Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

РОЗРОБКА КЛАЄНТ-СЕРВЕРНОГО ЗАСТОСУНКУ

Текстова частина до курсової роботи

за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи

с.в. _Борозенний С.О.__

(прізвище та ініціали)

_____

(підпис)

“____” _____ 2022 р.

Виконав студент _____

___Бойко Д. Р._____

(прізвище та ініціали)

“____” _____ 2022 р.

Київ 2022

Міністерство освіти і науки України

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

доцент _____

„____”_____2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Бойко Данилу Романовичу факультету інформатики 4-го курсу

ТЕМА: Розробка клієнт-серверного застосунку

Зміст ТЧ до курсової роботи:

Індивідуальне завдання

Вступ

Частина 1: Аналіз предметної області. Постановка завдання курсової роботи

Частина 2: Теоретичні відомості

Частина 3: Опис реалізації програмного продукту

Висновки

Список літератури

Додатки

Дата видачі „___” _____ 2022 р.

Керівник _____ (підпис)

Завдання отримав _____ (підпис)

Тема: Розробка клієнт-серверного застосунка

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітки
1	Отримання теми курсової роботи	10.11.2021	
2	Огляд технічної літератури за темою роботи	30.11.2021	
3	Встановлення необхідного програмного забезпечення	02.12.2021	
5	Написання основної частини курсової роботи	12.01.2022	
6	Перегляд змісту роботи з керівником	16.05.2022	
7	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	17.05.2022	

Студент Бойко Д. Р.

Керівник Борозений С. О.

“_____” _____

Зміст

Перелік умовних позначень

AWS – Amazon Web Serves

REST - Representational State Transfer

gRPC - Google Remote Procedure Call

Анотація

Робота присвячена розробці клієнт-серверного застосунку з використанням веб-сокетів. Застосунок дозволяє грати користувачам онлайн в гру «2048» та спілкуватися в чаті. Використовувались наступні технології: Golang (gin, gorilla), Redis та NATS. Для демонстрації застосунку був використаний AWS, EC2 instance.

Вступ

З плином часу вимоги до веб-розробників зростають. Кожна компанія має мати власну веб-сторінку, а клієнти веб-застосунків вибагливі до ресурсів, якими вони користуються. Через ці фактори веб-розробники мають вдосконалювати свої навички та постійно вчитися новим рішенням, які можуть їм допомогти. Тому для опанування таких технологій як веб-сокет та сервіс обробки повідомлень (в даному випадку NATS) була обрана відповідна тема курсової роботи.

Основна складність при роботі з веб-сокетами – це робота з багатопоточністю, адже відкритий веб-сокет не може записати одночасно два повідомлення. Також змінюється підхід при обробці відповіді від сервера. При REST або gRPC кожний запит повертає певну відповідь (статус 204 вважаємо також певною відповіддю), проте при роботі з веб-сокетами все працює трохи інакше. Ми можемо надсилати та отримувати повідомлення, які визначені будь-якою структурою. По-перше потрібні чіткі домовленості між сервером та клієнтом щодо повідомлень якими вони будуть обмінюватись, інакше не можливо буде правильно визначити, яку дію хоче виконати клієнт. Передача повідомлень через веб-сокет можлива в символьній формі або в бінарній формі.

Посилання на GitHub репозиторій - https://github.com/danilboiko1302/course-work

Посилання на демонстрацію застосунку - http://ec2-3-121-100-120.eu-central-1.compute.amazonaws.com/

Основна частина

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАВДАННЯ КУРСОВОЇ РОБОТИ

1.1. Аналіз предметної області та обґрунтування теми

Навички роботи з веб-сокетами є цінними для кандидата при пошуку роботи. Проаналізувавши велику кількість вакансій, стає зрозуміло, що такі навички не завжди потрібні для початкових спеціалістів, але все одне є перевагою. Для більш досвідчених спеціалістів – такі навички майже завжди є необхідними. Звісно такі навички не потрібні, якщо певний продукт не потребує веб-сокетів, проте це все одно буде перевагою для кандидата.

З переглянутих вакансій зрозуміло, що такі навички будуть корисними та збільшать цінність кандидата на посаду.

1.2. Постановка завдання курсової роботи

Задачею курсової роботи є розробка власного додатку з використанням веб-сокетів. Мова програмування використовується Golang. Для зберігання даних використовується розподілене сховище пар ключ-значення Redis[2]. Для коректної роботи з веб-сокетами, а саме надсилання повідомлення та отримання повідомлення для всіх користувачів використовується сервіс обробки повідомлень NATS[1] (ця система також реалізована на мові Golang).

Для клієнтської частини використовується html, css та чистий js. Всі повідомлення будуть передаватися в текстовому форматі, а саме в JSON. Більш оптимальним рішенням було б використовувати бінарних формат даних. Такий формат забезпечить вищу швидкість, проте з цим форматом складніше працювати, тому що не можна перевірити, які дані саме передаються.

Для правильної роботи з веб-сокетами використовуються засоби мови Golang. Структура chan[3] для роботи з багатопоточністю, аби уникнути ситуації, коли два різних потока спробують записати дані в веб-сокет одночасно.

РОЗДІЛ 2. ТЕОРЕТИЧНІ ВІДОМОСТІ

2.1. Огляд особливостей Golang для вирішення проблеми

Golang[4] – це мова програмування з статичною типізацією, з компілятором, розроблена Google. Синтаксис мови схожий до C, але має багато переваг відносно мови C. Go має garbage collector, структурну типізацію, безпечна робота при роботі з посиланнями на об’єкти та паралельність у стилі GSP. Цю мову називають часто саме Golang через його колишнє доменне ім'я golang.org, проте офіційна назва Go.
Особливості мови програмування Go[8]:

- Простота та зручність використання

Основна перевага мови Go – це простота. Синтаксис немає сильних відмінностей від інших мов програмування та легкий для вивчення. Особливо через схожість з C/C++, синтаксис виглядає знайомим.
Хоча йому не вистачає функціональних можливостей інших мов програмування, його область свідомо обмежена, щоб зробити його простим. Крім того, для тих, хто намагається працювати з багатопотоковими програмами, але їм не вдається, Golang є безпечним для таких завдань. Також компілятор цієї мови не дозволяє створення невикористаних змінних. Це означає, що при створенні змінної, яка не використовується, компілятор поверне помилку.

Незважаючи на простоту у використанні, ця мова не підходить для великих проектів (найчастіше ця мова використовується для мікросервісної архітектури, коли є багато невеликих додатків).
Також в цій мові особливе ставлення до помилок, вони вважаються звичайними змінними і це нормально повернути помилку для більшості системних бібліотек. В цій мові немає generics, exceptions та extensibility, override та деяких базових типів (set, замість нього треба використовувати

стороні бібліотеки чи map). Через це доводиться писати більше коду, який погіршує читабельність.

- Залежність віртуальної машини

Golang компілює файли з кодом у бінарні файли, тому для запуску не потребує віртуальної машини. Також залежності легко керуються, оскільки є спеціальний менеджер залежностей. Виконання без участі віртуальної машини забезпечує високу швидкість. Наприклад, в цій мові програмування будь-які цикли виконуються швидко в порівняні з іншими мовами.

При розробці застосунку використовувалися базові бібліотеки (пакети) та зовнішні бібліотеки. Особливість роботи з пакетами полягає в тому, що дуже легко створювати бібліотеки для інших користувачів. Достатньо опублікувати власну бібліотеку на GitHub. Цього достатньо, щоб будь-який користувач міг завантажити собі цю бібліотеку та користуватися нею.

Бінарні файли має свої недоліки. Основний з них це великий розмір. Програма «Hello world» буде займати 2МБ пам’яті. Тому багато зусиль було витрачено на оптимізацію та стискання цих файлів. Так наприклад невикористанні методи не будуть включені в двійкових файл.

- Автоматизація

Golang має такі переваги як: автоматичне оголошення змінних, швидкий час компіляції та збір сміття без затримок. Проте всі описанні переваги можуть стати недоліками, адже не можна сподіватися, що garbage collector запуститься точно тоді, коли це потрібно.
Існує багато різних бібліотек для роботи з http запитами. Серед них Banjo, Gin, goweb, Beego, Iris, Echo та Fiber. Для мого застосунку було обрано Gin[5] через декілька причин. По-перше я вже маю практичний досвід роботи з цією бібліотекою. По-друге ця бібліотека одна з найбільш популярних серед Golang розробників (50 тисяч зірок на GitHub

сторінці[5]). По-третє дана бібліотека містить весь необхідний функціонал: middleware, upload files, log, grouping routes, JSON, XML, YAML rendering та ще багато чого іншого. По-четверте ця бібліотека майже найшвидша серед усіх інших[6]. В мові Golang є окремий Benchmark, який дозволяє просто створювати різні порівняння для знаходження найкращого рішення. Benchmark оцінює кількість часу на операцію, кількість пам’яті на операцію в Байтах та кількість алокацій пам’яті.

Для роботи з веб-сокетами також існують різні бібліотеки. Серед них: STDLIB, gorilla[7], Gobwas, GOWebsockets. Для мого застосунку я обрав бібліотеку gorilla. Ця бібліотека немає найбільший функціонал з переліку, проте як раз відсутня надлишковість.

2.2. Огляд особливостей Redis для вирішення проблеми

Для зберігання усієї необхідної інформації було використано Redis. Redis – це сховище типу ключ-значення, яке зберігає інформацію в оперативній пам’яті, за рахунок чого досягається висока швидкість. Також Redis може зберігати копію даних на постійній пам’яті для відновлення цих даних (або частини) після перезавантаження системи.

Альтернативно можна було використати SQL базу даних або одну з NoSQL, проте основна увага була приділена саме роботі з веб-сокетами, тому сховище для даних було обране примітивне та легке для роботи.

Переваги при роботі з Redis[9]:

- Швидкість, оскільки дані зберігаються в оперативній пам’яті
- Дуже легкий в налаштуванні
- Підтримка майже всіх можливих структур, якщо не можливо зберегти структуру – завжди можна обгорнути дану структуру в JSON.
- Максимальний розмір ключа-пари 512 МБ.
- Redis використовує власний алгоритм хешування (Redis Hashing)

- Під час збільшення використовує мало ресурсів
- Весь код є у відкритому доступі

Недоліки:

- Використання оперативної пам’яті, якої набагато менше ніж постійної
- Кластер Redis має власну топологію, тому необхідно правильно конфігурувати кожного клієнта

2.3. Огляд особливостей NATS для вирішення проблеми

Для коректної роботи з веб-сокетами необхідно використовувати брокер повідомлень[10]. Під час підключення необхідно «підписатися» на брокер. Обирається певний ідентифікатор, за яким можна відокремити користувачів, які мають отримувати повідомлення. Наприклад якщо це чат – тоді назву кімнати можна обрати як ідентифікатор, щоб користувачі отримували повідомлення від інших учасників в цій самій кімнаті. Після того як новий користувач успішно «підписався» на брокер повідомлень, користувач зможе отримувати повідомлення від інших користувачів. Як правило метод, який «підписує» користувача повертає помилку, якщо щось пішло не так, та метод, який необхідно викликати, щоб відписатися від брокера під час розриву з’єднання або завершенню роботи.

Для відправлення необхідно «опублікувати» повідомлення, тоді його отримають всі користувачі, які перед цим підписалися на брокер. Важливо що користувач, який відправив повідомлення також його отримає – адже він підписаний до брокера. Після отримання повідомлення кожний користувач вже робить певні дії описані бізнес-логікою.

Для таких брокерів краще використовувати бінарні типи даних, тому що дані передаються саме в бінарному вигляді. Також можна передавати текст, який легко переводити в та з бінарного типу.

Redis також використовується як брокер повідомлень.

Для мого застосунку було обрано брокер повідомлень NATS, який реалізований на мові Golang. Можна встановити NATS або використовувати docker образ.

Також існують також багато інших брокерів:

- RabbitMQ.
- Kafka
- ZeroMQ
- Microsoft Azure Service Bus
- Oracle Message Broker
- Google Cloud Pub/Sub
- Eclipse Mosquitto MQTT Broker
- Amazon MQ

Всі брокери відрізняються пропускною можливістю, структурою та варіантами взаємодії.

2.4. Огляд особливостей AWS та Docker для вирішення проблеми

AWS[12] та Docker[13] були використані для розгортання застосунку на EC2[11]. Docker використовувався для двох основних задач. По-перше це розробка застосунка на двох різних системах (Windows та MacOS).

Завдяки Docker не потрібно встановлювати будь-які застосунки, оновлювати бібліотеки чи слідкувати які версії застосунків встановлено.

Доступні хмарні сервіси – це AWS, Google Cloud Platform, Microsoft Azure DigitalOcean. Був обраний саме AWS через вже наявний практичний досвід роботи.

РОЗДІЛ 3. ОПИС РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

3.1. Опис реалізації контролера

В додатку А доступний код контролера.

Контролер оброблює тільки три запита. Два GET запита для повернення статичних сторінок та один для роботи з веб-сокетом. Контролер завантажує необхідні статичні файли та готовий до роботи. Для підключення веб-сокету небхідно «покращити» підключення. Для цього використовується websocket.Upgrader (рисунок 3.1). В цій структурі можна сказати бажаний розмір буфера для отримання на надсилання даних, як довго сервер буде очікувати для підтвердження Handshake, яким чином необхідно повернути помилку та перевірку, якому підключенню дозволяється створити веб-сокет з'єднання. Потім необхідно переконатися, що з'єднання закриється в кінці – для цього використовується defer ws.Close(). Defer – гарантує що ця функція буде викликана вкінці.

```
var upGrader = websocket.Upgrader{
    CheckOrigin: func(r *http.Request) bool {
        return true
    },
}
```

Рисунок 3.1

```
go func() {
    <-c.Request.Context().Done()
}()
```

Рисунок 3.2

На рисунку 3.2 зображено код, який відповідальний за завершення всіх процесів, які розпочали свою роботу, коли клієнт під'єднався. Це анонімна функція, яка запускається в окремому потоці. <-c.Request.Context().Done() блокую потік, поки клієнт не від'єднається. Тут необхідно «відписатися» від

брокеру повідомлень та сповістити інших клієнтів, що цей клієнт закінчує свою роботу.

Для надсилання повідомлення до клієнта використовується структура chan. Спочатку ми створюємо об’єкт цієї структури, визначаєм що ми будемо передавати тип string та розмір 1. Розмір вказує скільки об’єктів буде зберігатися в буфері. Якщо буфер заповнено – тоді при спробі покласти ще один об’єкт призведе до блокування потоку, поки в буфері не звільниться місце. Це забезпечить безпечну роботу з веб-сокетом.

```
messagesChan := make(chan string, 1)
go func() {
    for {
        message := <-messagesChan
        if user.LoggedIn {
            ws.WriteMessage(websocket.TextMessage, []byte(message))
        }
    }
}()
```

Рисунок 3.3

В окремому потоці ми запускаєм анонімну функцію, яка буде в безкінечному циклі брати повідомлення з chan та передавати його в веб-сокет. Якщо буфер пустий, то спроба отримати повідомлення призведе до блокування потоку, поки в буфері не з’явиться повідомлення. При від’єднанні клієнта цей chan закриється.

3.2. Опис реалізації користувача

```
type User struct {      You, last week • ga
    Name     string `json:"name"`
    LoggedIn bool   `json:"loggedIn"`
}
```

Рисунок 3.4

На рисунку 3.4 зображена структура користувача. Для розробленого функціоналу достатньо ім’я користувача та прапорець, чи користувач увійшов до кімнати.

Коли клієнт під єднався до веб-сокета, він одразу підписується до брокера повідомлень, проте він отримає повідомлення виключно після того, як зайде до кімнати.

Ім’я користувача необхідне для ідентифікації його дій.

Посилання на об’єкт користувача передається до методу, який може змінювати будь-яке поле відповідно до бізнес-логіки.

3.3. Опис реалізації брокера повідомлень

```
type Publisher interface {
    Pub(topic string, data []byte) error
}

You, 2 weeks ago | 1 author (You)
type Subscriber interface {
    Sub(topic string, cb func(data []byte)) (unsub func() error, err error)
}

You, 2 weeks ago | 1 author (You)
type PubSub interface {
    Publisher
    Subscriber
}
```

Рисунок 3.5

Для роботи з брокером повідомлень я визначив інтерфейс, який зображений на рисунку 3.5.

Реалізація самого брокеру зображена в додатку Б.

Метод Init() відповідальний за ініціалізацію з’єднання з сервером брокера повідомлень. Всі посилання визначено в конфігураційному файлі (.env). Якщо з’єднання не встановлено – тоді застосунок не почне свою роботу.

Метод Close() відповідальний за закриття з’єднання, коли застосунок завершить свою роботу.

Метод Sub(topic string, cb func(data []byte)) (unsub func() error, err error) – метод, який підписує клієнта на отримання повідомлень. Приймає два параметра. Перший – це ідентифікатор, за яким брокер повідомлень буде

знати яким користувачам надсилати повідомлення. Другий – це функція, яка буде викликана, коли брокер отримає повідомлення. Метод повертає два параметра, помилку, якщо щось пішло не так при спробі підписатися та функцію, яка відпише клієнта від брокера.

```
unsub, err := nats.Connection.Sub(roomName, func(data []byte) {
    messagesChan <- string(data)
})

go func() {
    for {
        message := <-messagesChan
        if user.LoggedIn {
            ws.WriteMessage(websocket.TextMessage, []byte(message))
        }
    }
}()

go func() {
    <-c.Request.Context().Done()
    err := unsub()
}()
```

Рисунок 3.6

На рисунку 3.6 зображено використання брокера повідомлень в контролері.

Спочатку клієнт підписується на брокер повідомлень. Ідентифікатор використовується назва кімнати. Функція, яка буде викликатися при отримані повідомлення дуже проста. Необхідно отриману інформацію від брокера передати до об’єкта chan. Далі контролер візьме повідомлення з об’єкта chan та відправить до клієнта через веб-сокет. Більше детально описано в розділах 3.1 та 2.1

Метод Pub(topic string, data []byte) error. Даний метод відповідальний за надсилання повідомлення до брокера повідомлень, який в свою чергу надсилає його всім підписаним клієнтам. Якщо надіслати повідомлення до

брокера не вийшло, тоді метод поверне помилку, nil значення якщо успішно.

Використання даного методу зображено в додатку В.

Застосунок отримує повідомлення від клієнта, передає його в метод сервісу HandleMessage (більш детатльно в розділі 3.5), цей метод повертає три параметри. Якщо параметр pubMsg не nil, тоді це повідомлення передається до брокера повідомлень.

3.4. Опис реалізації сховища Redis

В додатку Г зображено ініціалізація з'єднання, завершення роботи та очистка сховища.

Метод Init() використовується для встановлення з'єднання з сховищем. Якщо воно успішне створюється об'єкт структури, яка зображена на рисунку 3.7. *redis.Client – це саме з'єднання, яке буде використовуватись для виконання всіх запитів. context.Context – це контекст в мові програмування Golang. Він необхідний для виконання запитів, щоб не створювати кожний раз новий контекст, буде використовуватись контекст створений при з'єднані з сховищем.

При кожному запуску застосунку все сховище очищується.

```
type RedisSession struct {
    Client *redis.Client
    Ctx    context.Context
}
```

Рисунок 3.7

В сховищі Redis зберігаються користувачі кожної кімнати, адмін кожної кімнати, ігрове поле та повідомлення. Тільки адмін має дозвіл робити наступний крок. Адмін може передати свої повноваження будь-якому іншому користувачу. Якщо адмін виходить з кімнати – інший

користувач в цій кімнаті стає адміном. Якщо кімната пуста та немає адміна – перший користувач, який зайде до кімнати стане адміном.

Користуватися чатом може кожний користувач в кімнаті.

3.5. Опис бізнес-логіки

Коли сервер отримав повідомлення від користувача, це повідомлення передається в метод HandleMessage. HandleMessage(room string, msg types.Message, user *types.User) (*types.MessageFront, *types.MessageFront, error). Для правильного опрацювання повідомлення нам необхідна назва кімнати, саме повідомлення та клієнт. Назва кімнати визначається під час підключення користувача.

```
type Message struct {
    Action Action `json:"action"`
    Data   string `json:"data"`
}
```

Рисунок 3.8

Структура повідомлення, яке сервер отримує від користувача зображене на рисунку 3.8. Тип повідомлення визначає, яку дію хоче виконати користувач. Data має тип string тому що в цей тип можна передати будь-які дані серіалізовані в JSON.

В додатку Д зображено всі можливі типи повідомлення. iota – це спеціальна зміна, яка дорівнює 0 і збільшується на 1 при кожному виклику цієї змінної постінкрементно. Відповідно кожний наступний тип буде + 1 до попередього значення. Тип ErrorAction використовується як помилковий тип. Це пов’язано з тим, що при спробі отримати повідомлення, тип повідомлення буде дорівнювати 0 (дефолтне значення), якщо його не задав користувач.

Спершу перевіряється тип повідомлення - switch msg.Action. Якщо такого типу повідомлення немає, тоді повертається помилка про те що це

невідомий тип повідомлення (невідома дія) - return nil, nil, errors.New(vocabulary.UNKNOWN_ACTION).

Якщо повідомлення має відомий тип – тоді повідомлення оброблюється відповідним методом.

Користувач може виконати наступні дії:

- Зайти в кімнату (login)
- Вийти з кімнати (logout)
- Отримати список користувачів в кімнаті (getUsers)
- Передати права адміна іншому користувачу в кімнаті (setAdmin)
- Отримати ігрове поле (getField)
- Зробити хід (move)
- Надіслати повідомлення в чаті (sendMessage)
- Отримати історію повідомлень (getHistroy)

Всі дії користувач може виконувати виключно після того, як він зайде до кімнати (откріім login). Передати права адміна іншому користувачу та зробити хід може виключно адмін кімнати.

Метод HandleMessage повертає три значення. Помилку, якщо виникли якісь помилки підчас дії користувача (наприклад немає дозволу для виконання цієї дії). Повідомлення, яке буде надіслано тільки цьому ж користувачу (наприклад отримати поле чи історію повідомлень). Повідомлення, яке буде надіслано усім користувачам кімнати через брокер повідомлень (наприклад коли адмін зробить хід, оновлене ігрове поле отримає кожний користувач кімнати).

```
const (
    Size       int = 4
    AmountOf2  int = 2
    AmountOf4  int = 1
    RandomFor4 int = 25
)
```

Рисунок 3.9

Якщо поля для гри не існує або користувач програв – необхідно створити нове ігрове поле. Необхідні для цього константи зображені на рисунку 3.9. Size – розмір поля, 4 для класичної гри. AmountOf2 та AmountOf4 – початкова кількість символів 2 та 4 відповідно. RandomFor4 – ймовірність випадання 4 замість 2 при кожному ході.

3.6. Опис роботи з багатопоточністю

В застосунку міститься небезпека при роботі з багатопоточністю в двох місцях. Перший випадок – це запис даних до веб-сокета. Другий – це зберігання повідомлень.

В першому випадку рішення було описано в розділі 3.1 та 2.1.

В другому використовується інший підхід, оскільки в першому випадку всі спроби запису до веб-сокету відбуваються виключно для одного клієнта. Тут необхідно правильно оброблювати процеси від усіх користувачів в кімнаті.

В додатку Е зображено шлях вирішення цієї проблеми.

var chans map[string]chan string = make(map[string]chan string)

Цей об’єкт використовується для зберігання об’єктів типу chan для кожної кімнати, де ключ – це назва кімнати.

Клієнт просить надати йому об’єкт до якого він зможе передати повідомлення (getChan). Якщо такий об’єкт існує, тоді метод просто поверне його. Якщо такого об’єкту не існує, його необхідно створити, зберегти та повернути.

Процес створення складається з 4 пунктів:

- Створити об’єкт
- Створити таймер
- Зберегти об’єкт
- Оброблювати отримані повідомлення

Пункти 1-3 прості та не потребують пояснень. Пункт 4 складається з анонімної функції, яка запускається в окремому потоці. Якщо надходить

повідомлення до об’єкта chan, тоді його необхідно зберегти (інші клієнти не зможуть передати туди повідомлення, поки не закінчиться обробка даного повідомлення). Якщо спливає таймер, тоді ця функція завершує свою роботу та прибирає створений об’єкт. Це зроблено з метою очищення пам’яті. Застосунок не буде вічно зберігати об’єкти типу chan, а створювати нові при потребі.

3.7. Опис UI

Клієнтська частина застосунку складається з двох сторінок. Перша сторінка необхідна лише для того, щоб клієнт ввів назву кімнати. Друга сторінка містить весь функціонал.

Спочатку клієнту необхідно ввести ім’я. Якщо таке ім’я в цій кімнаті вже існує, тоді клієнт не зможе зайти до кімнати.

Кімната поділена на три частини (колонки).

Перша колонка містить кнопку logout, список користувачів та кнопки, щоб зробити певного користувача адміном.

Друга колонка містить поле гри та 4 кнопки, які роблять один з чотирьох можливих кроків.

Третя колонка містить чат.

Для правильної роботи застосунку вкрай необхідно, щоб типи повідомлень, які відправляє клієнт та отримує співпадали з типами повідомлень на сервері. Інакше неможливо правильно ідентифікувати дію клієнта.

Весь UI виконано з використанням html, css та js. Дизайн простий та легкий у розумінні.

3.8. Опис деплою застосунку до хмарного сервісу

Процес деплою складається з наступних кроків:

- Створення необхідний файлів Docker (Dockerfile для застосунку та docker-compose.yml). Dockerfile для застосунку немає ніяких особливостей. Основа використовується образ golang:latest,

вказується робоча папка, копіюються необхідні файли, встановлюються всі модулі та задається порт. Файл docker-compose.yml містить перелік усіх необхідних сервісів для роботи застосунку – це Redis та NATS. Також створюється мережа, вказуються необхідні образи, залежності, порти та назви сервісів. Цей файл буде використовуватись для побудови та запуску застосунку.

- Створення екземпляру EC2. Для цього необхідно мати аккаунт AWS. Для роботи з AWS не бажано використовувати root аккаунт, рекомендації щодо безпеки радять створити IAM користувача, який буде мати усі необхідні дозволи для роботи. Екземпляр EC2 був створений на основі Ubuntu (через наявність практичних навичок) зі звичайними налаштуваннями по швидкістю процесору, обсягу пам’яті та інше. Необхідно налаштувати пару ssh ключів, щоб мати можливість підключити до екземпляру через ssh. Також за замовченням екземпляр не зможе отримувати будь-які запити за протоколом http, це потрібно додати в розділі «мережа». Також є можливість вказати які саме ip адреси зможуть отримати дозвіл до екземпляру по протоколу http, в даному випадку я вказав, що вказав будь-хто зможе надсилати запити по протоколу http до екземпляра.
- Копіювання та запуск застосунку. Спершу необхідно підключитися до екземпляра по протоколу ssh, використовуючи ключ створений в попередньому кроці. Далі потрібно встановити усе необхідне програмне забезпечення, в даному випадку це тільки docker. Після встановлення буде зручно налаштувати групу в Ubuntu так, щоб команду docker можна було запустити без використання команди sudo, при бажанні цей момент можна пропустити. Використовуючи власний Personal access token з GitHub треба скопіювати репозиторій до екземпляру. Зауважте, що необхідно використовувати саме

Personal access token, адже GitHub прибрав можливість зробити те саме за допомогою логіну та паролю. Залишилось тільки перейти в директорію з застосунком та запустити наступну команду «docker compose up –build». Всі необхідні образи для побудови застосунку завантажаться та запуститься застосунок. В самому файлі docker-compose.yml вказано, що порт 80 екземпляру (порт за замовчуванням для протоколу http) буде асоційовано з портом 8000 docker контейнера. Після цього екземпляр зможе отримувати http запити та передавати їх до застосунку для опрацювання.

Список використаної літератури

1. [Електронний ресурс] NATS Messaging - https://en.wikipedia.org/wiki/NATS_Messaging
2. [Електронний ресурс] Redis - https://en.wikipedia.org/wiki/Redis
3. [Електронний ресурс] Channels - https://go.dev/tour/concurrency/2
4. [Електронний ресурс] Golang - https://go.dev/
5. [Електронний ресурс] Gin - https://github.com/gin-gonic/gin
6. [Електронний ресурс] Gin Benchmark - https://github.com/gin-gonic/gin#benchmarks
7. [Електронний ресурс] Gorilla - https://github.com/gorilla/websocket
8. [Електронний ресурс] Pros and Cons Golang - https://www.mindinventory.com/blog/pros-and-cons-programming-in-golang/
9. Pros and Cons Redis - https://medium.com/weekly-webtips/redis-what-and-why-pros-cons-ae2f5bc750fd
10. Message broker - https://en.wikipedia.org/wiki/Message_broker
11. EC2 - https://uk.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud
12. AWS - https://uk.wikipedia.org/wiki/Amazon_Web_Services
13. Docker - https://uk.wikipedia.org/wiki/Docker

Додатки

Додаток А. Контролер

```
func GetRoute() *gin.Engine {
    route := gin.Default()

    route.LoadHTMLGlob("app/pages/html/*.html")

    route.Static("/js", "app/pages/js")
    route.Static("/css", "app/pages/css")

    route.GET("/", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.html", nil)
    })
    route.GET("/:roomName", func(c *gin.Context) {
        c.HTML(http.StatusOK, "room.html", nil)
    })
    room := route.Group("/room")
    {
        room.GET("/:roomName", func(c *gin.Context) {
            ws, err := upGrader.Upgrade(c.Writer, c.Request, nil)
            defer ws.Close()
            messagesChan := make(chan string, 1)
            go func() {
                for {
                    message := <-messagesChan
                    if user.LoggedIn {
                        ws.WriteMessage(websocket.TextMessage, []byte(message))
                    }
                }
            }()
            go func() {
                <-c.Request.Context().Done()
            }()
            for {
                var msg types.Message
                err = ws.ReadJSON(&msg)
                pubMsg, myMsg, err := service.HandleMessage(roomName, msg, user)
                if myMsg != nil {
                    bytes, err := json.Marshal(myMsg)
                    messagesChan <- string(bytes)
                }
            }
        })
    }
    return route
}
```

Додаток Б. Брокер повідомлень

```go
import (
    "os"

    "github.com/nats-io/nats.go"
)

type PubSub struct {
    Conn *nats.Conn
}

var Connection *PubSub

func Init() error {
    url := os.Getenv("NATS_URL")

    connection, err := nats.Connect(url)

    if err != nil {
        return err
    }

    Connection = &PubSub{Conn: connection}

    return nil
}

func Close() {
    Connection.Conn.Drain()
    Connection.Conn.Close()
}

func (ps *PubSub) Pub(topic string, data []byte) error {
    return ps.Conn.Publish(topic, data)
}

func (ps *PubSub) Sub(topic string, cb func(data []byte)) (unsub func() error, err error) {
    s, err := ps.Conn.Subscribe(topic, func(msg *nats.Msg) {
        cb(msg.Data)
    })

    if err != nil {
        return nil, err
    }

    return s.Unsubscribe, nil
}
```

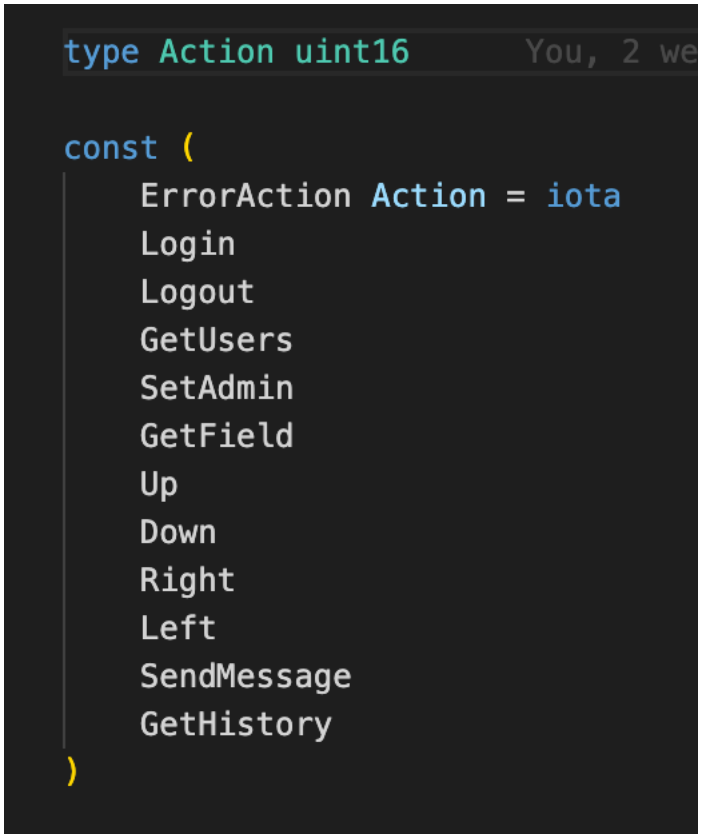
Додаток В.

```
var msg types.Message
err = ws.ReadJSON(&msg)

if err != nil {
    log.Println("error read json")
    return
}

pubMsg, myMsg, err := service.HandleMessage(roomName, msg, user)

if err != nil {
    log.Println(err.Error())

    bytes, err := json.Marshal(gin.H{"error": err.Error()})
    if err != nil {
        log.Println("error json.Marshal: " + err.Error())
        continue
    }
    messagesChan <- string(bytes)
    continue
}

if pubMsg != nil {
    err = service.Pub(roomName, pubMsg)
    if err != nil {
        log.Println("error pub msg nats: " + err.Error())
    }
}

if myMsg != nil {
    bytes, err := json.Marshal(myMsg)
    if err != nil {
        log.Println("error json.Marshal: " + err.Error())
        continue
    }
    messagesChan <- string(bytes)
}
```

Додаток Г. Redis

```go
var Session *types.RedisSession

func Init() error {
    var ctx context.Context = context.Background()

    url := os.Getenv("REDIS_URL")

    client := redis.NewClient(&redis.Options{
        Addr:     url,
        Password: "",
        DB:       0,
    })

    _, err := client.Ping(ctx).Result()
    if err != nil {
        return fmt.Errorf(vocabulary.REDIS_CONNECTION_PROBLEM, err.Error())
    }

    Session = &types.RedisSession{
        Ctx:    ctx,
        Client: client,
    }

    clearAll()

    return nil
}

func clearAll() {
    Session.Client.FlushAll(Session.Ctx)
}

func Close() {
    Session.Client.Close()
}
```

Додаток Д. Всі можливі дії користувача


type Action uint16 You, 2 we
const (
ErrorAction Action = iota
Login
Logout
GetUsers
SetAdmin
GetField
Up
Down
Right
Left
SendMessage
GetHistory
)

Додаток Е. Робота з багатопоточністю при зберіганні повідомлення з чату.

```
var chans map[string]chan string = make(map[string]chan string)

var limitForTimer time.Duration = 10 * time.Second

func getChan(room string) chan string {
    messageChan := chans[room]

    if messageChan != nil {
        return messageChan
    }

    newChan := make(chan string, 1)
    timer := time.NewTimer(limitForTimer)
    chans[room] = newChan

    go func() {
        open := true
        for open {
            select {
            case message := <-newChan:
                {
                    saveMessageRedis(room, message)
                }
            case <-timer.C:
                {
                    chans[room] = nil
                    open = false
                    break
                }
            }
        }
    }()

    return newChan
}
```