яка пов’язана з електронними пристроями, для збереження здоров’я працівників, ухилення професійних захворювань і підтримки працездатності, повинно передбачатися внутрішньозмінні регламентовані перерви на відпочинок.

Внутрішньозмінні режими праці та відпочинку повинні містити додаткові нетривалі перерви в період, які передують появі об’єктивних в суб’єктивних ознак стомлення та зниження працездатності.

При виконанні роботи, що належать до різної трудової діяльності, за основну роботу з ВДТ треба вважати таку, що займає не менше 50% робочого часу. Під час робочої зміни повинно передбачатися:

– перерви для відпочинку та вживання їжі;

– перерви для відпочинку та особистих потреб згідно з трудовими нормами;

– додаткові перерви, які вводяться для окремих професій з урахуванням особливостей трудової діяльності.

При 12–годинній робочій зміні регламентовані перерви повинні встановлюватися в перші 8 годин роботи, аналогічно перервам при 8–годинній робочій зміні, а протягом останніх 4–х годин роботи, незалежно від характеру трудової діяльності, через кожну годину тривалістю 15 хвилин[5].

4.5. Забруднення повітря на робочих місцях з використанням екранних пристроїв

Чимало досліджень було присвячено визначенню хімічного складу повітря на робочих місцях операторів ВДТ. Багатьма дослідниками було відмічено, що до кінця робочого дня в повітря робочої зони різко зростає концентрація CO_2 яка сягала від 0,12–0,13 до 0,19% (в атмосферному повітрі CO_2 міститься 0,03%).

Основними джерелами озону на комп’ютеризованих місцях є ЕПТ ВДТ та лазерні принтери. З огляду на це, необхідно виключати ВДТ у випадках, коли він не використовується, а лазерний принтер бажано розташовувати подалі від робочого місця оператора. Однак, це додаткові заходи, основним же заходом щодо запобігання несприятливого впливу озону та інших шкідливих речовин на здоров’я операторів є забезпечення функціонування припливно–витяжної вентиляції. Для того, щоб шкідливі речовини не проникали із сусідніх приміщень в приміщеннях з ВДТ необхідно створити деякий надлишковий тиск[6].

Відповідно до ГОСТ 12.1.005–88 вміст озону в повітрі робочої зони не повинен перевищувати 0,1 мг/м3; вміст оксидів азоту – 5 мг/м3; вміст пилу – 4 мг/м3.

Особливу небезпеку щодо впливу на здоров’я представляє підвищена концентрація озону – високотоксичного подразнюючого газу. З цієї причини він був внесений у список речовин, максимальні значення концентрації яких на робочих місцях обмежені та строго визначені. Надзвичайна небезпека озону для здоров’я людини пов’язана з тим, що він належить до так званих радіоміметичних речовин – хімічних сполук, що викликають в живих організмах зміни, схожі з тими, які виникають після дії іонізуючого випромінювання. Тому озон вважається не лише подразнюючою, а й канцерогенною речовиною[7].

4.6. Мінімальні вимоги безпеки до екранних пристроїв

1. Екранні пристрої не мають бути джерелом ризику для працівників.

2. Усе випромінювання, за винятком видимої частини електромагнітного спектра, має бути зведене до незначного рівня з погляду безпеки і охорони здоров’я працівників.

3. Символи на екранних пристроях мають бути чіткими, відповідного розміру. Між символами і рядками символів має бути належна відстань.

4. Зображення на екрані має бути стабільним, без миготінь або інших видів нестабільності.

5. Яскравість та/або контрастність символів має легко регулюватися працівником під час роботи з екранними пристроями, а також швидко адаптуватися до навколишніх умов.

6. Вибираючи екрани, слід надавати перевагу таким екранам, які легко та вільно повертаються і нахиляються відповідно до потреби працівника.

7. За необхідності може використовуватись окрема підставка або регульований стіл для розміщення екрана.

8. Екран не має відблискувати або відбивати світло, щоб не викликати дискомфорту у працівника під час роботи з екранними пристроями.

9. Вибираючи клавіатуру, слід надавати перевагу такій клавіатурі, яка відкидається і є автономною (відокремленою від екрана), щоб працівник міг вибрати зручну робочу позу й уникнути втоми рук (кисті і верхньої частини руки).

10. Поверхня клавіатури має бути матовою, щоб уникнути віддзеркалювання. Розташування клавіш і самі клавіші мають полегшувати роботу із клавіатурою. Позначення клавіш повинно бути достатньо контрастним і розбірливим.

11. Устаткування, яке входить до робочої станції, не має виділяти надлишкового тепла, що може спричинити незручності працівникам під час роботи з екранними пристроями.

12. Під час розробки, вибору, замовлення та модифікації програмного забезпечення, а також під час розробки завдань, що передбачають використання устаткування з екранними пристроями, роботодавець має керуватися таким програмним забезпеченням, яке відповідає розв’язуваним завданням і є

простим у використанні, а де необхідно – адаптованим до рівня знань і досвіду працівника[9].

ВИСНОВКИ ДО РОЗДІЛУ 4

Під час виконання спеціальної частини з охорони праці було досліджено умови праці людини на підприємствах та установах, а також роботу при користуванні електронними пристроями.

На сьогодні існує багато підприємств із дуже шкідливими умовами, які погіршують процес трудової діяльності людини, такі як: шум, вібрація, неправильне освітлення, шкідливі та\або маленькі екрани і т.д., такі підприємства потребують покращення.

На підприємствах мають дотримуватись вимоги та нормативи щодо охорони праці під час трудової діяльності. Дотримання поставлених вимог до працівників та власників підприємств дозволить мінімізувати шкідливі наслідки, які мають вплив на здоров’я людини.

Правила, які прописані у нормативних документах є актуальними на сьогодні через те, що майже в кожному підприємстві є працівники, які працюють з електронними пристроями. Під час роботи за персональним комп’ютером людина швидко втомлюється та від цього страждає опорно–руховий, зоровий апарат, нервова система и в окремих випадках психічне здоров’я.

ВИСНОВКИ

В процесі виконання кваліфікаційної роботи було проаналізовано стек технологій, обраних для розробки додатку, сучасні тенденції в розробці програмного забезпечення, такі як DDD, REST API, кешування, розглянуто плюси та мінуси обраних технологій. Було розроблено архітектуру back-end сервісу та Telegram бота та створено додатки на мові програмування Go відповідно до архітектури. Система була оптимізована шляхом використання бази даних Redis. Для розробки додатку використовувалися підходи, такі як гексагональна архітектура та Domain Driver Design.

Було підвищено зручність процесу читання історій за рахунок надання користувачу можливості впливати на сюжет історій. Це робить користувача більш залученим у процес читання.

Створений додаток дуже простий та комфортний у використанні. За рахунок того, що інтерфейс створений у вигляді Telegram бота, додаток являється доступним майже для кожного. Користувачам не потрібно завантажувати та інсталювати додаткові програми, а достатньо лише увімкнути бота у месенджері.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Go Documentation: веб-сайт. URL: https://go.dev/doc/ (дата звернення: 22.04.2022).
2. Redis Documentation: веб-сайт. URL: https://redis.io/docs/ (дата звернення: 13.04.2022).
3. Мова програмування Go / Брайан У. Керниган, Алан А. А. Донован – Диалектика-Вильямс, 2016 – 432 pp.
4. Чистий код / Роберт С. Мартін – Independently published, 2019. – 368 pp.
5. Чиста архітектура / Роберт С. Мартін – Independently published, 2019. – 368 pp.
6. Go in Action / William Kennedy, Brian Ketelsen, Erik St. Martin – Manning, 2015. – 264 pp.
7. Telegram API Documentation: веб-сайт. URL: https://core.telegram.org/bots/api (дата звернення: 27.04.2022).
8. Golang для профи. Работа с сетью, многопоточность, структуры данных и машинное обучение с Go / Цукалос Михалис – Питер, 2020. – 720 pp.
9. Head First. Изучаем Go / Макгаврен Джей – Питер, 2020. – 544 pp.
10. MongoDB Documentation: веб-сайт. URL: https://www.mongodb.com/docs/ (дата звернення: 01.05.2022).
11. Облачный GO / Титмус Мэтью А. – ДМК Пресс, 2022. – 418 pp.
12. Go на практике / Фарина Мэтт Мэтт, Butcher Matt – ДМК Пресс, 2017. – 374 pp.
13. PostgreSQL Documentation: веб-сайт. URL: https://www.postgresql.org/docs/ (дата звернення: 01.04.2022).
14. Go Design Patterns / Mario Castro Contreras – Packt Publishing, 2017. – 402 pp.

15. Hands-On Software Architecture with Golang / Jyotiswarup Raiturkar – Packt Publishing, 2018. – 500 pp.

16. Building Microservices with Go. Develop seamless, efficient, and robust microservices with Go / Nic Jackson – Packt Publishing, 2017. – 358 pp.

17. СН 4088-86. Санітарні норми мікроклімату виробничих приміщень URL: http://docs.cntd.ru/document/901710059 (дата звернення: 22.05.2022).

18. СНиП 2.09.04.-87. Адміністративні і побутові будівлі URL: https://dnaop.com/html/54074/doc -СНиП_2.09.04-87_ (дата звернення: 20.05.2022).

19. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин ДСанПІН 3.3.2.007-98 URL: https://zakon.rada.gov.ua/rada/show/v0007282-98#Text (дата звернення: 20.05.2022).

20. Державні санітарні правила і норми роботи з ВДТ ЕОМ ДСанПІН 3.3.2.007-98 URL: http://mozdocs.kiev.ua/view.php?id=2445 (дата звернення: 20.05.2022).

21. СНиП II-4-79. Природне і штучне освітлення URL: https://dnaop.com/html/45036/doc- СНиП_II-4-79_ (дата звернення: 20.05.2022).

22. СН 4557-88. Санітарні норми ультрафіолетового випромінювання URL: https://dnaop.com/html/2299/doc -СН_4557-88 (дата звернення: 22.05.2022).

23. Санітарно – гігієнічним нормам №2152-80 URL: https://dnaop.com/html/2296/doc -ГН_2152-80 (дата звернення: 22.05.2022).

24. ДСН 3.3.6.042-99. Санітарні норми мікроклімату виробничих приміщень URL: https://dnaop.com/html/34094/doc -ДСН_3.3.6.042-99 (дата звернення: 22.05.2022).

25. Про затвердження Вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями URL: http://zakon3.rada.gov.ua/laws/show/z0508-18 (дата звернення: 20.05.2022).

26. ГОСТ 12.1.005-88.ССБП URL:

http://docs.cntd.ru/document/1200003608 (дата звернення: 22.05.2022).

ДОДАТОК А

Код сервісу Storiks API

```
type SqlStorage struct {
  db *sql.DB
}

func (r SqlStorage) User(ctx context.Context, id uuid.UUID) (result domain.User, err error) {
  var (
    userId uuid.UUID
  )

  err = r.db.QueryRow("select * from users where id = $1", id).Scan(&userId)

  if err != nil && errors.Is(err, sql.ErrNoRows) {
    return result, derrors.NotFound("user is not found")
  }

  result = domain.NewUser(userId)

  return
}

func (r SqlStorage) CreateUser(ctx context.Context, user domain.User) error {
  result, err := r.db.Exec(
    "insert into users (id) values ($1);", user.Id(),
  )

  if err != nil {
    return err
  }
```

```
  affectedRows, err := result.RowsAffected()

  if err != nil {
    return err
  }

  if affectedRows != 1 {
    return derrors.Unknown("something went wrong during creating a user")
  }

  return nil
}

func (r SqlStorage) Bookmark(ctx context.Context, id domain.BookmarkId) (result
domain.Bookmark, err error) {
  var (
    storyId, userId uuid.UUID
  )

  err = r.db.QueryRowContext(ctx, "select * from bookmarks where user_id = $1 and story_id =
$2", id.UserId(), id.StoryId()).
    Scan(&userId, &storyId)

  if err != nil && errors.Is(err, sql.ErrNoRows) {
    return result, derrors.NotFound("bookmark is not found")
  }

  result = domain.NewBookmark(id)

  return
}
```

```
func (r SqlStorage) CreateBookmark(ctx context.Context, b domain.Bookmark) error {
  result, err := r.db.Exec(
    "insert into bookmarks (user_id, story_id) values ($1, $2);", b.Id().UserId(), b.Id().StoryId())

  if err != nil {
    return err
  }

  affectedRows, err := result.RowsAffected()

  if err != nil {
    return err
  }

  if affectedRows != 1 {
    return derrors.Unknown("something went wrong during creating a bookmark")
  }

  return nil
}

func (r SqlStorage) Bookmarks(ctx context.Context, userId uuid.UUID, o domain.Offset, l
domain.Limit) (result domain.Span[domain.Bookmark], err error) {
  rows, err := r.db.QueryContext(ctx, "select * from bookmarks where user_id = $1 limit $2 offset
$3;", userId, l+1, o)

  if err != nil {
    return
  }

  defer rows.Close()
```

```
  var (
    storyId uuid.UUID
  )

  bookmarks := make([]domain.Bookmark, 0)

  for rows.Next() {
    err = rows.Scan(&userId, &storyId)

    if err != nil {
      return
    }

    bookmarks = append(bookmarks, domain.NewBookmark(domain.NewBookmarkId(userId,
storyId)))
  }

  err = rows.Err()

  if err != nil {
    return
  }

  hasMore := len(bookmarks) == int(l+1)

  if hasMore {
    bookmarks = bookmarks[:len(bookmarks)-1]
  }

  result = domain.NewSpan(bookmarks, hasMore)
```

```
  return
}

func (r SqlStorage) DeleteBookmark(ctx context.Context, id domain.BookmarkId) error {
  _, err := r.db.Exec("delete from bookmarks where story_id = $1 and user_id = $2;", id.StoryId(),
id.UserId())
  return err
}

func (r SqlStorage) StoryAggregate(ctx context.Context, id uuid.UUID) (result
domain.StoryAggregate, err error) {
  var (
    storyId, ownerId   uuid.UUID
    title, description string
    createdAt          time.Time
    status             domain.StoryStatus
    labels             json.RawMessage
  )

  err = r.db.QueryRowContext(ctx, "select * from stories where id = $1", id).
    Scan(&storyId, &title, &description, &createdAt, &status, &ownerId, &labels)

  if err != nil && errors.Is(err, sql.ErrNoRows) {
    return result, derrors.NotFound("StoryAggregate is not found")
  }

  var l []string
  err = json.Unmarshal(labels, &l)

  if err != nil {
    return
```

```
  }

  parts, err := getParts(ctx, id, r.db)

  if err != nil {
    return
  }

  result, err = domain.NewStoryAggregate(
    storyId,
    title,
    description,
    createdAt,
    status,
    ownerId,
    l,
    parts)

  return
}

func getParts(ctx context.Context, storyId uuid.UUID, db *sql.DB) (result []domain.Part, err error) {
  result = make([]domain.Part, 0)
  rows, err := db.QueryContext(ctx, "select * from story_parts where story_id = $1", storyId)

  if err != nil {
    return
  }

  defer rows.Close()
```

```
parts := make([]struct {
  id, storyId    uuid.UUID
  title, content string
  partType       domain.PartType
}, 0)

for rows.Next() {
  part := struct {
    id, storyId    uuid.UUID
    title, content string
    partType       domain.PartType
  }{}

  err = rows.Scan(&part.id, &part.title, &part.content, &part.partType, &part.storyId)

  if err != nil {
    return
  }

  parts = append(parts, part)
}

err = rows.Err()

if err != nil {
  return
}

partsIds := make([]string, 0)

for _, part := range parts {
```

```
    partsIds = append(partsIds, part.id.String())
  }

  param := "{" + strings.Join(partsIds, ",") + "}"

  rows, err = db.QueryContext(ctx, "select * from story_links where source_part_id = ANY
($1::uuid[]) or target_part_id = ANY ($1::uuid[])", param)

  if err != nil {
    return
  }

  defer rows.Close()

  links := make([]struct {
    sourcePartId, targetPartId uuid.UUID
    title                      string
  }, 0)

  for rows.Next() {
    link := struct {
      sourcePartId, targetPartId uuid.UUID
      title                      string
    }{}
    err = rows.Scan(&link.title, &link.sourcePartId, &link.targetPartId)

    if err != nil {
      return
    }

    links = append(links, link)
```

```
}

err = rows.Err()

if err != nil {
  return
}

for _, p := range parts {
  lks := make([]domain.Link, 0)

  for _, l := range links {
    if p.id == l.sourcePartId {
      link, err := domain.NewLink(l.title, l.targetPartId)

      if err != nil {
        return result, err
      }

      lks = append(lks, link)
    }
  }

  part, err := domain.NewPart(
    p.id,
    p.title,
    p.content,
    p.partType,
    lks)

  if err != nil {
```

```
        return result, err
      }

      result = append(result, part)
    }

    return
}

func (r SqlStorage) Stories(ctx context.Context, search string, o domain.Offset, l domain.Limit)
(result domain.Span[domain.Story], err error) {
    var (
      stories          = make([]domain.Story, 0)
      storyId
     uuid.UUID
      title, description string
      createdAt         time.Time
      status           domain.StoryStatus
      ownerId           uuid.UUID
      labels           json.RawMessage
    )

    rows, err := r.db.QueryContext(ctx, "select * from stories where starts_with(title, $3) limit $1
offset $2", l+1, o, search)

    if err != nil {
      return
    }

    defer rows.Close()

    for rows.Next() {
```

```
    err = rows.Scan(&storyId, &title, &description, &createdAt, &status, &ownerId, &labels)

    if err != nil {
      return
    }

    var l []string
    err = json.Unmarshal(labels, &l)

    if err != nil {
      return
    }

    s, _ := domain.NewStory(
      storyId,
      title,
      description,
      createdAt,
      status,
      ownerId,
      l)

    stories = append(stories, s)
  }

  hasMore := len(stories) == int(l+1)

  if hasMore {
    stories = stories[:len(stories)-1]
  }
```

```
  result = domain.NewSpan(stories, hasMore)

  err = rows.Err()

  return
}

func (r SqlStorage) Story(ctx context.Context, id uuid.UUID) (result domain.Story, err error) {
  var (
    storyId          uuid.UUID
    title, description string
    createdAt        time.Time
    status           domain.StoryStatus
    ownerId          uuid.UUID
    labels           json.RawMessage
  )

  err = r.db.QueryRowContext(ctx, "select * from stories where id = $1", id).
    Scan(&storyId, &title, &description, &createdAt, &status, &ownerId, &labels)

  if err != nil && errors.Is(err, sql.ErrNoRows) {
    return result, derrors.NotFound("story is not found")
  }

  var l []string
  err = json.Unmarshal(labels, &l)

  if err != nil {
    return
  }
```

```
  result, err = domain.NewStory(storyId, title, description, createdAt, status, ownerId, l)

  return
}

func (r SqlStorage) Trails(ctx context.Context, userId uuid.UUID, o domain.Offset, l domain.Limit) (
  result domain.Span[domain.Trail],
  err error,
) {
  rows, err := r.db.QueryContext(ctx, "select * from marks where user_id = $1 limit $2 offset $3;",
userId, l+1, o)

  if err != nil {
    return
  }

  defer rows.Close()

  marks := make(map[uuid.UUID][]domain.Mark, 0)

  var (
    storyId, partId uuid.UUID
    sequenceNumber  int
  )

  for rows.Next() {
    err = rows.Scan(&userId, &storyId, &partId, &sequenceNumber)

    if err != nil {
      return
    }
```

```
    _, ok := marks[storyId]

    if !ok {
      marks[storyId] = make([]domain.Mark, 0)
    }

    mark, err := domain.NewMark(partId, sequenceNumber)

    if err != nil {
      return result, err
    }

    marks[storyId] = append(marks[storyId], mark)
  }

  trails := make([]domain.Trail, 0)
  for k, v := range marks {
    s, err := r.StoryAggregate(ctx, k)

    if err != nil {
      return result, err
    }

    t := domain.NewTrail(domain.NewTrailId(userId, s.Id()), v)
    trails = append(trails, t)
  }

  err = rows.Err()

  if err != nil {
```

```
    return
  }

  hasMore := len(trails) == int(l+1)

  if hasMore {
    trails = trails[:len(trails)-1]
  }

  result = domain.NewSpan(trails, hasMore)

  return
}

func (r SqlStorage) Trail(ctx context.Context, trailId domain.TrailId) (domain.Trail, error) {
  rows, err := r.db.QueryContext(ctx, "select * from marks where user_id = $1 and story_id = $2;",
trailId.UserId(), trailId.StoryId())

  if err != nil {
    return domain.Trail{}, err
  }

  defer rows.Close()

  var (
    userId, storyId, partId uuid.UUID
    sequenceNumber          int
  )

  marks := make([]domain.Mark, 0)
```

```
  for rows.Next() {
    err = rows.Scan(&userId, &storyId, &partId, &sequenceNumber)

    if err != nil {
      return domain.Trail{}, err
    }

    mark, err := domain.NewMark(partId, sequenceNumber)

    if err != nil {
      return domain.Trail{}, err
    }

    marks = append(marks, mark)
  }

  err = rows.Err()

  if err != nil {
    return domain.Trail{}, err
  }

  if len(marks) == 0 {
    return domain.Trail{}, derrors.NotFound("trail is not found")
  }

  return domain.NewTrail(trailId, marks), err
}

func (r SqlStorage) CreateTrail(ctx context.Context, trail domain.Trail) error {
  tx, err := r.db.Begin()
```

```
if err != nil {
  return err
}

defer tx.Rollback()

for _, mark := range trail.Marks() {
  result, err := r.db.ExecContext(ctx,
    "insert into marks (user_id, story_id, part_id, sequence_number) values ($1, $2, $3, $4);",
    trail.Id().UserId(),
    trail.Id().StoryId(),
    mark.PartId(),
    mark.SequenceNumber(),
  )

  if err != nil {
    return err
  }

  affectedRows, err := result.RowsAffected()

  if err != nil {
    return err
  }

  if affectedRows != 1 {
    return derrors.Unknown("something went wrong during creating a trail")
  }
}
```

```
  return tx.Commit()


  return nil
}


func (r SqlStorage) UpdateTrail(ctx context.Context, trail domain.Trail) error {
  tx, err := r.db.Begin()


  if err != nil {
    return err
  }


  defer tx.Rollback()


  for _, mark := range trail.Marks() {
    _, err = r.db.ExecContext(
      ctx,
      "INSERT INTO marks (user_id, story_id, part_id, sequence_number) VALUES ($1, $2, $3, $4)
ON CONFLICT DO NOTHING;",
      trail.Id().UserId(),
      trail.Id().StoryId(),
      mark.PartId(),
      mark.SequenceNumber())


    if err != nil {
      return err
    }
  }


  return tx.Commit()
```

```
  return nil
}

func (r SqlStorage) DeleteTrail(ctx context.Context, id domain.TrailId) error {
  _, err := r.db.Exec("delete from marks where story_id = $1 and user_id = $2;", id.StoryId(),
id.UserId())
  return err
}

func NewSqlStorage(db *sql.DB) SqlStorage {
  return SqlStorage{
    db,
  }
}

var (
  connection *sql.DB
  once       sync.Once
)

func CreateDbConnection(driver, dbSource string) (err error) {
  once.Do(func() {
    if connection, err = sql.Open(driver, dbSource); err != nil {
      return
    }

    err = connection.Ping()
  })

  return
}
```

```
func GetConnection() *sql.DB {
  return connection
}

type RedisStorage struct {
  c          *redis.Client
  sqlStorage storage.SqlStorage
}

func NewRedisStorage(c *redis.Client, s storage.SqlStorage) domain.Storage {
  return RedisStorage{c, s}
}

func (s RedisStorage) Bookmark(ctx context.Context, id domain.BookmarkId) (domain.Bookmark,
error) {
  return s.sqlStorage.Bookmark(ctx, id)
}
func (s RedisStorage) Bookmarks(ctx context.Context, userId uuid.UUID, o domain.Offset, l
domain.Limit) (domain.Span[domain.Bookmark], error) {
  return s.sqlStorage.Bookmarks(ctx, userId, o, l)
}

func (s RedisStorage) CreateBookmark(ctx context.Context, b domain.Bookmark) error {
  return s.CreateBookmark(ctx, b)
}

func (s RedisStorage) DeleteBookmark(ctx context.Context, id domain.BookmarkId) error {
  return s.sqlStorage.DeleteBookmark(ctx, id)
}
```

```
func (s RedisStorage) StoryAggregate(ctx context.Context, id uuid.UUID) (domain.StoryAggregate, error) {
  val, err := s.c.Get(ctx, fmt.Sprintf("stories/%s", id)).Bytes()
  story := domain.StoryAggregate{}

  if err != nil && err == redis.Nil {
    story, err = s.sqlStorage.StoryAggregate(ctx, id)

    if err != nil {
      return story, err
    }

    err = s.c.Set(ctx, fmt.Sprintf("stories/%s", id), story, time.Minute*10).Err()

    return story, err
  }

  if err != nil {
    return story, err
  }

  err = json.Unmarshal(val, &story)

  return story, err
}

func (s RedisStorage) Story(ctx context.Context, id uuid.UUID) (domain.Story, error) {
  return s.sqlStorage.Story(ctx, id)
}

func (s RedisStorage) Stories(ctx context.Context, title string, o domain.Offset, l domain.Limit) (domain.Span[domain.Story], error) {
```

```
  return s.sqlStorage.Stories(ctx, title, o, l)
}

func (s RedisStorage) Trail(ctx context.Context, id domain.TrailId) (domain.Trail, error) {
  return s.sqlStorage.Trail(ctx, id)
}

func (s RedisStorage) Trails(ctx context.Context, userId uuid.UUID, o domain.Offset, l
domain.Limit) (domain.Span[domain.Trail], error) {
  return s.sqlStorage.Trails(ctx, userId, o, l)
}

func (s RedisStorage) CreateTrail(ctx context.Context, t domain.Trail) error {
  return s.sqlStorage.CreateTrail(ctx, t)
}

func (s RedisStorage) UpdateTrail(ctx context.Context, t domain.Trail) error {
  return s.sqlStorage.UpdateTrail(ctx, t)
}

func (s RedisStorage) DeleteTrail(ctx context.Context, id domain.TrailId) error {
  return s.sqlStorage.DeleteTrail(ctx, id)
}

func (s RedisStorage) User(ctx context.Context, id uuid.UUID) (domain.User, error) {
  return s.sqlStorage.User(ctx, id)
}

func (s RedisStorage) CreateUser(ctx context.Context, u domain.User) error {
  return s.sqlStorage.CreateUser(ctx, u)
}
```

```
var (
  client *redis.Client
  once   sync.Once
)

func CreateClient(connection string) (err error) {
  once.Do(func() {
    client = redis.NewClient(&redis.Options{
      Addr:     connection,
      Password: "",
      DB:       0,
    })

    _, err = client.Ping(context.Background()).Result()
  })

  return
}

func GetClient() *redis.Client {
  return client
}



func Run() error {
  c := config.Get()
  router := gin.Default()

  v1.RegisterHandlers(router, c)
```

```
  err := postgres.CreateDbConnection(c.DbDriver, c.DbHost)

  if err != nil {
    return err
  }

  err = redis.CreateClient(c.RedisConnection)

  if err != nil {
    return err
  }

  return router.Run(c.ServerAddress)
}

type Config struct {
  DbDriver       string
  DbHost         string
  ServerAddress  string
  AuthToken      string
  RedisConnection string
}

var (
  config Config
  once   sync.Once
)

func Get() *Config {
  once.Do(func() {
    viper.SetConfigFile("./../../.env")
```

```
    if err := viper.ReadInConfig(); err != nil {
      return
    }

    config = Config{
      DbDriver:      viper.GetString("DB_DRIVER"),
      DbHost:        viper.GetString("DB_SOURCE"),
      ServerAddress:  viper.GetString("SERVER_ADDRESS"),
      AuthToken:     viper.GetString("AUTH_TOKEN"),
      RedisConnection: viper.GetString("REDIS_CONNECTION"),
    }

    viper.AddConfigPath("./../../configs")
    viper.SetConfigName("main")
    viper.SetConfigType("yaml")

    if err := viper.ReadInConfig(); err != nil {
      return
    }
  })
  return &config
}

type BookmarkDto struct {
  StoryId uuid.UUID `json:"story_id"`
}

func NewBookmarkDto(b domain.Bookmark) BookmarkDto {
  return BookmarkDto{
    StoryId: b.Id().StoryId(),
```

```
  }
}

type LinkDto struct {
  Title       string    `json:"title"`
  TargetPartId uuid.UUID `json:"target_part_id"`
}

func NewLinkDto(link domain.Link) LinkDto {
  return LinkDto{
    link.Title(),
    link.TargetPartId(),
  }
}
type PartDto struct {
  Id      uuid.UUID      `json:"id"`
  Title   string         `json:"title"`
  Content string         `json:"content"`
  Type    domain.PartType `json:"type"`
  Links   []LinkDto      `json:"links"`
}

func NewPartDto(part domain.Part) PartDto {
  links := make([]LinkDto, 0)
  for _, link := range part.Links() {
    links = append(links, NewLinkDto(link))
  }
  return PartDto{
    part.Id(),
    part.Title(),
    part.Content(),
```

```
    part.PartType(),
    links,
  }
}

type SpanDto[T any] struct {
  Value     []T  `json:"value"`
  MoreExist bool `json:"more_exist"`
}

func NewSpanDto[T any, K any](value domain.Span[K], mapping func(K) T) SpanDto[T] {
  items := make([]T, 0)
  for _, item := range value.Value() {
    items = append(items, mapping(item))
  }
  return SpanDto[T]{
    items,
    value.MoreExist(),
  }
}

type StoryDto struct {
  Id          uuid.UUID          `json:"id"`
  Title       string             `json:"title"`
  Description string             `json:"description"`
  CreatedAt   time.Time          `json:"created_at"`
  Status      domain.StoryStatus `json:"status"`
  OwnerId     uuid.UUID          `json:"owner_id"`
  Labels      domain.Labels      `json:"labels"`
}
```

```
func NewStoryDto(s domain.Story) StoryDto {
  return StoryDto{
    Id:          s.Id(),
    Title:       s.Title(),
    Description: s.Description(),
    CreatedAt:   s.CreatedAt(),
    Status:      s.Status(),
    OwnerId:     s.OwnerId(),
    Labels:      s.Labels(),
  }
}

type TrailDto struct {
  StoryId       uuid.UUID `json:"story_id"`
  CurrentPartId uuid.UUID `json:"current_part_id,omitempty"`
}

func NewTrailDto(trail domain.Trail) TrailDto {
  mark, _ := trail.CurrentMark()
  return TrailDto{
    StoryId:       trail.Id().StoryId(),
    CurrentPartId: mark.PartId(),
  }
}

type UserDto struct {
  Id    uuid.UUID `json:"id"`
  Token string    `json:"token"`
}

func NewUserDto(s domain.User, t string) UserDto {
```

```
  return UserDto{
    Id:    s.Id(),
    Token: t,
  }
}

type GetListRequest struct {
  Offset domain.Offset `form:"offset" binding:"gte=0"`
  Limit  domain.Limit  `form:"limit" binding:"required,gte=5,lte=100"`
}

func GetBookmarks(ctx *gin.Context) {
  var req GetListRequest
  if err := ctx.ShouldBindQuery(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewBookmarksService(s)

  userId, hasId := h.UserId(ctx)

  if !hasId {
    ctx.Status(http.StatusForbidden)
    return
  }

  bookmarks, err := service.Bookmarks(ctx, userId, req.Offset, req.Limit)
```

```
  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  d := dto.NewSpanDto(bookmarks, func(b domain.Bookmark) dto.BookmarkDto { return
dto.NewBookmarkDto(b) })

  ctx.JSON(http.StatusOK, d)
}

type GetBookmarkRequest struct {
  StoryId string `uri:"story_id" binding:"required,uuid"`
}

func GetBookmark(ctx *gin.Context) {
  var req GetBookmarkRequest
  if err := ctx.ShouldBindUri(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewBookmarksService(s)

  userId, hasId := h.UserId(ctx)

  if !hasId {
    ctx.Status(http.StatusForbidden)
    return
  }
```

```
  storyId, err := uuid.Parse(req.StoryId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  b, err := service.Bookmark(ctx, domain.NewBookmarkId(userId, storyId))

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  d := dto.NewBookmarkDto(b)

  ctx.JSON(http.StatusOK, d)
}

type CreateBookmarkRequest struct {
  StoryId string `uri:"story_id" binding:"required,uuid"`
}

func CreateBookmark(ctx *gin.Context) {
  var req CreateBookmarkRequest
  if err := ctx.ShouldBindUri(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }
```

```
  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewBookmarksService(s)

  storyId, err := uuid.Parse(req.StoryId)

  if err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  userId, hasId := h.UserId(ctx)

  if !hasId {
    ctx.Status(http.StatusForbidden)
    return
  }

  bookmark, err := service.Create(ctx, userId, storyId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  d := dto.NewBookmarkDto(bookmark)

  ctx.JSON(http.StatusOK, d)
}

type DeleteBookmarkRequest struct {
```

```
  StoryId string `uri:"story_id" binding:"required,uuid"`
}

func DeleteBookmark(ctx *gin.Context) {
  var req DeleteBookmarkRequest
  if err := ctx.ShouldBindUri(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewBookmarksService(s)

  storyId, err := uuid.Parse(req.StoryId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  userId, hasId := h.UserId(ctx)

  if !hasId {
    ctx.Status(http.StatusForbidden)
    return
  }

  err = service.Delete(ctx, domain.NewBookmarkId(userId, storyId))

  if err != nil {
```

```
    h.RenderError(ctx, err)
    return
  }

  ctx.Status(http.StatusNoContent)
}


type GetStoriesRequest struct {
  Offset domain.Offset `form:"offset" binding:"gte=0"`
  Limit  domain.Limit  `form:"limit" binding:"required,gte=5,lte=100"`
  Title  string        `form:"title"`
}

func GetStories(ctx *gin.Context) {
  var req GetStoriesRequest
  if err := ctx.ShouldBindQuery(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewStoriesService(s)

  stories, err := service.Stories(ctx, req.Title, req.Offset, req.Limit)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }
```

```
  d := dto.NewSpanDto(stories, func(s domain.Story) dto.StoryDto { return dto.NewStoryDto(s) })

  ctx.JSON(http.StatusOK, d)
}

type GetStoryRequest struct {
  Id string `uri:"story_id" binding:"required,uuid"`
}

func GetStory(ctx *gin.Context) {
  var req GetStoryRequest
  if err := ctx.ShouldBindUri(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewStoriesService(s)

  id, err := uuid.Parse(req.Id)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  story, err := service.Story(ctx, id)

  if err != nil {
```

```
    h.RenderError(ctx, err)
    return
  }

  d := dto.NewStoryDto(story)

  ctx.JSON(http.StatusOK, d)
}

type GetStoryPartRequest struct {
  StoryId string `uri:"story_id" binding:"required,uuid"`
  PartId  string `uri:"part_id" binding:"required,uuid"`
}

func GetStoryPart(ctx *gin.Context) {
  var req GetStoryPartRequest
  if err := ctx.ShouldBindUri(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewStoriesService(s)

  storyId, err := uuid.Parse(req.StoryId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }
```

```
  partId, err := uuid.Parse(req.PartId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  story, err := service.StoryAggregate(ctx, storyId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  part, hasPart := story.Part(partId)

  if !hasPart {
    h.NotFound(ctx, "part is not found")
    return
  }

  d := dto.NewPartDto(part)

  ctx.JSON(http.StatusOK, d)
}

type GetTrailsRequest struct {
  Offset domain.Offset `form:"offset" binding:"gte=0"`
  Limit  domain.Limit  `form:"limit" binding:"required,gte=5,lte=100"`
}
```

```
func GetTrails(ctx *gin.Context) {
  var req GetTrailsRequest
  if err := ctx.ShouldBindQuery(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewTrailsService(s)

  userId, hasId := h.UserId(ctx)

  if !hasId {
    ctx.Status(http.StatusForbidden)
    return
  }

  trails, err := service.Trails(ctx, userId, req.Offset, req.Limit)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  d := dto.NewSpanDto(trails, func(t domain.Trail) dto.TrailDto { return dto.NewTrailDto(t) })

  ctx.JSON(http.StatusOK, d)
}
```

```
type GetTrailRequest struct {
  TrailId string `uri:"story_id" binding:"required,uuid"`
}

func GetTrail(ctx *gin.Context) {
  var req GetTrailRequest
  if err := ctx.ShouldBindUri(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewTrailsService(s)

  userId, hasId := h.UserId(ctx)

  if !hasId {
    ctx.Status(http.StatusForbidden)
    return
  }

  trailId, err := uuid.Parse(req.TrailId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  t, err := service.Trail(ctx, domain.NewTrailId(userId, trailId))
```

```
  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  d := dto.NewTrailDto(t)

  ctx.JSON(http.StatusOK, d)
}

type CreateMarkRequestUri struct {
  StoryId string `uri:"story_id" binding:"required,uuid"`
}

type CreateMarkRequestBody struct {
  PartId string `json:"part_id"`
}

func CreateMark(ctx *gin.Context) {
  var req CreateMarkRequestUri
  if err := ctx.ShouldBindUri(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  var body CreateMarkRequestBody
  if err := ctx.ShouldBindBodyWith(&body, binding.JSON); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }
```

```
  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewTrailsService(s)

  storyId, err := uuid.Parse(req.StoryId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  userId, hasId := h.UserId(ctx)

  if !hasId {
    ctx.Status(http.StatusForbidden)
    return
  }

  partId, err := uuid.Parse(body.PartId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  trail, err := service.Move(ctx, domain.NewTrailId(userId, storyId), partId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }
```

```
  d := dto.NewTrailDto(trail)

  ctx.JSON(http.StatusOK, d)
}

type CreateTrailRequest struct {
  StoryId string `uri:"story_id" binding:"required,uuid"`
}

func CreateTrail(ctx *gin.Context) {
  var req CreateTrailRequest
  if err := ctx.ShouldBindUri(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewTrailsService(s)

  storyId, err := uuid.Parse(req.StoryId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  userId, hasId := h.UserId(ctx)

  if !hasId {
```

```
    ctx.Status(http.StatusForbidden)
    return
  }

  trail, err := service.Create(ctx, userId, storyId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  d := dto.NewTrailDto(trail)

  ctx.JSON(http.StatusOK, d)
}

type RestartTrailRequest struct {
  StoryId string `uri:"story_id" binding:"required,uuid"`
}

func RestartTrail(ctx *gin.Context) {
  var req RestartTrailRequest
  if err := ctx.ShouldBindUri(&req); err != nil {
    h.BadRequest(ctx, err.Error())
    return
  }

  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewTrailsService(s)
```

```
  storyId, err := uuid.Parse(req.StoryId)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  userId, hasId := h.UserId(ctx)

  if !hasId {
    ctx.Status(http.StatusForbidden)
    return
  }

  trail, err := service.Restart(ctx, domain.NewTrailId(userId, storyId))

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  d := dto.NewTrailDto(trail)

  ctx.JSON(http.StatusOK, d)
}

func CreateUser(ctx *gin.Context) {
  s := redisStorage.NewRedisStorage(redis.GetClient(),
storage.NewSqlStorage(postgres.GetConnection()))
  service := domain.NewUsersService(s)
```

```
  user, err := service.Create(ctx)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  token, err := generateToken(user)

  if err != nil {
    h.RenderError(ctx, err)
    return
  }

  d := dto.NewUserDto(user, token)

  ctx.JSON(http.StatusOK, d)
}

func generateToken(u domain.User) (string, error) {
  claims := jwt.MapClaims{
    "user_id": u.Id(),
  }
  token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
  return token.SignedString([]byte("super_secret_token_auth"))
}

func RegisterHandlers(engine *gin.Engine, c *config.Config) {
  engine.POST("/users", CreateUser)
  engine.GET("/stories/:story_id", GetStory)
  engine.GET("/stories", GetStories)
```

```
  engine.GET("/stories/:story_id/parts/:part_id", GetStoryPart)

  authorized := engine.Group("", http.Auth(c.AuthToken))

  authorized.POST("/stories/:story_id/trail/marks", CreateMark)
  authorized.POST("/stories/:story_id/trail", CreateTrail)
  authorized.POST("/stories/:story_id/trail/position/started", RestartTrail)
  authorized.GET("/stories/:story_id/trail", GetTrail)
  authorized.GET("/trails", GetTrails)

  authorized.POST("/stories/:story_id/bookmark", CreateBookmark)
  authorized.GET("/stories/:story_id/bookmark", GetBookmark)
  authorized.GET("/bookmarks", GetBookmarks)
  authorized.DELETE("/stories/:story_id/bookmark", DeleteBookmark)
}

type ErrorResponse struct {
  Message string `json:"message"`
}

func RenderError(ctx *gin.Context, err error) {
  resp := ErrorResponse{Message: "internal error"}
  status := http.StatusInternalServerError

  var ierr *derrors.Error
  if errors.As(err, &ierr) {
    resp.Message = err.Error()
    switch ierr.Code() {
    case derrors.ErrorCodeNotFound:
      status = http.StatusNotFound
    case derrors.ErrorCodeValidationFailed:
```

```
      status = http.StatusBadRequest
    default:
      status = http.StatusInternalServerError
    }
  }

  ctx.JSON(status, resp)
}

func BadRequest(ctx *gin.Context, msg string) {
  resp := ErrorResponse{Message: msg}
  ctx.JSON(http.StatusBadRequest, resp)
}

func NotFound(ctx *gin.Context, msg string) {
  resp := ErrorResponse{Message: msg}
  ctx.JSON(http.StatusNotFound, resp)
}

func Auth(token string) gin.HandlerFunc {
  return func(c *gin.Context) {
    header := c.GetHeader("Authorization")
    parts := strings.Split(header, " ")

    if len(parts) != 2 || parts[0] != "Bearer" {
      c.Status(http.StatusForbidden)
      return
    }

    claims := jwt.MapClaims{}
    t, err := jwt.ParseWithClaims(parts[1], claims, func(t *jwt.Token) (interface{}, error) {
```

```
        if _, ok := t.Method.(*jwt.SigningMethodHMAC); !ok {
            return nil, errors.New("unexpected signing method")
        }
        return []byte(token), nil
    })

    if err != nil {
        c.Status(http.StatusForbidden)
        return
    }

    if !t.Valid {
        c.Status(http.StatusForbidden)
        return
    }

    c.Set("user_id", claims["user_id"])
    c.Next()
  }
}

func UserId(c *gin.Context) (domain.Id, bool) {
  v, ok := c.Get("user_id")

  if !ok {
    return domain.Id{}, false
  }

  id, err := uuid.Parse(v.(string))

  if err != nil {
```

```
    return domain.Id{}, false
  }

  return id, true
}

type ErrorCode string

const (
  ErrorCodeNotFound         ErrorCode = "not found"
  ErrorCodeValidationFailed ErrorCode = "validation failed"
  ErrorCodeUnknown          ErrorCode = "unknown"
)

type Error struct {
  original error
  message  string
  code     ErrorCode
}

func (e Error) Error() string {
  if e.original != nil {
    return fmt.Sprintf("%s: %v", e.message, e.original)
  }

  return e.message
}

func (e Error) Unwrap() error {
  return e.original
}
```

```
func (e Error) Code() ErrorCode {
  return e.code
}

func NotFound(message string) error {
  return &Error{
    original: nil,
    message:  message,
    code:     ErrorCodeNotFound,
  }
}

func ValidationFailed(message string) error {
  return &Error{
    original: nil,
    message:  message,
    code:     ErrorCodeValidationFailed,
  }
}

func ValidationFailedf(format string, a ...interface{}) error {
  return &Error{
    original: nil,
    message:  fmt.Sprintf(format, a...),
    code:     ErrorCodeValidationFailed,
  }
}

func Unknown(message string) error {
  return &Error{
```

```
    original: nil,
    message:  message,
    code:     ErrorCodeUnknown,
  }
}

func IsNotFound(err error) bool {
  var e *Error
  return errors.As(err, &e) && e.Code() == ErrorCodeNotFound
}

type Bookmark struct {
  id BookmarkId
}

func (b Bookmark) Id() BookmarkId {
  return b.id
}

func NewBookmark(id BookmarkId) Bookmark {
  return Bookmark{
    id,
  }
}


type BookmarkId struct {
  userId  uuid.UUID
  storyId uuid.UUID
}
```

```
func (t BookmarkId) UserId() uuid.UUID {
  return t.userId
}

func (t BookmarkId) StoryId() uuid.UUID {
  return t.storyId
}

func (t BookmarkId) String() string {
  return t.storyId.String() + " " + t.userId.String()
}

func NewBookmarkId(userId, storyId uuid.UUID) BookmarkId {
  return BookmarkId{
    userId,
    storyId,
  }
}
type BookmarksService interface {
  Bookmark(context.Context, BookmarkId) (Bookmark, error)
  Bookmarks(context.Context, uuid.UUID, Offset, Limit) (Span[Bookmark], error)
  Create(ctx context.Context, userId uuid.UUID, storyId uuid.UUID) (Bookmark, error)
  Delete(context.Context, BookmarkId) error
}

type bookmarksService struct {
  s Storage
}

func (s bookmarksService) Bookmark(ctx context.Context, id BookmarkId) (Bookmark, error) {
  return s.s.Bookmark(ctx, id)
```

```
}

func (s bookmarksService) Bookmarks(ctx context.Context, userId uuid.UUID, offset Offset, limit
Limit) (Span[Bookmark], error) {
  return s.s.Bookmarks(ctx, userId, offset, limit)
}

func (s bookmarksService) Create(ctx context.Context, userId uuid.UUID, storyId uuid.UUID) (b
Bookmark, err error) {
  _, err = s.s.Story(ctx, storyId)

  if err != nil {
    return
  }

  b = NewBookmark(NewBookmarkId(userId, storyId))

  return b, s.s.CreateBookmark(ctx, b)
}

func (s bookmarksService) Delete(ctx context.Context, id BookmarkId) error {
  return s.s.DeleteBookmark(ctx, id)
}

func NewBookmarksService(s Storage) BookmarksService {
  return bookmarksService{
    s,
  }
}
type Id = uuid.UUID

type Offset int
```

```
type Limit int

type Span[T any] struct {
  value     []T
  moreExist bool
}

func (s Span[T]) Value() []T {
  return s.value
}

func (s Span[T]) MoreExist() bool {
  return s.moreExist
}

func NewSpan[T any](value []T, moreExist bool) Span[T] {
  return Span[T]{
    value,
    moreExist,
  }
}

type Labels = []string

func NewLabels(labels []string) Labels {
  keys := make(map[string]bool)
  list := make([]string, 0)
  for _, entry := range labels {
    if _, value := keys[entry]; !value {
      keys[entry] = true
```

```
        list = append(list, entry)
      }
    }
    return list
}

const (
    linkTitleMaxLength = 256
)

type Link struct {
    title       string
    targetPartId uuid.UUID
}

func (l Link) Title() string {
    return l.title
}

func (l Link) TargetPartId() uuid.UUID {
    return l.targetPartId
}

func NewLink(title string, targetPartId uuid.UUID) (Link, error) {
    if utf8.RuneCountInString(title) > linkTitleMaxLength {
        return Link{}, derrors.ValidationFailedf("StoryAggregate link title length should be less than
%d", linkTitleMaxLength)
    }
    return Link{title, targetPartId}, nil
}
```

```
const (
  minSequenceNumber = 1
)

type Mark struct {
  partId        uuid.UUID
  sequenceNumber int
}

func (m Mark) PartId() uuid.UUID {
  return m.partId
}

func (m Mark) SequenceNumber() int {
  return m.sequenceNumber
}

func NewMark(partId uuid.UUID, sequenceNumber int) (Mark, error) {
  if sequenceNumber < minSequenceNumber {
    return Mark{}, derrors.ValidationFailedf("mark sequence number should be greater than %d",
minSequenceNumber)
  }
  return Mark{partId, sequenceNumber}, nil
}

const (
  partTitleMaxLength = 256
)

type PartType string
```

```
const (
  BeginningPart PartType = "beginning"
  RegularPart           = "regular"
  EndPart               = "end"
)

type Part struct {
  id       uuid.UUID
  title    string
  content  string
  partType PartType
  links    []Link
}

func (p Part) Id() uuid.UUID {
  return p.id
}

func (p Part) Title() string {
  return p.title
}

func (p Part) Content() string {
  return p.content
}

func (p Part) PartType() PartType {
  return p.partType
}

func (p Part) Links() []Link {
```

```
  return p.links
}

func (p Part) HasLink(part Part) bool {
  for _, link := range p.links {
    if link.targetPartId == part.id {
      return true
    }
  }

  return false
}

func NewPart(id uuid.UUID, title, content string, partType PartType, links []Link) (Part, error) {
  if utf8.RuneCountInString(title) > partTitleMaxLength {
    return Part{}, derrors.ValidationFailedf("StoryAggregate part title length should be less than
%d", partTitleMaxLength)
  }
  return Part{id, title, content, partType, links}, nil
}

type Storage interface {
  Bookmark(context.Context, BookmarkId) (Bookmark, error)
  Bookmarks(ctx context.Context, userId uuid.UUID, o Offset, l Limit) (Span[Bookmark], error)
  CreateBookmark(context.Context, Bookmark) error
  DeleteBookmark(context.Context, BookmarkId) error
  StoryAggregate(ctx context.Context, id uuid.UUID) (StoryAggregate, error)
  Story(ctx context.Context, id uuid.UUID) (Story, error)
  Stories(ctx context.Context, title string, o Offset, l Limit) (Span[Story], error)
  Trail(ctx context.Context, id TrailId) (Trail, error)
  Trails(ctx context.Context, userId uuid.UUID, o Offset, l Limit) (Span[Trail], error)
```

```
  CreateTrail(context.Context, Trail) error
  UpdateTrail(context.Context, Trail) error
  DeleteTrail(context.Context, TrailId) error
  User(context.Context, uuid.UUID) (User, error)
  CreateUser(context.Context, User) error
}

type StoriesService interface {
  StoryAggregate(ctx context.Context, id uuid.UUID) (StoryAggregate, error)
  Story(ctx context.Context, id uuid.UUID) (Story, error)
  Stories(ctx context.Context, title string, o Offset, l Limit) (Span[Story], error)
}

type storiesService struct {
  s Storage
}

func (s storiesService) StoryAggregate(ctx context.Context, id uuid.UUID) (StoryAggregate, error) {
  return s.s.StoryAggregate(ctx, id)
}

func (s storiesService) Story(ctx context.Context, id uuid.UUID) (Story, error) {
  return s.s.Story(ctx, id)
}

func (s storiesService) Stories(ctx context.Context, title string, o Offset, l Limit) (Span[Story], error)
{
  return s.s.Stories(ctx, title, o, l)
}

func NewStoriesService(s Storage) StoriesService {
```

```
  return storiesService{
    s,
  }
}

type Story struct {
  id          uuid.UUID
  title       string
  description string
  createdAt   time.Time
  status      StoryStatus
  ownerId     uuid.UUID
  labels      Labels
}

func (s Story) Id() uuid.UUID {
  return s.id
}

func (s Story) Title() string {
  return s.title
}

func (s Story) Description() string {
  return s.description
}

func (s Story) CreatedAt() time.Time {
  return s.createdAt
}
```

```
func (s Story) Status() StoryStatus {
  return s.status
}

func (s Story) OwnerId() uuid.UUID {
  return s.ownerId
}

func (s Story) Labels() Labels {
  return s.labels
}

func NewStory(
  id uuid.UUID,
  title, description string,
  createdAt time.Time,
  status StoryStatus,
  ownerId uuid.UUID,
  labels Labels,
) (story Story, err error) {
  if utf8.RuneCountInString(title) > storyTitleMaxLength {
    err = derrors.ValidationFailedf("StoryAggregate title length should be less than %d",
storyTitleMaxLength)
    return
  }

  if utf8.RuneCountInString(description) > storyDescriptionMaxLength {
    err = derrors.ValidationFailedf("StoryAggregate description length should be less than %d",
storyDescriptionMaxLength)
    return
  }
```

```
  story = Story{
    id,
    title,
    description,
    createdAt,
    status,
    ownerId,
    labels,
  }

  return
}

const (
  storyTitleMaxLength       = 256
  storyDescriptionMaxLength = 2048
)

type StoryStatus string

const (
  StatusPublished = "published"
)

type StoryAggregate struct {
  Story
  parts []Part
}

func (s StoryAggregate) Status() StoryStatus {
  return s.status
```

```
}

func (s StoryAggregate) Id() uuid.UUID {
  return s.id
}

func (s StoryAggregate) Title() string {
  return s.title
}

func (s StoryAggregate) Part(id uuid.UUID) (Part, bool) {
  for _, part := range s.parts {
    if part.Id() == id {
      return part, true
    }
  }
  return Part{}, false
}

func (s StoryAggregate) BeginningPart() (Part, bool) {
  for _, part := range s.parts {
    if part.PartType() == BeginningPart {
      return part, true
    }
  }
  return Part{}, false
}

func NewStoryAggregate(
  id uuid.UUID,
  title, description string,
```

```
  createdAt time.Time,
  status StoryStatus,
  ownerId uuid.UUID,
  labels Labels,
  parts []Part,
) (aggregate StoryAggregate, err error) {
  story, err := NewStory(id, title, description, createdAt, status, ownerId, labels)

  if err != nil {
    return StoryAggregate{}, err
  }

  aggregate = StoryAggregate{
    story,
    parts,
  }

  return
}

type Trail struct {
  id    TrailId
  marks []Mark
}

func (t Trail) Id() TrailId {
  return t.id
}

func (t Trail) Marks() []Mark {
  return t.marks
```

```
}

func (t Trail) CurrentMark() (Mark, bool) {
  if len(t.marks) == 0 {
    return Mark{}, false
  }

  max := t.marks[0]

  for _, mark := range t.marks {
    if mark.sequenceNumber > max.sequenceNumber {
      max = mark
    }
  }

  return max, true
}

func NewTrail(id TrailId, marks []Mark) Trail {
  return Trail{
    id,
    marks,
  }
}
type TrailAggregate struct {
  trail Trail
  story StoryAggregate
}

func (t TrailAggregate) Trail() Trail {
  return t.trail
```

```
}

func (t *TrailAggregate) MoveTo(partId uuid.UUID) (err error) {
  target, hasPart := t.story.Part(partId)

  if !hasPart {
    return derrors.ValidationFailedf("story does not contain part with %s id", partId)
  }

  lastMark, hasMark := t.trail.CurrentMark()

  if !hasMark && target.PartType() != BeginningPart {
    return derrors.ValidationFailed("first mark should link to beginning part")
  }

  sequenceNumber := 1

  if hasMark {
    sequenceNumber = lastMark.sequenceNumber + 1
    lastPart, hasLastPart := t.story.Part(lastMark.PartId())

    if !hasLastPart {
      return derrors.ValidationFailedf("story does not contain part with %s id", lastMark.PartId())
    }

    if !lastPart.HasLink(target) {
      return derrors.ValidationFailedf("there is no link for %s source part and %s target part",
lastPart.Id(), target.Id())
    }
  }
```

```
  mark, err := NewMark(target.Id(), sequenceNumber)

  if err != nil {
    return
  }

  t.trail.marks = append(t.trail.marks, mark)

  return
}

func NewTrailAggregate(trail Trail, story StoryAggregate) TrailAggregate {
  return TrailAggregate{trail, story}
}

type TrailId struct {
  userId  uuid.UUID
  storyId uuid.UUID
}

func (t TrailId) UserId() uuid.UUID {
  return t.userId
}

func (t TrailId) StoryId() uuid.UUID {
  return t.storyId
}

func (t TrailId) String() string {
  return t.storyId.String() + " " + t.userId.String()
}
```

```
func NewTrailId(userId, storyId uuid.UUID) TrailId {
  return TrailId{
    userId,
    storyId,
  }
}

type TrailsService interface {
  Create(ctx context.Context, userId, storyId uuid.UUID) (Trail, error)
  Move(ctx context.Context, trailId TrailId, partId uuid.UUID) (Trail, error)
  Trails(ctx context.Context, userId uuid.UUID, o Offset, l Limit) (Span[Trail], error)
  Trail(context.Context, TrailId) (Trail, error)
  Restart(context.Context, TrailId) (Trail, error)
}

type trailsService struct {
  s Storage
}

func (t trailsService) Create(ctx context.Context, userId, storyId uuid.UUID) (trail Trail, err error) {
  id := NewTrailId(userId, storyId)

  _, err = t.s.Trail(ctx, id)

  if err == nil {
    return Trail{}, derrors.ValidationFailed("trail already exists")
  }

  if !derrors.IsNotFound(err) {
    return Trail{}, err
```

```
  }

  story, err := t.s.StoryAggregate(ctx, storyId)

  if err != nil {
    return
  }

  if story.Status() != StatusPublished {
    return Trail{}, derrors.ValidationFailed("story is not published. can't create a trail")
  }

  beginningPart, hasBeginningPart := story.BeginningPart()

  if !hasBeginningPart {
    return Trail{}, derrors.ValidationFailed("story has no beginning part")
  }

  aggregate := NewTrailAggregate(NewTrail(id, make([]Mark, 0)), story)
  err = aggregate.MoveTo(beginningPart.Id())

  if err != nil {
    return Trail{}, err
  }

  trail = aggregate.Trail()

  err = t.s.CreateTrail(ctx, trail)

  return
}
```

```
func (t trailsService) Move(ctx context.Context, trailId TrailId, partId uuid.UUID) (trail Trail, err error)
{
  trail, err = t.s.Trail(ctx, trailId)

  if err != nil {
    return
  }

  story, err := t.s.StoryAggregate(ctx, trailId.storyId)

  if err != nil {
    return
  }

  aggregate := NewTrailAggregate(trail, story)
  err = aggregate.MoveTo(partId)

  if err != nil {
    return
  }

  trail = aggregate.Trail()
  err = t.s.UpdateTrail(ctx, trail)

  return
}

func (t trailsService) Trails(ctx context.Context, userId uuid.UUID, o Offset, l Limit) (
  Span[Trail],
  error,
```

```
) {
  return t.s.Trails(ctx, userId, o, l)
}

func (t trailsService) Trail(ctx context.Context, trailId TrailId) (Trail, error) {
  return t.s.Trail(ctx, trailId)
}

func (t trailsService) Restart(ctx context.Context, id TrailId) (result Trail, err error) {
  err = t.s.DeleteTrail(ctx, id)

  if err != nil {
    return
  }

  result, err = t.Create(ctx, id.UserId(), id.StoryId())
  return
}

func NewTrailsService(s Storage) TrailsService {
  return trailsService{
    s,
  }
}

type User struct {
  id uuid.UUID
}

func (u User) Id() uuid.UUID {
  return u.id
```

```
}

func NewUser(id uuid.UUID) User {
  return User{
    id,
  }
}

type UsersService interface {
  User(context.Context, uuid.UUID) (User, error)
  Create(context.Context) (User, error)
}

type usersService struct {
  s Storage
}

func (t usersService) Create(ctx context.Context) (User, error) {
  user := NewUser(uuid.New())
  err := t.s.CreateUser(ctx, user)
  return user, err
}

func (t usersService) User(ctx context.Context, id uuid.UUID) (User, error) {
  return t.s.User(ctx, id)
}

func NewUsersService(s Storage) UsersService {
  return usersService{
    s,
  }
```

```
}

func main() {
	if err := app.Run(); err != nil {
		log.Fatal(err)
	}
}

module gitlab.com/aleks260920/storiks

go 1.18

require (
	github.com/dgrijalva/jwt-go v3.2.0+incompatible
	github.com/gin-gonic/gin v1.7.7
	github.com/google/uuid v1.3.0
	github.com/spf13/viper v1.11.0
)

require (
	github.com/cespare/xxhash/v2 v2.1.2 // indirect
	github.com/dgryski/go-rendezvous v0.0.0-20200823014737-9f7001d12a5f // indirect
	github.com/fsnotify/fsnotify v1.5.1 // indirect
	github.com/gin-contrib/sse v0.1.0 // indirect
	github.com/go-playground/locales v0.14.0 // indirect
	github.com/go-playground/universal-translator v0.18.0 // indirect
	github.com/go-playground/validator/v10 v10.11.0 // indirect
	github.com/go-redis/cache/v8 v8.4.3 // indirect
	github.com/go-redis/redis/v8 v8.11.3 // indirect
	github.com/golang/protobuf v1.5.2 // indirect
	github.com/hashicorp/hcl v1.0.0 // indirect
```

```
	github.com/json-iterator/go v1.1.12 // indirect
	github.com/klauspost/compress v1.13.6 // indirect
	github.com/leodido/go-urn v1.2.1 // indirect
	github.com/lib/pq v1.10.6 // indirect
	github.com/magiconair/properties v1.8.6 // indirect
	github.com/mattn/go-isatty v0.0.14 // indirect
	github.com/mitchellh/mapstructure v1.4.3 // indirect
	github.com/modern-go/concurrent v0.0.0-20180306012644-bacd9c7ef1dd // indirect
	github.com/modern-go/reflect2 v1.0.2 // indirect
	github.com/pelletier/go-toml v1.9.4 // indirect
	github.com/pelletier/go-toml/v2 v2.0.0-beta.8 // indirect
	github.com/spf13/afero v1.8.2 // indirect
	github.com/spf13/cast v1.4.1 // indirect
	github.com/spf13/jwalterweatherman v1.1.0 // indirect
	github.com/spf13/pflag v1.0.5 // indirect
	github.com/subosito/gotenv v1.2.0 // indirect
	github.com/ugorji/go/codec v1.2.7 // indirect
	github.com/vmihailenco/go-tinylfu v0.2.2 // indirect
	github.com/vmihailenco/msgpack/v5 v5.3.4 // indirect
	github.com/vmihailenco/tagparser/v2 v2.0.0 // indirect
	golang.org/x/crypto v0.0.0-20220513210258-46612604a0f9 // indirect
	golang.org/x/exp v0.0.0-20210916165020-5cb4fee858ee // indirect
	golang.org/x/sync v0.0.0-20210220032951-036812b2e83c // indirect
	golang.org/x/sys v0.0.0-20220513210249-45d2b4557a2a // indirect
	golang.org/x/text v0.3.7 // indirect
	google.golang.org/protobuf v1.28.0 // indirect
	gopkg.in/ini.v1 v1.66.4 // indirect
	gopkg.in/yaml.v2 v2.4.0 // indirect
	gopkg.in/yaml.v3 v3.0.0-20210107192922-496545a6307b // indirect
)
```

ДОДАТОК Б

Код Telegram бота

```
type Storage interface {
  User(TelegramId) (User, bool, error)
  CreateUser(User) error
  UpdateUser(User) error
}

type TelegramId = int64

type Position string

const (
  StartedPosition         Position = "Started"
  StoriesListPosition     Position = "StoriesList"
  StoryInfoPosition       Position = "StoryInfo"
  StoryInProgressPosition Position = "StoryInProgress"
  SearchPosition          Position = "Search"
  TrailsListPosition      Position = "TrailsList"
  BookmarksListPosition   Position = "BookmarksList"
)

var stateMachine = map[Position][]Position{
  StartedPosition: {
    StoriesListPosition,
    SearchPosition,
    TrailsListPosition,
    StartedPosition,
    BookmarksListPosition,
  },
  StoriesListPosition: {
```

```
    StoriesListPosition,
    StoryInfoPosition,
    StartedPosition,
  },
  BookmarksListPosition: {
    BookmarksListPosition,
    StartedPosition,
    StoryInfoPosition,
  },
  StoryInProgressPosition: {
    StoryInProgressPosition,
    StartedPosition,
  },
  StoryInfoPosition: {
    StoryInProgressPosition,
    StartedPosition,
    StoryInfoPosition,
  },
  SearchPosition: {
    StoriesListPosition,
  },
  TrailsListPosition: {
    TrailsListPosition,
    StartedPosition,
    StoryInfoPosition,
  },
}

type UserState map[string]interface{}

var (
```

```
  EmptyState UserState = make(map[string]interface{})
)

type User struct {
  telegramId    TelegramId
  storiksUserId uuid.UUID
  storiksToken  string
  position      Position
  state         UserState
}

func (u User) StoriksToken() string {
  return u.storiksToken
}

func (u User) State() UserState {
  return u.state
}

func (u User) Position() Position {
  return u.position
}

func (u User) StoriksUserId() uuid.UUID {
  return u.storiksUserId
}

func (u User) TelegramId() int64 {
  return u.telegramId
}
```

```
func (u *User) Move(position Position, state UserState) error {
  positions, ok := stateMachine[u.position]

  if !ok || len(positions) == 0 {
    return errors.New("there are no stateMachine for current position")
  }

  canMove := false
  for _, p := range positions {
    if p == position {
      canMove = true
      break
    }
  }

  if !canMove {
    return errors.New(fmt.Sprintf("can't move to %s position", position))
  }

  u.position = position
  u.state = state

  return nil
}

func NewStoriesListState(title string, page int) UserState {
  return map[string]interface{}{
    "page":  page,
    "title": title,
  }
}
```

```
func NewTrailsListState(page int) UserState {
  return map[string]interface{}{
    "page": page,
  }
}

func NewBookmarksListState(page int) UserState {
  return map[string]interface{}{
    "page": page,
  }
}

func NewStoryInProgressState(storyId, partId uuid.UUID) UserState {
  return map[string]interface{}{
    "story_id": storyId,
    "part_id":  partId,
  }
}

func NewStoryFinishedState() UserState {
  return EmptyState
}

func NewStartedState() UserState {
  return EmptyState
}

func NewStoryInfoState(storyId uuid.UUID) UserState {
  return map[string]interface{}{
    "story_id": storyId,
```

```
  }
}

func NewUser(telegramId TelegramId, storiksUserId uuid.UUID, storiksToken string, position
Position, state UserState) User {
  return User{
    telegramId,
    storiksUserId,
    storiksToken,
    position,
    state,
  }
}

var client *mongo.Client

func CreateMongoClient(uri string) (c *mongo.Client, err error) {
  client, err = mongo.Connect(context.Background(), options.Client().ApplyURI(uri))

  if err != nil {
    return
  }

  err = client.Ping(context.Background(), readpref.Primary())

  if err != nil {
    return
  }

  return client, err
}
```

```
type user struct {
  TelegramId    domain.TelegramId `bson:"telegramId"`
  StoriksUserId uuid.UUID        `bson:"storiksUserId"`
  StoriksToken  string           `bson:"storiksToken"`
  Position      domain.Position   `bson:"position"`
  State         domain.UserState  `bson:"state"`
}

func (u user) toDomain() domain.User {
  return domain.NewUser(u.TelegramId, u.StoriksUserId, u.StoriksToken, u.Position, u.State)
}

func newUser(u domain.User) user {
  return user{
    u.TelegramId(),
    u.StoriksUserId(),
    u.StoriksToken(),
    u.Position(),
    u.State(),
  }
}

type MongoStorage struct {
  client *mongo.Client
}

func (s MongoStorage) User(id domain.TelegramId) (u domain.User, ok bool, err error) {
  collection := s.client.Database("storiksBotDB").Collection("users")
  var result user
  err = collection.FindOne(context.Background(), bson.D{{"telegramId", id}}).Decode(&result)
```

```
  if err == mongo.ErrNoDocuments {
    err = nil
    return
  }

  if err != nil {
    return
  }

  ok = true
  u = result.toDomain()

  return
}

func (s MongoStorage) CreateUser(u domain.User) error {
  collection := s.client.Database("storiksBotDB").Collection("users")
  _, err := collection.InsertOne(context.Background(), newUser(u))
  return err
}

func (s MongoStorage) UpdateUser(u domain.User) error {
  collection := s.client.Database("storiksBotDB").Collection("users")
  document := bson.D{
    {"$set", newUser(u)},
  }
  _, err := collection.UpdateOne(context.Background(), bson.M{"telegramId": u.TelegramId()},
document)
  return err
}
```

```
func NewMongoStorage(c *mongo.Client) domain.Storage {
  return MongoStorage{
    c,
  }
}

type Span[T any] struct {
  Value    []T  `json:"value"`
  MoreExist bool `json:"more_exist"`
}

type User struct {
  Id    uuid.UUID `json:"id"`
  Token string    `json:"token"`
}

type Labels = []string

type Story struct {
  Id          uuid.UUID `json:"id"`
  Title       string    `json:"title"`
  Description string    `json:"description"`
  CreatedAt   time.Time `json:"created_at"`
  Status      string    `json:"status"`
  OwnerId     uuid.UUID `json:"owner_id"`
  Labels      Labels    `json:"labels"`
}

type PartType string
```

```
const (
  EndPart = "end"
)

type Part struct {
  Id      uuid.UUID `json:"id"`
  Title   string    `json:"title"`
  Content string    `json:"content"`
  Type    PartType  `json:"type"`
  Links   []Link    `json:"links"`
}

type Link struct {
  Title        string    `json:"title"`
  TargetPartId uuid.UUID `json:"target_part_id"`
}

type Trail struct {
  StoryId       uuid.UUID `json:"story_id"`
  CurrentPartId uuid.UUID `json:"current_part_id"`
}

type Bookmark struct {
  StoryId uuid.UUID `json:"story_id"`
}

type Adapter interface {
  Stories(title string, offset, limit int) (Span[Story], error)
  Story(uuid.UUID) (Story, error)
  Part(storyId, partId uuid.UUID) (Part, bool, error)
  CreateUser() (result User, err error)
```

```
  Trails(token string, offset, limit int) (Span[Trail], error)
  Trail(token string, id uuid.UUID) (Trail, bool, error)
  CreateTrail(token string, storyId uuid.UUID) (Trail, error)
  RestartTrail(token string, storyId uuid.UUID) (Trail, error)
  CreateMark(token string, storyId, partId uuid.UUID) (Trail, error)
  Bookmarks(token string, offset, limit int) (Span[Bookmark], error)
  Bookmark(token string, id uuid.UUID) (Bookmark, bool, error)
  CreateBookmark(token string, storyId uuid.UUID) (Bookmark, error)
  DeleteBookmark(token string, storyId uuid.UUID) error
}

type adapter struct {
  client *resty.Client
}

func NewStoriksAdapter(client *resty.Client, host string) Adapter {
  client.SetBaseURL(host)
  return adapter{
    client,
  }
}

func (a adapter) Stories(title string, offset, limit int) (result Span[Story], err error) {
  _, err = a.client.R().
    SetQueryParams(map[string]string{
      "offset": strconv.Itoa(offset),
      "limit":  strconv.Itoa(limit),
      "title":  title,
    }).
    SetHeader("Accept", "application/json").
    SetResult(&result).
```

```
    Get("/stories")
  return
}

func (a adapter) Story(id uuid.UUID) (result Story, err error) {
  _, err = a.client.R().
    SetHeader("Accept", "application/json").
    SetResult(&result).
    Get(fmt.Sprintf("/stories/%s", id))
  return
}

func (a adapter) CreateUser() (result User, err error) {
  client := resty.New()
  _, err = client.R().
    SetResult(&result).
    Post("/users")
  return
}

func (a adapter) CreateTrail(token string, storyId uuid.UUID) (result Trail, err error) {
  _, err = a.client.R().
    SetHeader("Authorization", fmt.Sprintf("Bearer %s", token)).
    SetResult(&result).
    Post(fmt.Sprintf("/stories/%s/trail", storyId))
  return
}

func (a adapter) Trail(token string, id uuid.UUID) (result Trail, found bool, err error) {
  response, err := a.client.R().
    SetHeader("Accept", "application/json").
```

```
    SetHeader("Authorization", fmt.Sprintf("Bearer %s", token)).
    SetResult(&result).
    Get(fmt.Sprintf("/stories/%s/trail", id))
  if err != nil || response.StatusCode() == http.StatusNotFound {
    return
  }
  return result, true, nil
}

func (a adapter) Trails(token string, offset, limit int) (result Span[Trail], err error) {
  _, err = a.client.R().
    SetQueryParams(map[string]string{
      "offset": strconv.Itoa(offset),
      "limit":  strconv.Itoa(limit),
    }).
    SetHeader("Accept", "application/json").
    SetHeader("Authorization", fmt.Sprintf("Bearer %s", token)).
    SetResult(&result).
    Get("/trails")
  return
}

func (a adapter) Part(storyId, partId uuid.UUID) (result Part, found bool, err error) {
  response, err := a.client.R().
    SetHeader("Accept", "application/json").
    SetResult(&result).
    Get(fmt.Sprintf("/stories/%s/parts/%s", storyId, partId))
  if err != nil || response.StatusCode() == http.StatusNotFound {
    return
  }
  return result, true, nil
```

```
}

func (a adapter) Bookmarks(token string, offset, limit int) (result Span[Bookmark], err error) {
  _, err = a.client.R().
    SetQueryParams(map[string]string{
      "offset": strconv.Itoa(offset),
      "limit":  strconv.Itoa(limit),
    }).
    SetHeader("Accept", "application/json").
    SetHeader("Authorization", fmt.Sprintf("Bearer %s", token)).
    SetResult(&result).
    Get("/bookmarks")
  return
}

func (a adapter) Bookmark(token string, id uuid.UUID) (result Bookmark, found bool, err error) {
  response, err := a.client.R().
    SetHeader("Accept", "application/json").
    SetHeader("Authorization", fmt.Sprintf("Bearer %s", token)).
    SetResult(&result).
    Get(fmt.Sprintf("/stories/%s/bookmark", id))
  if err != nil || response.StatusCode() == http.StatusNotFound {
    return
  }
  return result, true, nil
}

func (a adapter) CreateMark(token string, storyId, partId uuid.UUID) (result Trail, err error) {
  _, err = a.client.R().
    SetHeader("Content-Type", "application/json").
    SetHeader("Authorization", fmt.Sprintf("Bearer %s", token)).
```

```
    SetBody(map[string]interface{}{"part_id": partId.String()}).
    SetResult(&result).
    Post(fmt.Sprintf("/stories/%s/trail/marks", storyId))
  return
}

func (a adapter) RestartTrail(token string, storyId uuid.UUID) (result Trail, err error) {
  _, err = a.client.R().
    SetHeader("Authorization", fmt.Sprintf("Bearer %s", token)).
    SetResult(&result).
    Post(fmt.Sprintf("/stories/%s/trail/position/started", storyId))
  return
}

func (a adapter) CreateBookmark(token string, storyId uuid.UUID) (result Bookmark, err error) {
  _, err = a.client.R().
    SetHeader("Authorization", fmt.Sprintf("Bearer %s", token)).
    SetResult(&result).
    Post(fmt.Sprintf("/stories/%s/bookmark", storyId))
  return
}

func (a adapter) DeleteBookmark(token string, storyId uuid.UUID) error {
  _, err := a.client.R().
    SetHeader("Authorization", fmt.Sprintf("Bearer %s", token)).
    Delete(fmt.Sprintf("/stories/%s/bookmark", storyId))
  return err
}

type Config struct {
  offset        int
```

```
  timeout        int
  debug          bool
  mongoConnection string
  telegramToken   string
  storiksHost     string
}

func (c Config) StoriksHost() string {
  return c.storiksHost
}

func (c Config) MongoConnection() string {
  return c.mongoConnection
}

func (c Config) TelegramToken() string {
  return c.telegramToken
}

func (c Config) Debug() bool {
  return c.debug
}

func (c Config) Timeout() int {
  return c.timeout
}

func (c Config) Offset() int {
  return c.offset
}
```

```
var (
  config Config
  once   sync.Once
)

func Get() *Config {
  once.Do(func() {
    viper.SetConfigFile("./../../.env")

    if err := viper.ReadInConfig(); err != nil {
      return
    }

    config = Config{
      mongoConnection: viper.GetString("MONGO_CONNECTION"),
      telegramToken:   viper.GetString("TELEGRAM_TOKEN"),
      storiksHost:     viper.GetString("STORIKS_HOST"),
    }

    viper.AddConfigPath("./../../configs")
    viper.SetConfigName("main")
    viper.SetConfigType("yaml")

    if err := viper.ReadInConfig(); err != nil {
      config.timeout = viper.GetInt("BOT_TIMEOUT")
      config.offset = viper.GetInt("BOT_OFFSET")
      config.debug = viper.GetBool("DEBUG")
      return
    }
  })
```

```
  return &config
}

type AddBookmarkHandler struct {
  Bot     *tgbotapi.BotAPI
  Storage domain.Storage
  Storiks storiks.Adapter
}

func (h AddBookmarkHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
  }

  if !strings.HasPrefix(update.CallbackQuery.Data, "add_bookmark") {
    return false, nil
  }

  split := strings.Split(update.CallbackQuery.Data, " ")

  if len(split) != 2 {
    return false, errors.New("command is wrong")
  }

  storyId, err := uuid.Parse(split[1])

  if err != nil {
    return false, err
  }

  user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)
```

```
if err != nil {
  return false, err
}

if !userFound {
  return false, errors.New("user is not found")
}

_, err = h.Storiks.CreateBookmark(user.StoriksToken(), storyId)

if err != nil {
  return false, err
}

msg := tgbotapi.NewMessage(update.CallbackQuery.From.ID, "\u2B50 Bookmark is added")

if _, err = h.Bot.Send(msg); err != nil {
  return false, err
}

err = user.Move(domain.StoryInfoPosition, domain.NewStoryInfoState(storyId))

if err != nil {
  return false, err
}

story, err := h.Storiks.Story(storyId)

if err != nil {
  return false, err
```

```
  }

  _, trailFound, err := h.Storiks.Trail(user.StoriksToken(), storyId)

  if err != nil {
    return false, err
  }

  msg = templates.StoryInfo(update.CallbackQuery.Message.Chat.ID, story, trailFound, true)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}

type BackToMenuHandler struct {
  Bot     *tgbotapi.BotAPI
  Storage domain.Storage
}

func (h BackToMenuHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.Message == nil {
    return false, nil
  }
```

```
  if update.Message.Text != BackMenuItem.Text {
    return false, nil
  }

  user, userFound, err := h.Storage.User(update.Message.Chat.ID)

  if err != nil {
    return false, err
  }

  if !userFound {
    return false, errors.New("user is not found")
  }

  if err = user.Move(domain.StartedPosition, domain.EmptyState); err != nil {
    return false, err
  }

  msg := tgbotapi.NewMessage(user.TelegramId(), "You are in the main menu!")
  msg.ReplyMarkup = MenuKeyboard

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}
```

```
const (
  DefaultPageSize = 5
)

var (
  StoriesMenuItem     = tgbotapi.NewKeyboardButton("\U0001F4D6 Stories")
  TrailsMenuItem      = tgbotapi.NewKeyboardButton("\u2728 My Reading")
  InformationMenuItem = tgbotapi.NewKeyboardButton("\u2139 Information")
  BookmarksMenuItem   = tgbotapi.NewKeyboardButton("\u2B50 Bookmarks")
  SearchMenuItem      = tgbotapi.NewKeyboardButton("\U0001F50D Search")
  BackMenuItem        = tgbotapi.NewKeyboardButton("\u2934 Back")
)

var MenuKeyboard = tgbotapi.NewReplyKeyboard(
  tgbotapi.NewKeyboardButtonRow(
    StoriesMenuItem,
    TrailsMenuItem,
  ),
  tgbotapi.NewKeyboardButtonRow(
    InformationMenuItem,
    SearchMenuItem,
  ),
  tgbotapi.NewKeyboardButtonRow(
    BookmarksMenuItem,
  ),
)

var BackKeyboard = tgbotapi.NewReplyKeyboard(
  tgbotapi.NewKeyboardButtonRow(
    BackMenuItem,
```

```
  ),
)

func FirstPage() int {
  return 0
}

func Offset(pageNumber, pageSize int) int {
  return pageNumber * pageSize
}

type ContinueTrailHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h ContinueTrailHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
  }

  if !strings.HasPrefix(update.CallbackQuery.Data, "continue_trail") {
    return false, nil
  }

  split := strings.Split(update.CallbackQuery.Data, " ")

  if len(split) != 2 {
    return false, errors.New("command is wrong")
  }
```

```
  trailId, err := uuid.Parse(split[1])

  user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

  if err != nil {
    return false, err
  }

  if !userFound {
    return false, errors.New("user is not found")
  }

  trail, trailFound, err := h.Storiks.Trail(user.StoriksToken(), trailId)

  if err != nil {
    return false, err
  }

  if !trailFound {
    return false, errors.New("trail is not found")
  }

  if err = user.Move(domain.StoryInProgressPosition,
domain.NewStoryInProgressState(trail.StoryId, trail.CurrentPartId)); err != nil {
    return false, err
  }

  nextPart, partFound, err := h.Storiks.Part(trail.StoryId, trail.CurrentPartId)

  if err != nil {
```

```
    return false, err
  }

  if !partFound {
    return false, errors.New("part is not found")
  }

  msg := templates.StoryPart(update.CallbackQuery.Message.Chat.ID, nextPart)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, nil
  }

  return true, nil
}

type DoSearchHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h DoSearchHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.Message == nil {
    return false, nil
  }
```

```
user, userFound, err := h.Storage.User(update.Message.Chat.ID)

if err != nil {
  return false, err
}

if !userFound {
  return false, errors.New("user is not found")
}

if user.Position() != domain.SearchPosition {
  return false, nil
}

storyTitle := update.Message.Text
page := 0

stories, err := h.Storiks.Stories(storyTitle, 0, DefaultPageSize)

if err != nil {
  return false, err
}

if len(stories.Value) == 0 {
  if err = user.Move(domain.StartedPosition, domain.EmptyState); err != nil {
    return false, err
  }

  msg := tgbotapi.NewMessage(update.Message.Chat.ID, "There are no stories with such title!")

  if _, err = h.Bot.Send(msg); err != nil {
```

```
        return false, err
      }

      return true, nil
    }

    if err = user.Move(domain.StoriesListPosition, domain.NewStoriesListState(storyTitle, page));
err != nil {
      return false, err
    }

    msg := templates.StoriesList(user.TelegramId(), stories.Value, page, !stories.MoreExist)

    if _, err = h.Bot.Send(msg); err != nil {
      return false, err
    }

    if err = h.Storage.UpdateUser(user); err != nil {
      return false, err
    }

    return true, nil
}

type FinishStoryHandler struct {
    Bot     *tgbotapi.BotAPI
    Storiks storiks.Adapter
    Storage domain.Storage
}

func (h FinishStoryHandler) Handle(update tgbotapi.Update) (bool, error) {
```

```
if update.CallbackQuery == nil {
  return false, nil
}

if !strings.HasPrefix(update.CallbackQuery.Data, "finish_story") {
  return false, nil
}

user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

if err != nil {
  return false, err
}

if !userFound {
  return false, errors.New("user is not found")
}

storyId := user.State()["story_id"].(primitive.Binary)

sid, err := uuid.FromBytes(storyId.Data)

if err != nil {
  return false, err
}

err = user.Move(domain.StartedPosition, domain.NewStoryFinishedState())

if err != nil {
  return false, err
}
```

```
  story, err := h.Storiks.Story(sid)

  if err != nil {
    return false, err
  }

  msg := tgbotapi.NewMessage(user.TelegramId(), fmt.Sprintf("Congratulations! %s is finished!",
story.Title))
  msg.ReplyMarkup = MenuKeyboard

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}

type InformationHandler struct {
  Bot *tgbotapi.BotAPI
}

func (h InformationHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.Message == nil || update.Message.Text != InformationMenuItem.Text {
    return false, nil
  }
```

```
  msg := tgbotapi.NewMessage(update.Message.Chat.ID, "This is a storiks bot! Here you can read different stories and relax. Happy reading!")

  if _, err := h.Bot.Send(msg); err != nil {
    return false, err
  }

  return true, nil
}

type ListBookmarksHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h ListBookmarksHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.Message == nil || update.Message.Text != BookmarksMenuItem.Text {
    return false, nil
  }

  user, userFound, err := h.Storage.User(update.Message.Chat.ID)

  if err != nil {
    return false, err
  }

  if !userFound {
    return false, errors.New("user is not found")
  }
```

```
  page := FirstPage()

  if err = user.Move(domain.BookmarksListPosition, domain.NewBookmarksListState(page));
err != nil {
    return false, err
  }

  msg := tgbotapi.NewMessage(user.TelegramId(), BookmarksMenuItem.Text)
  msg.ReplyMarkup = BackKeyboard

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  bookmarks, err := h.Storiks.Bookmarks(user.StoriksToken(), 0, DefaultPageSize)

  if err != nil {
    return false, err
  }

  if len(bookmarks.Value) == 0 {
    msg = tgbotapi.NewMessage(update.Message.Chat.ID, "There are no bookmarks!")

    if _, err = h.Bot.Send(msg); err != nil {
      return false, err
    }

    return true, nil
  }

  t := make([]storiks.Story, 0)
```

```
  for _, bookmark := range bookmarks.Value {
    s, err := h.Storiks.Story(bookmark.StoryId)
    if err != nil {
      return false, err
    }
    t = append(t, s)
  }

  msg = templates.StoriesList(user.TelegramId(), t, page, !bookmarks.MoreExist)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}

type ListBookmarksTurnPageHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h ListBookmarksTurnPageHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
  }
```

```
  if update.CallbackQuery.Data != "bookmarks_next_page" && update.CallbackQuery.Data !=
"bookmarks_previous_page" {
    return false, nil
  }

  user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

  if err != nil {
    return false, err
  }

  if !userFound {
    return false, errors.New("user is not found")
  }

  page := user.State()["page"].(int)

  if update.CallbackQuery.Data == "bookmarks_next_page" {
    page = page + 1
  }

  if update.CallbackQuery.Data == "bookmarks_previous_page" {
    page = page - 1
  }

  if err = user.Move(domain.BookmarksListPosition, domain.NewBookmarksListState(page));
err != nil {
    return false, err
  }

  msg := tgbotapi.NewMessage(user.TelegramId(), BookmarksMenuItem.Text)
```

```
  msg.ReplyMarkup = BackKeyboard

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  bookmarks, err := h.Storiks.Bookmarks(user.StoriksToken(), Offset(page, DefaultPageSize),
DefaultPageSize)

  if err != nil {
    return false, err
  }

  stories := make([]storiks.Story, 0)

  for _, b := range bookmarks.Value {
    s, err := h.Storiks.Story(b.StoryId)

    if err != nil {
      return false, err
    }

    stories = append(stories, s)
  }

  msg = templates.StoriesList(user.TelegramId(), stories, page, !bookmarks.MoreExist)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }
```

```
  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}

type ListStoriesHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h ListStoriesHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.Message == nil || update.Message.Text != StoriesMenuItem.Text {
    return false, nil
  }

  user, userFound, err := h.Storage.User(update.Message.Chat.ID)

  if err != nil {
    return false, err
  }

  if !userFound {
    return false, errors.New("user is not found")
  }

  title := ""
  page := FirstPage()
```

```
  err = user.Move(domain.StoriesListPosition, domain.NewStoriesListState(title, page))

  if err != nil {
    return false, err
  }

  msg := tgbotapi.NewMessage(user.TelegramId(), StoriesMenuItem.Text)
  msg.ReplyMarkup = BackKeyboard

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  stories, err := h.Storiks.Stories(title, Offset(page, DefaultPageSize), DefaultPageSize)

  if err != nil {
    return false, err
  }

  msg = templates.StoriesList(user.TelegramId(), stories.Value, page, !stories.MoreExist)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}
```

```
type ListStoriesTurnPageHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h ListStoriesTurnPageHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
  }

  if update.CallbackQuery.Data != "stories_next_page" && update.CallbackQuery.Data !=
"stories_previous_page" {
    return false, nil
  }

  user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

  if err != nil {
    return false, err
  }

  if !userFound {
    return false, errors.New("user is not found")
  }

  page := user.State()["page"].(int32)
  title := user.State()["title"].(string)

  if update.CallbackQuery.Data == "stories_next_page" {
```

```
    page = page + 1
  }

  if update.CallbackQuery.Data == "stories_previous_page" {
    page = page - 1
  }

  if err = user.Move(domain.StoriesListPosition, domain.NewStoriesListState(title, int(page))); err !=
nil {
    return false, err
  }

  msg := tgbotapi.NewMessage(user.TelegramId(), StoriesMenuItem.Text)
  msg.ReplyMarkup = BackKeyboard

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  stories, err := h.Storiks.Stories(title, Offset(int(page), DefaultPageSize), DefaultPageSize)

  if err != nil {
    return false, err
  }

  msg = templates.StoriesList(user.TelegramId(), stories.Value, int(page), !stories.MoreExist)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }
```

```
  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}

type ListTrailsHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h ListTrailsHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.Message == nil || update.Message.Text != TrailsMenuItem.Text {
    return false, nil
  }

  user, userFound, err := h.Storage.User(update.Message.Chat.ID)

  if err != nil {
    return false, err
  }

  if !userFound {
    return false, errors.New("user is not found")
  }

  if err = user.Move(domain.TrailsListPosition, domain.NewTrailsListState(0)); err != nil {
    return false, err
  }
```

```
msg := tgbotapi.NewMessage(user.TelegramId(), TrailsMenuItem.Text)
msg.ReplyMarkup = BackKeyboard

if _, err = h.Bot.Send(msg); err != nil {
  return false, err
}

trails, err := h.Storiks.Trails(user.StoriksToken(), 0, DefaultPageSize)

if err != nil {
  return false, err
}

if len(trails.Value) == 0 {
  msg = tgbotapi.NewMessage(update.Message.Chat.ID, "There are no trails!")

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  return true, nil
}

t := make([]storiks.Story, 0)
for _, trail := range trails.Value {
  s, err := h.Storiks.Story(trail.StoryId)
  if err != nil {
    return false, err
  }
  t = append(t, s)
```

```
  }

  msg = templates.StoriesList(user.TelegramId(), t, 0, !trails.MoreExist)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}

type ListTrailsTurnPageHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h ListTrailsTurnPageHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
  }

  if update.CallbackQuery.Data != "trails_next_page" && update.CallbackQuery.Data !=
"trails_previous_page" {
    return false, nil
  }
```

```
user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

if err != nil {
  return false, err
}

if !userFound {
  return false, errors.New("user is not found")
}

page := user.State()["page"].(int)

if update.CallbackQuery.Data == "trails_next_page" {
  page = page + 1
}

if update.CallbackQuery.Data == "trails_previous_page" {
  page = page - 1
}

if err = user.Move(domain.TrailsListPosition, domain.NewTrailsListState(page)); err != nil {
  return false, err
}

msg := tgbotapi.NewMessage(user.TelegramId(), TrailsMenuItem.Text)
msg.ReplyMarkup = BackKeyboard

if _, err = h.Bot.Send(msg); err != nil {
  return false, err
}
```

```
  trails, err := h.Storiks.Trails(user.StoriksToken(), page*DefaultPageSize, DefaultPageSize)

  if err != nil {
    return false, err
  }

  stories := make([]storiks.Story, 0)
  for _, t := range trails.Value {
    s, err := h.Storiks.Story(t.StoryId)

    if err != nil {
      return false, err
    }

    stories = append(stories, s)
  }

  msg = templates.StoriesList(user.TelegramId(), stories, page, !trails.MoreExist)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}

type MoveToNextPartHandler struct {
```

```
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h MoveToNextPartHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
  }

  if !strings.HasPrefix(update.CallbackQuery.Data, "next_part") {
    return false, nil
  }

  split := strings.Split(update.CallbackQuery.Data, " ")

  if len(split) != 2 {
    return false, errors.New("command is wrong")
  }

  nextPartId, err := uuid.Parse(split[1])

  if err != nil {
    return false, err
  }

  user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

  if err != nil {
    return false, err
  }
```

```
  if !userFound {
    return false, errors.New("user is not found")
  }

  storyId := user.State()["story_id"].(primitive.Binary)

  sid, err := uuid.FromBytes(storyId.Data)

  if err != nil {
    return false, err
  }

  err = user.Move(domain.StoryInProgressPosition, domain.NewStoryInProgressState(sid,
nextPartId))

  if err != nil {
    return false, err
  }

  nextPart, partFound, err := h.Storiks.Part(sid, nextPartId)

  if err != nil {
    return false, err
  }

  if !partFound {
    return false, errors.New("part is not found")
  }

  _, err = h.Storiks.CreateMark(user.StoriksToken(), sid, nextPartId)
```

```
  if err != nil {
    return false, err
  }

  msg := templates.StoryPart(update.CallbackQuery.Message.Chat.ID, nextPart)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, nil
  }

  return true, nil
}

type RemoveBookmarkHandler struct {
  Bot     *tgbotapi.BotAPI
  Storage domain.Storage
  Storiks storiks.Adapter
}

func (h RemoveBookmarkHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
  }

  if !strings.HasPrefix(update.CallbackQuery.Data, "remove_bookmark") {
    return false, nil
```

```
}

split := strings.Split(update.CallbackQuery.Data, " ")

if len(split) != 2 {
  return false, errors.New("command is wrong")
}

storyId, err := uuid.Parse(split[1])

if err != nil {
  return false, err
}

user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

if err != nil {
  return false, err
}

if !userFound {
  return false, errors.New("user is not found")
}

err = h.Storiks.DeleteBookmark(user.StoriksToken(), storyId)

if err != nil {
  return false, err
}

msg := tgbotapi.NewMessage(update.CallbackQuery.From.ID, "\u274C Bookmark is deleted")
```

```
if _, err = h.Bot.Send(msg); err != nil {
  return false, err
}

err = user.Move(domain.StoryInfoPosition, domain.NewStoryInfoState(storyId))

if err != nil {
  return false, err
}

story, err := h.Storiks.Story(storyId)

if err != nil {
  return false, err
}

_, trailFound, err := h.Storiks.Trail(user.StoriksToken(), storyId)

if err != nil {
  return false, err
}

msg = templates.StoryInfo(update.CallbackQuery.Message.Chat.ID, story, trailFound, false)

if _, err = h.Bot.Send(msg); err != nil {
  return false, err
}

if err = h.Storage.UpdateUser(user); err != nil {
  return false, err
```

```
  }

  return true, nil
}

type RestartTrailHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h RestartTrailHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
  }

  if !strings.HasPrefix(update.CallbackQuery.Data, "restart_trail") {
    return false, nil
  }

  split := strings.Split(update.CallbackQuery.Data, " ")

  if len(split) != 2 {
    return false, errors.New("command is wrong")
  }

  trailId, err := uuid.Parse(split[1])

  user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

  if err != nil {
```

```
    return false, err
  }

  if !userFound {
    return false, errors.New("user is not found")
  }

  trail, trailFound, err := h.Storiks.Trail(user.StoriksToken(), trailId)

  if err != nil {
    return false, err
  }

  if !trailFound {
    return false, errors.New("trail is not found")
  }

  trail, err = h.Storiks.RestartTrail(user.StoriksToken(), trail.StoryId)

  if err != nil {
    return false, err
  }

  if err = user.Move(domain.StoryInProgressPosition,
domain.NewStoryInProgressState(trail.StoryId, trail.CurrentPartId)); err != nil {
    return false, err
  }

  nextPart, partFound, err := h.Storiks.Part(trail.StoryId, trail.CurrentPartId)

  if err != nil {
```

```
    return false, err
  }

  if !partFound {
    return false, errors.New("part is not found")
  }

  msg := templates.StoryPart(update.CallbackQuery.Message.Chat.ID, nextPart)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, nil
  }

  return true, nil
}

type SearchHandler struct {
  Bot     *tgbotapi.BotAPI
  Storage domain.Storage
}

func (h SearchHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.Message == nil || update.Message.Text != SearchMenuItem.Text {
    return false, nil
  }
```

```
  msg := tgbotapi.NewMessage(update.Message.Chat.ID, "\U0001F50D What story are you
looking for?\nSearch: by title")

  if _, err := h.Bot.Send(msg); err != nil {
    return false, err
  }

  user, userFound, err := h.Storage.User(update.Message.Chat.ID)

  if err != nil {
    return false, err
  }

  if !userFound {
    return false, errors.New("user is not found")
  }

  if err = user.Move(domain.SearchPosition, domain.EmptyState); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}
type StartCommandHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
```

```
}

func (h StartCommandHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.Message == nil {
    return false, nil
  }

  if !update.Message.IsCommand() && update.Message.Command() != "start" {
    return false, nil
  }

  user, userFound, err := h.Storage.User(update.Message.From.ID)

  if err != nil {
    return false, err
  }

  if userFound {
    err = user.Move(domain.StartedPosition, domain.NewStartedState())

    if err != nil {
      return false, err
    }

    if err = h.Storage.UpdateUser(user); err != nil {
      return false, err
    }

  } else {
    storiksUser, err := h.Storiks.CreateUser()
```

```
    if err != nil {
      return false, err
    }

    user = domain.NewUser(update.Message.From.ID, storiksUser.Id, storiksUser.Token,
domain.StartedPosition, domain.NewStartedState())

    if err = h.Storage.CreateUser(user); err != nil {
      return false, err
    }
  }

  msg := tgbotapi.NewMessage(update.Message.Chat.ID, "Welcome!")
  msg.ReplyMarkup = MenuKeyboard

  if _, err := h.Bot.Send(msg); err != nil {
    return false, err
  }

  return true, nil
}

type StartStoryHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h StartStoryHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
```

```
}

if !strings.HasPrefix(update.CallbackQuery.Data, "start_trail") {
  return false, nil
}

split := strings.Split(update.CallbackQuery.Data, " ")

if len(split) != 2 {
  return false, errors.New("command is wrong")
}

storyId, err := uuid.Parse(split[1])

if err != nil {
  return false, err
}

user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

if err != nil {
  return false, err
}

if !userFound {
  return false, errors.New("user is not found")
}

_, trailExist, err := h.Storiks.Trail(user.StoriksToken(), storyId)

if err != nil {
```

```
    return false, err
  }

  if trailExist {
    return false, errors.New("trail is already exist")
  }

  trail, err := h.Storiks.CreateTrail(user.StoriksToken(), storyId)

  if err != nil {
    return false, err
  }

  part, partFound, err := h.Storiks.Part(trail.StoryId, trail.CurrentPartId)

  if err != nil {
    return false, err
  }

  if !partFound {
    return false, errors.New("part is not found")
  }

  err = user.Move(domain.StoryInProgressPosition, domain.NewStoryInProgressState(storyId,
part.Id))

  if err != nil {
    return false, err
  }

  msg := templates.StoryPart(update.CallbackQuery.Message.Chat.ID, part)
```

```
  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}

type StoryInfoHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}

func (h StoryInfoHandler) Handle(update tgbotapi.Update) (bool, error) {
  if update.CallbackQuery == nil {
    return false, nil
  }

  if !strings.HasPrefix(update.CallbackQuery.Data, "story_info") {
    return false, nil
  }

  split := strings.Split(update.CallbackQuery.Data, " ")

  if len(split) != 2 {
    return false, errors.New("command is wrong")
```

```
}

storyId, err := uuid.Parse(split[1])

if err != nil {
  return false, err
}

user, userFound, err := h.Storage.User(update.CallbackQuery.From.ID)

if err != nil {
  return false, err
}

if !userFound {
  return false, errors.New("user is not found")
}

err = user.Move(domain.StoryInfoPosition, domain.NewStoryInfoState(storyId))

if err != nil {
  return false, err
}

story, err := h.Storiks.Story(storyId)

if err != nil {
  return false, err
}

_, trailFound, err := h.Storiks.Trail(user.StoriksToken(), storyId)
```

```
  if err != nil {
    return false, err
  }

  _, bookmarkFound, err := h.Storiks.Bookmark(user.StoriksToken(), storyId)

  if err != nil {
    return false, err
  }

  msg := templates.StoryInfo(update.CallbackQuery.Message.Chat.ID, story, trailFound,
bookmarkFound)

  if _, err = h.Bot.Send(msg); err != nil {
    return false, err
  }

  if err = h.Storage.UpdateUser(user); err != nil {
    return false, err
  }

  return true, nil
}

type UnknownActionHandler struct {
  Bot     *tgbotapi.BotAPI
  Storiks storiks.Adapter
  Storage domain.Storage
}
```

```
func (h UnknownActionHandler) Handle(update tgbotapi.Update) (bool, error) {
  msg := tgbotapi.NewMessage(update.Message.Chat.ID, "Unknown command!")

  if _, err := h.Bot.Send(msg); err != nil {
    return false, err
  }

  return true, nil
}

type UpdateHandler interface {
  Handle(tgbotapi.Update) (bool, error)
}

func List[T any](
  id domain.TelegramId,
  items []T,
  itemContent func(item T, index, pageNumber, pageSize int) string,
  itemBtnData func(T) string,
  pageNumber int,
  lastPage bool,
  nextPageBtnData, previousPageBtnData string,
) tgbotapi.MessageConfig {
  var (
    sb          = strings.Builder{}
    pageSize    = len(items)
    itemsButtons = make([]tgbotapi.InlineKeyboardButton, 0, pageSize)
    controls    = make([]tgbotapi.InlineKeyboardButton, 0, 2)
  )

  for i, item := range items {
```

```
    itemsButtons = append(itemsButtons,
tgbotapi.NewInlineKeyboardButtonData(strconv.Itoa((pageNumber*pageSize)+i+1),
itemBtnData(item))) // fmt.Sprintf("story_info %s", story.Id)

    sb.WriteString(itemContent(item, i, pageNumber, pageSize))


    if i != len(items)-1 {
      sb.WriteString("\n\n")
    }
  }


  if pageNumber != 0 {
    controls = append(controls, tgbotapi.NewInlineKeyboardButtonData("\u23EE Previous",
previousPageBtnData))
  }
  controls = append(controls, tgbotapi.NewInlineKeyboardButtonData(fmt.Sprintf("Page: %d",
pageNumber+1), "page_number"))
  if !lastPage {
    controls = append(controls, tgbotapi.NewInlineKeyboardButtonData("\u23ED Next",
nextPageBtnData))
  }


  var numericKeyboard = tgbotapi.NewInlineKeyboardMarkup(
    tgbotapi.NewInlineKeyboardRow(
      itemsButtons...,
    ),
    tgbotapi.NewInlineKeyboardRow(
      controls...,
    ),
  )


  msg := tgbotapi.NewMessage(id, sb.String())
  msg.ParseMode = "MarkDown"
  msg.ReplyMarkup = numericKeyboard
```

```
  return msg
}

func StoriesList(id domain.TelegramId, stories []storiks.Story, pageNumber int, lastPage bool)
tgbotapi.MessageConfig {
  itemContent := func(story storiks.Story, index, pageNumber, pageSize int) string {
    sb := strings.Builder{}
    sb.WriteString(strconv.Itoa((pageNumber * pageSize) + index + 1))
    sb.WriteRune(')')
    sb.WriteRune(' ')
    sb.WriteString(story.Title)
    sb.WriteString("\n")
    sb.WriteRune('`')
    sb.WriteString("Created: ")
    sb.WriteString(story.CreatedAt.Format("2006-02-01"))

    if len(story.Labels) > 0 {
      for _, label := range story.Labels {
        sb.WriteRune(' ')
        sb.WriteRune('|')
        sb.WriteRune(' ')
        sb.WriteString(label)
      }
    }

    sb.WriteRune('`')
    return sb.String()
  }

  itemBtnData := func(story storiks.Story) string {
```

```
    return fmt.Sprintf("story_info %s", story.Id)
  }

  return List[storiks.Story](id, stories, itemContent, itemBtnData, pageNumber, lastPage,
"stories_next_page", "stories_previous_page")
}

func StoryInfo(telegramId int64, story storiks.Story, started, bookmarked bool)
tgbotapi.MessageConfig {
  sb := strings.Builder{}

  sb.WriteString("<b>")
  sb.WriteString(story.Title)
  sb.WriteString("</b>")
  sb.WriteString("\n\n")
  sb.WriteString("<b>Created:</b> ")
  sb.WriteString(story.CreatedAt.Format("2006-02-01"))
  sb.WriteString("\n\n")
  sb.WriteString("<b>Description:</b> ")
  sb.WriteString("\n")
  sb.WriteString(story.Description)

  msg := tgbotapi.NewMessage(telegramId, sb.String())

  buttons := make([]tgbotapi.InlineKeyboardButton, 0)

  if started {
    buttons = append(buttons, tgbotapi.NewInlineKeyboardButtonData("\u25C0 Continue",
fmt.Sprintf("continue_trail %s", story.Id)))
    buttons = append(buttons, tgbotapi.NewInlineKeyboardButtonData("\U0001F501 Restart",
fmt.Sprintf("restart_trail %s", story.Id)))
  } else {
```

```
    buttons = append(buttons, tgbotapi.NewInlineKeyboardButtonData("\U0001F440 Start",
fmt.Sprintf("start_trail %s", story.Id)))
  }

  if bookmarked {
    buttons = append(buttons, tgbotapi.NewInlineKeyboardButtonData("\u274C Remove
Bookmark", fmt.Sprintf("remove_bookmark %s", story.Id)))
  } else {
    buttons = append(buttons, tgbotapi.NewInlineKeyboardButtonData("\u2B50 To Bookmarks",
fmt.Sprintf("add_bookmark %s", story.Id)))
  }

  var keyboard = tgbotapi.NewInlineKeyboardMarkup(
    buttons,
  )

  msg.ReplyMarkup = keyboard
  msg.ParseMode = "HTML"

  return msg
}

func StoryPart(telegramId int64, part storiks.Part) tgbotapi.MessageConfig {
  sb := strings.Builder{}
  sb.WriteString("<b>")
  sb.WriteString(part.Title)
  sb.WriteString("</b>")
  sb.WriteString("\n\n")
  sb.WriteString(part.Content)

  msg := tgbotapi.NewMessage(telegramId, sb.String())
```

```
  if part.Type == storiks.EndPart {
    var keyboard = tgbotapi.NewInlineKeyboardMarkup(
      tgbotapi.NewInlineKeyboardRow(
        tgbotapi.NewInlineKeyboardButtonData("\U0001F3C1 Finish", "finish_story"),
      ),
    )

    msg.ReplyMarkup = keyboard
  } else {
    buttons := make([][]tgbotapi.InlineKeyboardButton, 0, len(part.Links))
    for _, link := range part.Links {
      row := tgbotapi.NewInlineKeyboardRow(
        tgbotapi.NewInlineKeyboardButtonData(link.Title, fmt.Sprintf("next_part %s",
link.TargetPartId)),
      )
      buttons = append(buttons, row)
    }

    var keyboard = tgbotapi.NewInlineKeyboardMarkup(
      buttons...,
    )

    msg.ReplyMarkup = keyboard
  }

  msg.ParseMode = "HTML"

  return msg
}

func Run() error {
```

```
c := config.Get()

bot, err := tgbotapi.NewBotAPI(c.TelegramToken())

if err != nil {
  log.Panic(err)
}

bot.Debug = c.Debug()
u := tgbotapi.NewUpdate(c.Offset())
u.Timeout = c.Timeout()

storiksAdapter := storiks.NewStoriksAdapter(resty.New(), c.StoriksHost())

client, err := mongodb.CreateMongoClient(c.MongoConnection())

if err != nil {
  log.Panic(err)
}

storage := mongodb.NewMongoStorage(client)

handlers := []telegram.UpdateHandler{
  telegram.StartCommandHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
  telegram.StartStoryHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
  telegram.MoveToNextPartHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
  telegram.FinishStoryHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
  telegram.InformationHandler{Bot: bot},
  telegram.BackToMenuHandler{Bot: bot, Storage: storage},
  telegram.SearchHandler{Bot: bot, Storage: storage},
  telegram.DoSearchHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
```

```
    telegram.ListTrailsHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.ListTrailsTurnPageHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.ContinueTrailHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.RestartTrailHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.ListStoriesHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.ListStoriesTurnPageHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.StoryInfoHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.ListBookmarksHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.ListBookmarksTurnPageHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.AddBookmarkHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.RemoveBookmarkHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
    telegram.UnknownActionHandler{Bot: bot, Storiks: storiksAdapter, Storage: storage},
  }

  updates := bot.GetUpdatesChan(u)

  for update := range updates {
    for _, handler := range handlers {
      ok, err := handler.Handle(update)

      if err != nil {
        log.Printf("%v", err)
        break
      }

      if ok {
        break
      }
    }
  }
```

```
  return nil
}



func main() {
  if err := app.Run(); err != nil {
    log.Fatal(err)
  }
}

module storiks_bot

go 1.18

require (
  github.com/go-resty/resty/v2 v2.7.0
  github.com/go-telegram-bot-api/telegram-bot-api/v5 v5.5.1
  github.com/google/uuid v1.3.0
  go.mongodb.org/mongo-driver v1.9.1
)

require (
  github.com/fsnotify/fsnotify v1.5.4 // indirect
  github.com/go-stack/stack v1.8.0 // indirect
  github.com/golang/snappy v0.0.1 // indirect
  github.com/google/go-cmp v0.5.8 // indirect
  github.com/hashicorp/hcl v1.0.0 // indirect
  github.com/klauspost/compress v1.13.6 // indirect
  github.com/magiconair/properties v1.8.6 // indirect
  github.com/mitchellh/mapstructure v1.5.0 // indirect
  github.com/pelletier/go-toml v1.9.5 // indirect
```

```
  github.com/pelletier/go-toml/v2 v2.0.1 // indirect
  github.com/pkg/errors v0.9.1 // indirect
  github.com/spf13/afero v1.8.2 // indirect
  github.com/spf13/cast v1.5.0 // indirect
  github.com/spf13/jwalterweatherman v1.1.0 // indirect
  github.com/spf13/pflag v1.0.5 // indirect
  github.com/spf13/viper v1.12.0 // indirect
  github.com/stretchr/testify v1.7.1 // indirect
  github.com/subosito/gotenv v1.3.0 // indirect
  github.com/xdg-go/pbkdf2 v1.0.0 // indirect
  github.com/xdg-go/scram v1.0.2 // indirect
  github.com/xdg-go/stringprep v1.0.2 // indirect
  github.com/youmark/pkcs8 v0.0.0-20181117223130-1be2e3e5546d // indirect
  golang.org/x/crypto v0.0.0-20220411220226-7b82a4e95df4 // indirect
  golang.org/x/net v0.0.0-20220520000938-2e3eb7b945c2 // indirect
  golang.org/x/sync v0.0.0-20201207232520-09787c993a3a // indirect
  golang.org/x/sys v0.0.0-20220520151302-bc2c85ada10a // indirect
  golang.org/x/text v0.3.7 // indirect
  golang.org/x/xerrors v0.0.0-20220517211312-f3a8303e98df // indirect
  gopkg.in/ini.v1 v1.66.4 // indirect
  gopkg.in/yaml.v2 v2.4.0 // indirect
  gopkg.in/yaml.v3 v3.0.0 // indirect
)
```

ДОДАТОК В

Міграції бази даних

```
CREATE TABLE users (
    id uuid NOT NULL,
```

```
    PRIMARY KEY(id)
);

CREATE TYPE story_status AS ENUM ('created', 'published');

CREATE TABLE stories (
    id uuid NOT NULL PRIMARY KEY,
    title varchar(256) NOT NULL,
    description varchar(2048) NOT NULL,
    created_at timestamptz NOT NULL,
    status story_status NOT NULL,
    owner_id uuid NOT NULL,
    labels jsonb NOT NULL,
    FOREIGN KEY (owner_id) REFERENCES users (id)
);

CREATE TYPE story_part_type AS ENUM ('beginning', 'regular', 'end');

CREATE TABLE story_parts (
    id uuid NOT NULL,
    title varchar(256) NOT NULL,
    content varchar NOT NULL,
    type story_part_type NOT NULL,
    story_id uuid NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY (story_id) REFERENCES stories (id),
);

CREATE TABLE story_links (
    title varchar(256) NOT NULL,
    source_part_id uuid NOT NULL,
```

```
    target_part_id uuid NOT NULL,
    FOREIGN KEY (source_part_id) REFERENCES story_parts (id),
    FOREIGN KEY (target_part_id) REFERENCES story_parts (id)
);

CREATE TABLE marks (
    user_id uuid NOT NULL,
    story_id uuid NOT NULL,
    part_id uuid NOT NULL,
    sequence_number int NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users (id),
    FOREIGN KEY (story_id) REFERENCES stories (id),
    FOREIGN KEY (part_id) REFERENCES story_parts (id),
    UNIQUE (sequence_number, story_id, user_id)

);

CREATE TABLE bookmarks (
    user_id uuid NOT NULL,
    story_id uuid NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users (id),
    FOREIGN KEY (story_id) REFERENCES stories (id),
    UNIQUE (user_id, story_id)
);

DROP TABLE bookmarks
DROP TABLE marks

DROP TABLE story_links;
DROP TABLE story_parts;
DROP TYPE story_part_type;
```

```
DROP TABLE stories;

DROP TYPE story_status;

DROP TABLE users;
```