НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Лабораторна робота №1

з предмету «Проектування розподілених систем»

Виконав:

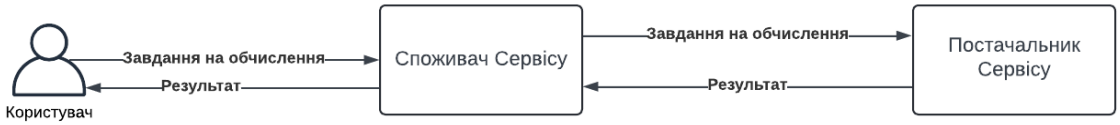
студент групи ІМ-31мн,

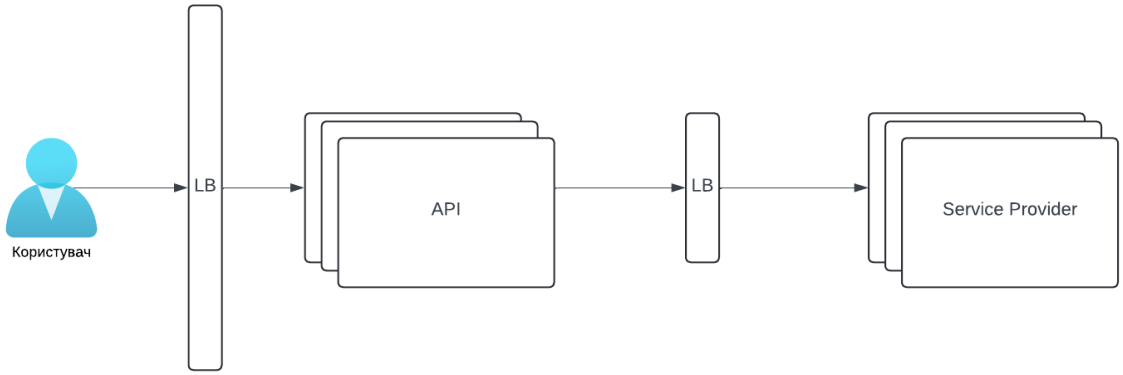
Онищук Микола

Київ 2024

Завдання

- Реалізувати синхронну комунікація між 2ма сервісами. Споживач Сервісу генерує завдання на обчислення і чекає відповіді від Постачальник Сервісу.
- Постачальник Сервісу має підраховувати час обчислення і логувати його для подальшого аналізу
- Споживач Сервісу має підраховувати час виконання запиту і логувати його для подальшого аналізу
- Розгорнути Load Balancer перед Споживачем Сервісу і/або Постачальником сервісу
- Опціонально: реалізувати протокол gRPC
- Опціонально: авторизація на рівні Споживача Сервісу
- Опціонально: авторизація на рівні Постачальника Сервісу





Виконання

Лабораторну роботу було виконано на мові Golang та розгорнуто у Docker. Також було використано Nginx для балансування навантаження. Авторизацію було реалізовано за допомогою jwt-токенів.

Було виконано всі пункти окрім реалізації протоколу gRPC.

Розроблена система містить балансувальники навантаження для споживача та постачальника, 3 інстанси споживача та 3 інстанси постачальника.

Сервіс споживача має 2 ендпоінти: “/randomize_jwt” - для генерації випадкового jwt-токена та “/create_task” - для створення завдання про

обчилення та надсилання запиту до постачальника, а сервіс постачальника 1 - “/compute” - для математичних обчислень за даними, надісланими споживачем.

Протестуємо роботу, надсилаючи запити за допомогою “Postman”.

Спочатку надішлемо GET-запит за адресою http://localhost:80/randomize_jwt та отримаємо у відповідь згенерований токен (рис. 1).

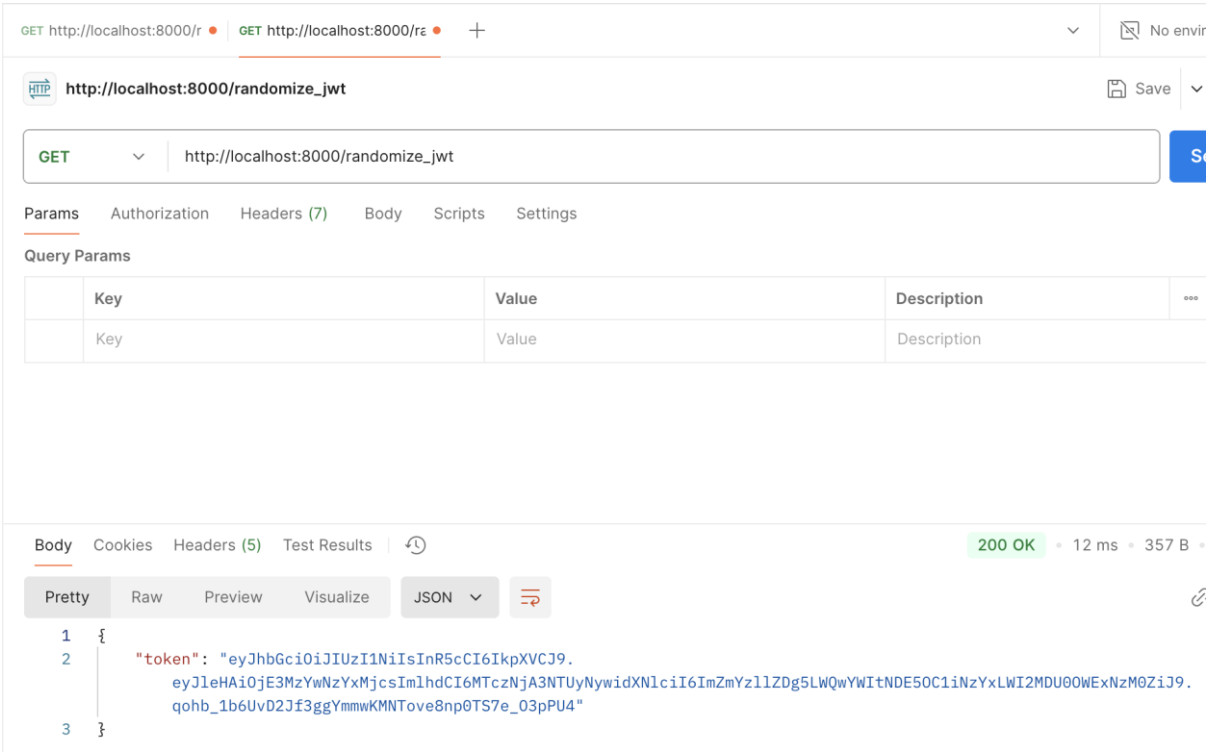
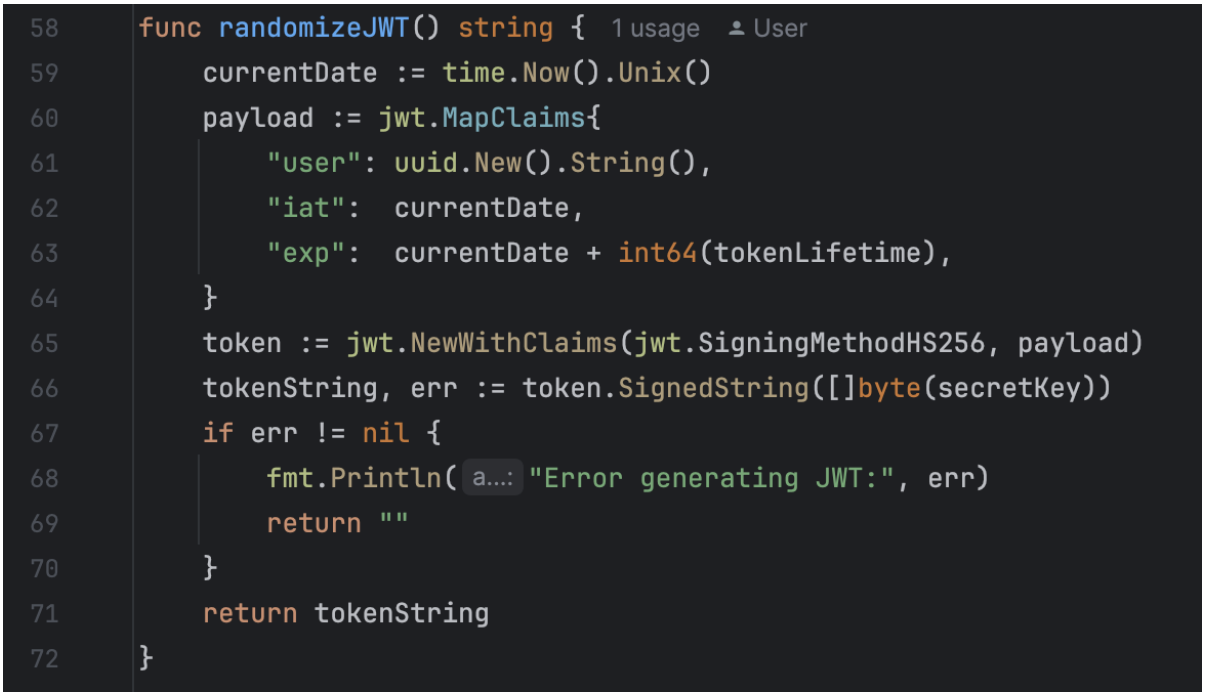


Рисунок 1 - Генерація токену.

Генерація токену відбувається у наступному блоці коду (рис. 2).



```
func randomizeJWT() string {  1 usage  User
    currentDate := time.Now().Unix()
    payload := jwt.MapClaims{
        "user": uuid.New().String(),
        "iat":  currentDate,
        "exp":  currentDate + int64(tokenLifetime),
    }
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, payload)
    tokenString, err := token.SignedString([]byte(secretKey))
    if err != nil {
        fmt.Println( a...: "Error generating JWT:", err)
        return ""
    }
    return tokenString
}
```

Рисунок 2 - Код генерації токену.

Тепер відправимо GET-запит споживачу за адресою http://localhost:8000/create_task. Тут необхідно також додати параметр “input_data”. Надамо йому значення 3299. Тому шлях виглядатиме наступним чином: http://localhost:8000/create_task?input_data=3299. Також до заголовку запиту додамо ключ “Authorization” та значення, що містить префікс “Bearer ” та jwt-ключ згенерований попередньо (рис. 1). У якості відповіді отримаємо імена інстансів споживача та постачальника, що обробили запит, час, що пішов на обробку запиту споживачем та на обчислення постачальником, а також результат, отриманий внаслідок обчислень (рис. 3). Адже споживач у свою чергу відправляє запит постачальнику та повертає нам його відповідь разом з іншими даними.

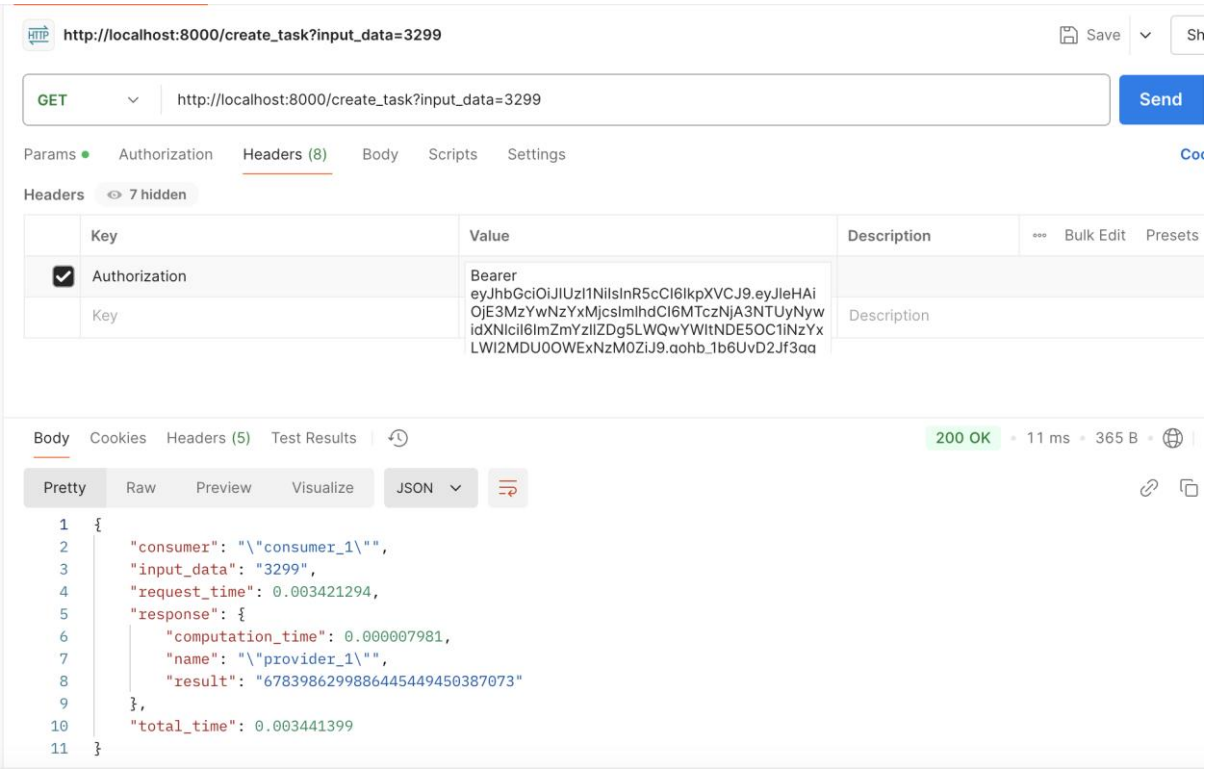


Рисунок 3 - Запит на створення завдання про обчислення.

Обробка отриманих даних (рис. 4) та підготовка і відправка запиту постачальнику (рис. 5) відбувається в наступних блоках коду.

```
77      inputData := r.URL.Query().Get( key: "input_data")
78      if inputData == "" {
79          http.Error(w, error: `{"error":"Input data is required"}`, http.StatusBadRequest)
80          return
81      }
82
83      authHeader := r.Header.Get( key: "Authorization")
84      if authHeader == "" {
85          http.Error(w, error: `{"error":"Authorization header missing"}`, http.StatusUnauthorized)
86          return
87      }
```

Рисунок 4 - Обробка даних з GET-запиту.

```
    headers := map[string]string{
        "Authorization": authHeader,
        "Accept":        "application/json",
        "Content-Type":  "application/json",
    }

    bodyData := map[string]string{
        "input_data": inputData,
    }
    body, err := json.Marshal(bodyData)
    if err != nil {
        http.Error(w, error: `{"error":"Failed to encode JSON body"}`, http.StatusInternalServerError)
        return
    }

    req, err := http.NewRequest( method: "POST", providerUrl, bytes.NewBuffer(body))
    if err != nil {
        http.Error(w, error: `{"error":"Failed to create request"}`, http.StatusInternalServerError)
        return
    }
    for key, value := range headers {
        req.Header.Set(key, value)
    }

    client := &http.Client{}
    startTime := time.Now()
    resp, err := client.Do(req)
    if err != nil {
        http.Error(w, error: `{"error":"Failed to connect to provider"}`, http.StatusInternalServerError)
        return
    }
```

Рисунок 5 - Підготовка і відправка запиту постачальнику.

Спробуємо надіслати запит постачальнику напряму. Це POST-запит за адресою http://localhost:80/compute. У якості відповіді отримаємо частину відповіді з попереднього запиту, що містилась у полі “response” (рис. 6). Ми так само додаємо до заголовку запиту поле “Authorization”, а також переносимо “input_data” з параметрів до тіла запиту (рис. 7).

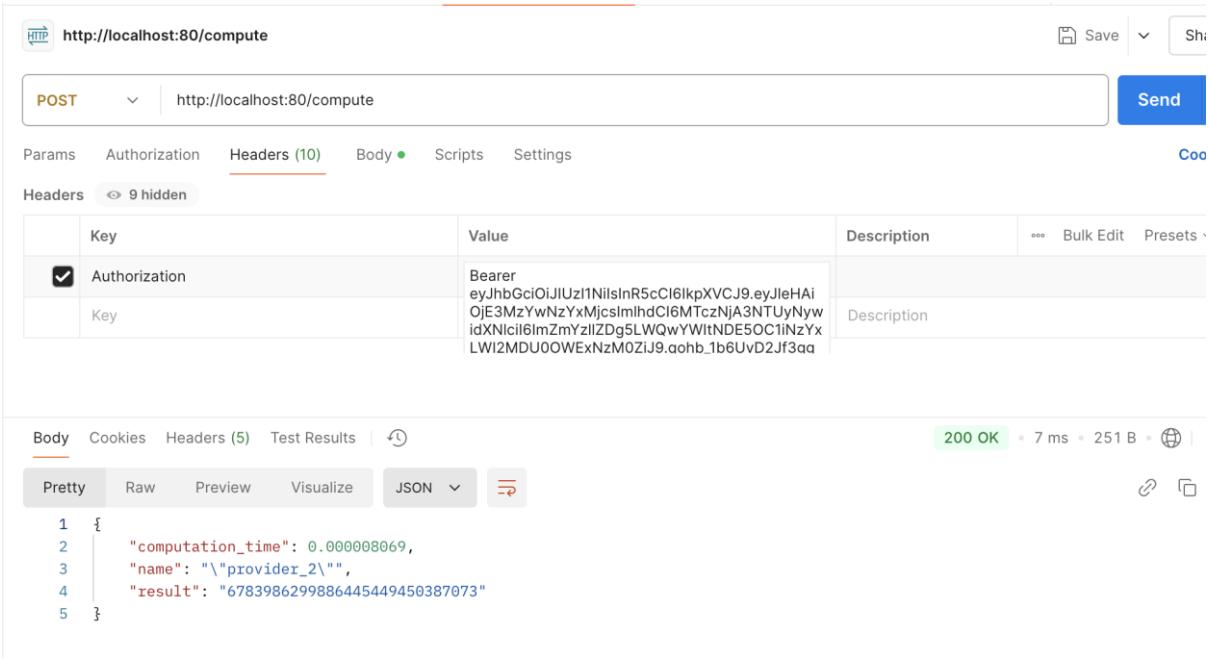


Рисунок 6 - Запит на обчислення постачальнику.

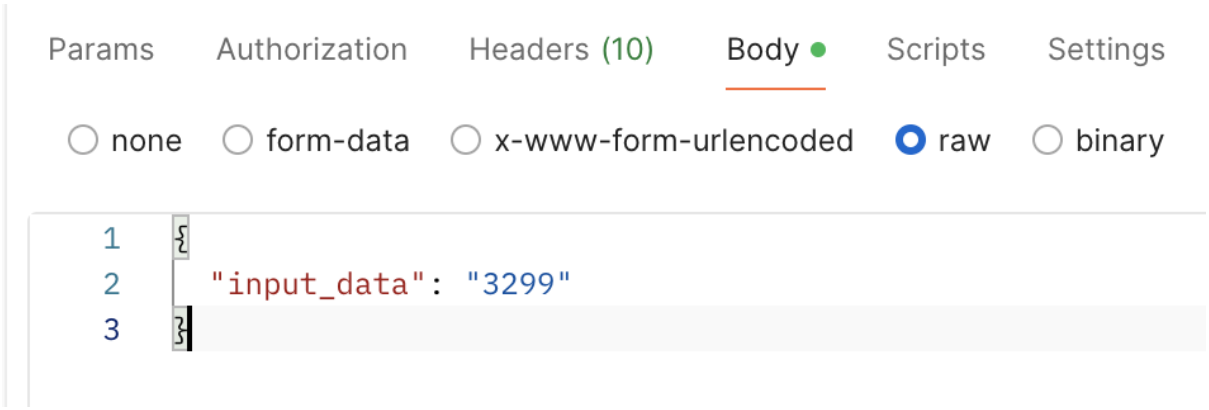


Рисунок 7 - ”input_data” у тілі запиту.

Обчислення та формування відповіді від постачальника відбувається у наступному блоці коду (рис. 8). В якості обчислень знаходиться сума геометричної прогресії з n=101, де початковим елементом є “input_data”, а знаменник дорівнює 2. Потім це отримана сума підноситься до модулю 3^{60} для зменшення отриманого числа.

```go
sum := new(big.Int)
currentTerm := new(big.Int).Set(number)

for i := 0; i < 100; i++ {
    sum.Add(sum, currentTerm)
    currentTerm.Mul(currentTerm, big.NewInt( x: 2))
}

modulus := new(big.Int).Exp(big.NewInt( x: 3), big.NewInt( x: 60), m: nil) // 3^60
sum.Mod(sum, modulus)

computationTime := time.Since(startTime).Seconds()

fmt.Printf( format: "Computed result for input_data=%s in %.6f seconds. Result modulo 3^60: %s\n", inputData, computationTime, sum.String())

response := map[string]interface{}{
    "result":           sum.String(),
    "computation_time": computationTime,
    "name":             name,
}
w.Header().Set( key: "Content-Type", value: "application/json")
if err := json.NewEncoder(w).Encode(response); err != nil {
    http.Error(w, error: `{"error":"Failed to encode response"}`, http.StatusInternalServerError)
}
```

Рисунок 8 - Проведення обчислень та формування відповіді.

Також внаслідок надісланих запитів отримано наступні логи (рис. 8).

```
load_balancer_c-1  | 172.18.0.1 - - [05/Jan/2025:11:12:07 +0000] "GET /randomize_jwt HTTP/1.1" 200 202 "-" "PostmanRuntime/7.43.0" "-"
provider-1-1       | Computed result for input_data=3299 in 0.000008 seconds. Result modulo 3^60: 678398629988645449450387073
load_balancer_p-1  | 172.18.0.12 - - [05/Jan/2025:11:15:40 +0000] "POST /compute HTTP/1.1" 200 97 "-" "Go-http-client/1.1" "-"
consumer-1-1       | Processed request for input_data=3299 in 0.003442 seconds (Request to provider: 0.003421 seconds)
load_balancer_c-1  | 172.18.0.1 - - [05/Jan/2025:11:15:40 +0000] "GET /create_task?input_data=3299 HTTP/1.1" 200 210 "-" "PostmanRuntime/7.43.0" "-"
load_balancer_p-1  | 172.18.0.1 - - [05/Jan/2025:11:18:16 +0000] "POST /calculate HTTP/1.1" 404 19 "-" "PostmanRuntime/7.43.0" "-"
provider-2-1       | Computed result for input_data=3299 in 0.000008 seconds. Result modulo 3^60: 678398629988645449450387073
load_balancer_p-1  | 172.18.0.1 - - [05/Jan/2025:11:18:28 +0000] "POST /compute HTTP/1.1" 200 97 "-" "PostmanRuntime/7.43.0" "-"
```

Рисунок 8 - Логи сервісів.