ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

Методи інтеграції та узгодження мікросервісів в хмарній архітектурі

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Кафедра «Електронних обчислювальних машин»

Виконав: ст. гр. СПм-22-3 Дюльгер В. Д.
Керівник: ст. викл. Сорокін А.Р.

2024

Актуальність проблеми

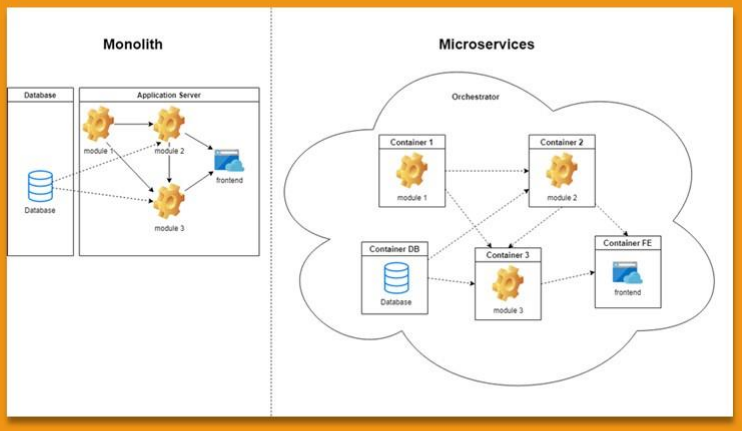
- Інтеграція компонентів
- Узгодження даних
- Оркестрація та координація
- Хмарні обчислення
- Зростання популярності криптовалют
- Складність та обсяг даних
- Необхідність обробки даних у реальному часі
- Зростання кількості інвесторів

Мета та задачі

- Аналіз мікросервісної архітектури
- Дослідження хмарних обчислень
- Вивчення методів інтеграції
- Аналіз існуючих рішень
- Розробка архітектури платформи
- Реалізація платформи
- Інтеграція з хмарними сервісами

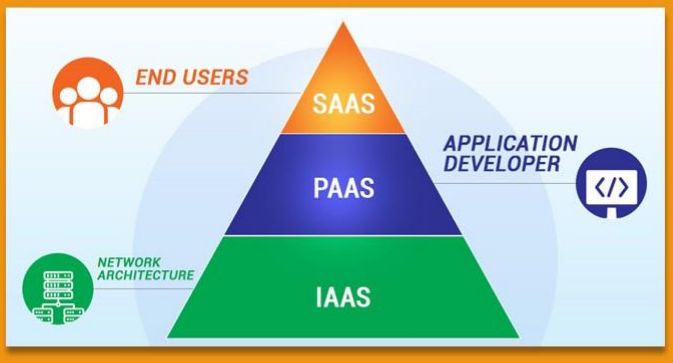
Розуміння мікросервісів та хмарної парадигми

- Невеликі, незалежні служби
- Чітко визначені інтерфейси
- Масштабованість
- Гнучкість
- Економічність.



Роль мікросервісів у хмарі

- Модульність та незалежність
- Масштабованість
- Відмовостійкість
- Швидкість розгортання та оновлення

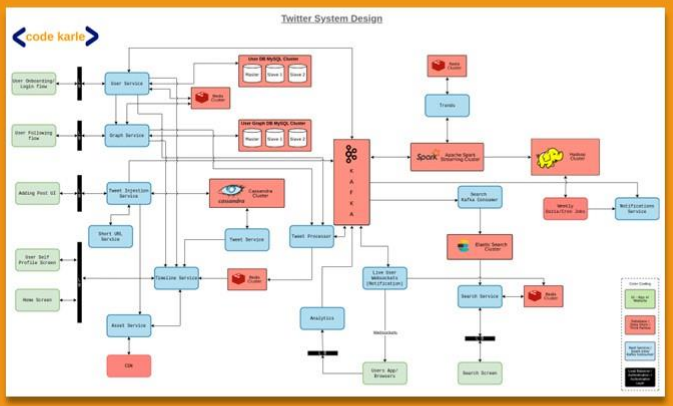


Дослідження поточних тенденцій

- Зростання популярності мікросервісів
- Контейнеризація
- Оркестрація контейнерів
- DevOps та CI/CD
- Серверлесс технології
- Безпека та управління даними

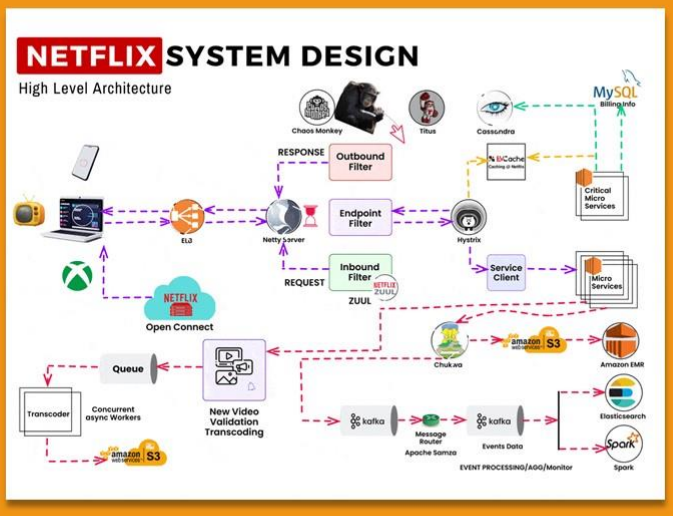
Приклад використання мікросервісів - Twitter

- Проблеми з масштабуванням
- Низька стійкість до збоїванням
- Ускладнене розгортання



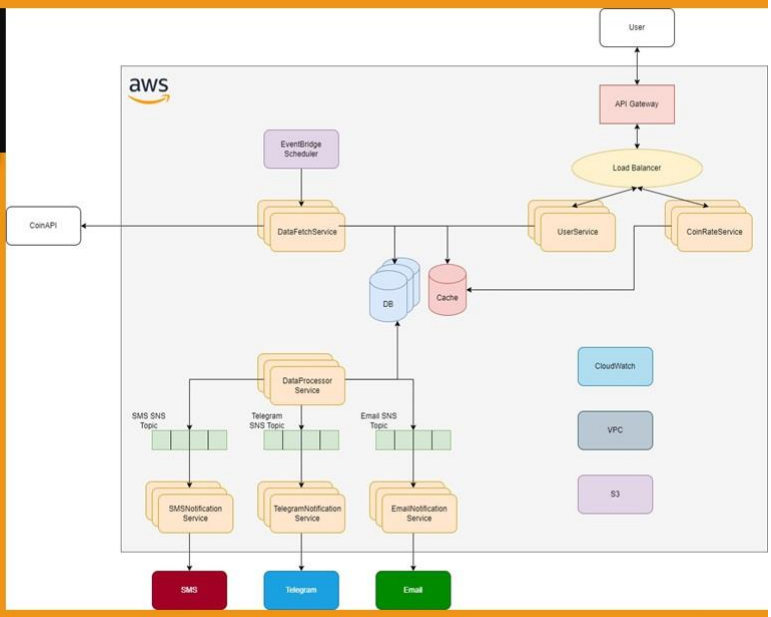
Приклад використання мікросервісів - Netflix

- Масштабованість
- Швидкість розробки та впровадження
- Стійкість до збоїв
- Потокове передавання відео
- Рекомендаційна система
- Обробка платежів



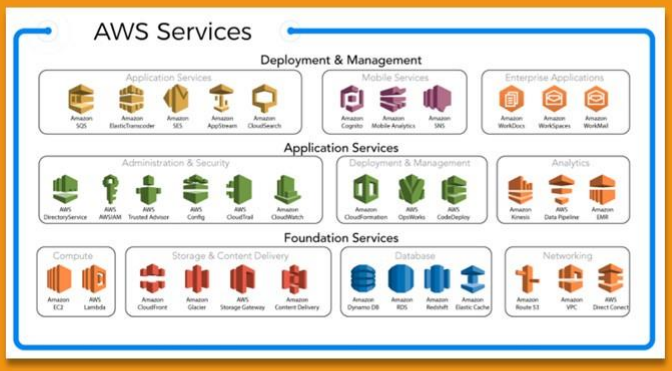
Архітектура та дизайн платформи

- Мікросервісна архітектура
- Використання AWS Lambda
- Зберігання даних в Amazon S3 та PostgreSQL
- Використання Amazon SNS
- Моніторинг та відстеження
- Масштабованість та надійність



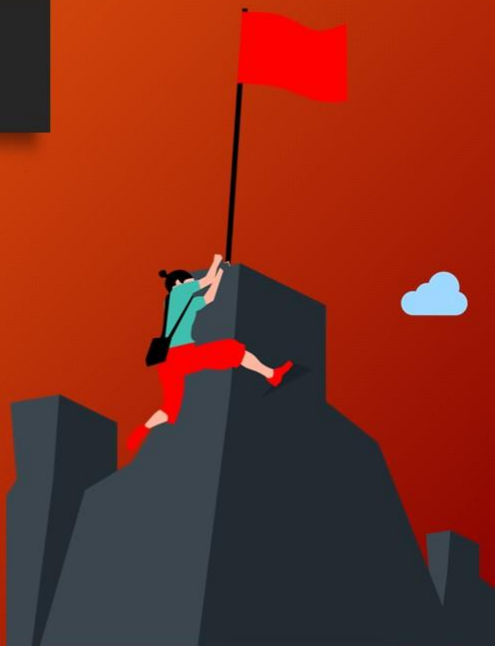
Реалізація системи

- Збір даних
- Обробка даних
- Зберігання та управління даними
- Надсилання повідомлень
- Моніторинг та відстеження
- Безпека та надійність



Виклики та перспективи

- Масштабованість
- Інтеграція компонентів
- Безпека даних
- Перспективи розвитку



Висновки

- Розуміння мікросервісної архітектури
- Інтеграція мікросервісів
- Узгодження даних
- Оркестрація та координація
- Практичне впровадження
- Оптимізація та моніторинг

ДОДАТОК Б

Код проекту

```
package com.lambda

import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.RequestHandler
import com.amazonaws.services.lambda.runtime.events.SNSEvent
import com.fasterxml.jackson.module.kotlin.jacksonObjectMapper
import com.fasterxml.jackson.module.kotlin.readValue
import okhttp3.MediaType.Companion.toMediaType
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.RequestBody.Companion.toRequestBody

class TelegramNotificationHandler : RequestHandler<SNSEvent, Unit> {
    private val objectMapper = jacksonObjectMapper()
    private val client = OkHttpClient()
    private val telegramBotToken = "5396849544:AAFjSoBzBZybmTQiUX-
r416F2I8pkmOlpN0"

    override fun handleRequest(event: SNSEvent, context: Context) {
        event.records.forEach { record ->
            val message = record.sns.message
            val notification: UserNotification =
objectMapper.readValue(message)
            sendTelegramMessage(notification)
        }
    }

    private fun sendTelegramMessage(notification: UserNotification) {
        val telegramApiUrl =
"https://api.telegram.org/bot$telegramBotToken/sendMessage"
        val chatId = notification.notificationId
        val text = buildMessageText(notification)

        val requestBody = """
            {
                "chat_id": "$chatId",
                "text": "$text"
            }
        """.trimIndent().toRequestBody("application/json; charset=utf-
8".toMediaType())

        val request = Request.Builder()
            .url(telegramApiUrl)
            .post(requestBody)
            .build()

        client.newCall(request).execute().use { response ->
            if (!response.isSuccessful) {
                throw RuntimeException("Failed to send message:
${response.body?.string()}")
            }
        }
    }

    private fun buildMessageText(notification: UserNotification): String {
        val assetsInfo = notification.assets.joinToString(separator =
"\n") {
            "Asset: ${it.assetIdBase}/${it.assetIdQuote}, Rate:
${it.rate}"
```

```
            }
            return "Hello ${notification.userName},\n\nHere are your tracked
assets:\n$assetsInfo"
        }
    }

    data class UserNotification(
        val userId: Int,
        val userName: String,
        val notificationId: String,
        val channel: String,
        val assets: List<Asset>
    )

    data class Asset(
        val assetIdBase: String,
        val assetIdQuote: String,
        val rate: Double
    )


    package com.lambda

    import com.amazonaws.services.lambda.runtime.Context
    import com.amazonaws.services.lambda.runtime.RequestHandler
    import
com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent
    import
com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent
    import com.amazonaws.services.sns.AmazonSNS
    import com.amazonaws.services.sns.AmazonSNSClientBuilder
    import com.fasterxml.jackson.module.kotlin.jacksonObjectMapper
    import java.sql.Connection
    import java.sql.DriverManager

    class DataProcessorHandler : RequestHandler<APIGatewayProxyRequestEvent,
APIGatewayProxyResponseEvent> {
        private val dbUrl = "jdbc:postgresql://postgres.cl0es064muyk.eu-north-
1.rds.amazonaws.com:5432/postgres"
        private val dbUser = "postgres"
        private val dbPassword = "postgres"
        private val snsClient: AmazonSNS =
AmazonSNSClientBuilder.defaultClient()
        private val objectMapper = jacksonObjectMapper()

        private val snsTopics = mapOf(
            "telegram" to "arn:aws:sns:eu-north-1:381491857026:telegram-
topic",
            "sms" to "arn:aws:sns:eu-north-1:381491857026:sms-topic",
            "email" to "arn:aws:sns:eu-north-1:381491857026:email-topic"
        )

        override fun handleRequest(input: APIGatewayProxyRequestEvent,
context: Context): APIGatewayProxyResponseEvent {
            val connection = getConnection()
            val notifications = queryUserNotifications(connection)
            connection.close()

            notifications.forEach { notification ->
                val message = objectMapper.writeValueAsString(notification)
                snsClient.publish(snsTopics[notification.channel], message)
            }

            return APIGatewayProxyResponseEvent().apply {
```

```
                statusCode = 200
                body = "Notifications published successfully"
            }
        }

        private fun getConnection(): Connection {
            return DriverManager.getConnection(dbUrl, dbUser, dbPassword)
        }

        private fun queryUserNotifications(connection: Connection):
List<UserNotification> {
            val query = """
                SELECT u.id AS user_id, u.name AS user_name, n.channel AS
notification_channel,
                       COALESCE(u.telegram_id, u.phone_number, u.email) AS
notification_id,
                       ta.asset_id_base, ta.asset_id_quote, er.rate
                FROM users u
                JOIN notifications n ON u.id = n.user_id
                JOIN tracked_assets ta ON u.id = ta.user_id
                JOIN exchange_rates er ON ta.asset_id_base = er.asset_id_base
AND ta.asset_id_quote = er.asset_id_quote
                ORDER BY u.id, n.channel
            """
            val statement = connection.createStatement()
            val resultSet = statement.executeQuery(query)
            val notifications = mutableListOf<UserNotification>()

            while (resultSet.next()) {
                val userId = resultSet.getInt("user_id")
                val userName = resultSet.getString("user_name")
                val channel = resultSet.getString("notification_channel")
                val notificationId = resultSet.getString("notification_id")
                val assetIdBase = resultSet.getString("asset_id_base")
                val assetIdQuote = resultSet.getString("asset_id_quote")
                val rate = resultSet.getDouble("rate")

                val existingNotification = notifications.find { it.userId ==
userId && it.channel == channel }

                if (existingNotification != null) {
                    existingNotification.assets.add(Asset(assetIdBase,
assetIdQuote, rate))
                } else {
                    val assets = mutableListOf(Asset(assetIdBase,
assetIdQuote, rate))
                    notifications.add(UserNotification(userId, userName,
notificationId, channel, assets))
                }
            }

            resultSet.close()
            statement.close()

            return notifications
        }
    }

    data class UserNotification(
        val userId: Int,
        val userName: String,
        val notificationId: String,
        val channel: String,
        val assets: MutableList<Asset>
```

```
)

data class Asset(
    val assetIdBase: String,
    val assetIdQuote: String,
    val rate: Double
)
package com.lambda
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.RequestHandler
import
com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent
import
com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent
import com.fasterxml.jackson.annotation.JsonCreator
import com.fasterxml.jackson.annotation.JsonProperty
import com.fasterxml.jackson.databind.ObjectMapper
import java.sql.Connection
import java.sql.DriverManager
import java.sql.PreparedStatement
import java.sql.SQLException

data class CreateUserRequest @JsonCreator constructor(
    @JsonProperty("name") val name: String,
    @JsonProperty("telegramId") val telegramId: String?,
    @JsonProperty("email") val email: String?,
    @JsonProperty("phoneNumber") val phoneNumber: String?,
    @JsonProperty("notifications") val notifications: List<String>?,
    @JsonProperty("trackedAssets") val trackedAssets: List<TrackedAsset>?
)

data class TrackedAsset @JsonCreator constructor(
    @JsonProperty("assetIdBase") val assetIdBase: String,
    @JsonProperty("assetIdQuote") val assetIdQuote: String
)
data class CreateUserResponse(val userId: Int?, val error: String?)
class   CreateUserHandler   :   RequestHandler<APIGatewayProxyRequestEvent,
APIGatewayProxyResponseEvent> {
    private val dbUrl = "jdbc:postgresql://postgres.cl0es064muyk.eu-north-
1.rds.amazonaws.com:5432/postgres"
    private val dbUser = "postgres"
    private val dbPassword = "postgres"
    override    fun    handleRequest(input:    APIGatewayProxyRequestEvent,
context: Context): APIGatewayProxyResponseEvent {
        val requestBody = input.body
        val objectMapper = ObjectMapper()
        val createUserRequest: CreateUserRequest? = try {
            objectMapper.readValue(requestBody,
CreateUserRequest::class.java)
        } catch (e: Exception) {
            context.logger.log("Error parsing input JSON: ${e.message}")
            return APIGatewayProxyResponseEvent().apply {
                statusCode = 400
                body = "Error parsing input JSON"
            }
        }
        return createUserRequest?.let { request ->
            val response = saveUserToDatabase(request)
            APIGatewayProxyResponseEvent().apply {
                statusCode = if (response.userId != null) 200 else 500
                body = objectMapper.writeValueAsString(response)
            }
        } ?: APIGatewayProxyResponseEvent().apply {
            statusCode = 400
```

```
                body = "Invalid input JSON format"
            }
        }
        private fun saveUserToDatabase(request: CreateUserRequest):
CreateUserResponse {
            var connection: Connection? = null
            var preparedStatement: PreparedStatement? = null
            return try {
                connection = DriverManager.getConnection(dbUrl, dbUser,
dbPassword)

                // Create users table if it does not exist
                val createUserTableSQL = """
                    CREATE TABLE IF NOT EXISTS users (
                        id SERIAL PRIMARY KEY,
                        name VARCHAR(255) NOT NULL,
                        telegram_id VARCHAR(255),
                        email VARCHAR(255),
                        phone_number VARCHAR(255)
                    )
                """
                connection.createStatement().execute(createUserTableSQL)

                // Create notifications table if it does not exist
                val createNotificationsTableSQL = """
                    CREATE TABLE IF NOT EXISTS notifications (
                        user_id INT REFERENCES users(id) ON DELETE CASCADE,
                        channel VARCHAR(255),
                        PRIMARY KEY (user_id, channel)
                    )
                """

connection.createStatement().execute(createNotificationsTableSQL)
                // Create tracked_assets table if it does not exist
                val createTrackedAssetsTableSQL = """
                    CREATE TABLE IF NOT EXISTS tracked_assets (
                        user_id INT REFERENCES users(id) ON DELETE CASCADE,
                        asset_id_base VARCHAR(255),
                        asset_id_quote VARCHAR(255),
                        PRIMARY KEY (user_id, asset_id_base, asset_id_quote),
                        FOREIGN KEY (asset_id_base, asset_id_quote) REFERENCES
exchange_rates(asset_id_base, asset_id_quote)
                    )
                """

connection.createStatement().execute(createTrackedAssetsTableSQL)

                // Insert user
                val insertUserSQL = """
                    INSERT INTO users (name, telegram_id, email, phone_number)
                    VALUES (?, ?, ?, ?) RETURNING id
                """
                preparedStatement = connection.prepareStatement(insertUserSQL)
                preparedStatement.setString(1, request.name)
                preparedStatement.setString(2, request.telegramId)
                preparedStatement.setString(3, request.email)
                preparedStatement.setString(4, request.phoneNumber)
                val resultSet = preparedStatement.executeQuery()
                if (resultSet.next()) {
                    val userId = resultSet.getInt(1)
                    // Insert notifications
                    request.notifications?.let { notifications ->
                        val insertNotificationSQL = """
                            INSERT INTO notifications (user_id, channel)
```

```
                            VALUES (?, ?)
                        """
                        val               notificationStatement               =
connection.prepareStatement(insertNotificationSQL)
                        for (channel in notifications) {
                            notificationStatement.setInt(1, userId)
                            notificationStatement.setString(2, channel)
                            notificationStatement.addBatch()
                        }
                        notificationStatement.executeBatch()
                        notificationStatement.close()
                    }
                    // Insert tracked assets
                    request.trackedAssets?.let { assets ->
                        val insertAssetSQL = """
                            INSERT      INTO      tracked_assets      (user_id,
asset_id_base, asset_id_quote)
                            VALUES (?, ?, ?)
                        """
                        val                   assetStatement                   =
connection.prepareStatement(insertAssetSQL)
                        for (asset in assets) {
                            assetStatement.setInt(1, userId)
                            assetStatement.setString(2, asset.assetIdBase)
                            assetStatement.setString(3, asset.assetIdQuote)
                            assetStatement.addBatch()
                        }
                        assetStatement.executeBatch()
                        assetStatement.close()
                    }
                    CreateUserResponse(userId, null)
                } else {
                    CreateUserResponse(null, "User creation failed")
                }
            } catch (e: SQLException) {
                CreateUserResponse(null, "An error occurred: ${e.message}")
            } finally {
                preparedStatement?.close()
                connection?.close()
            }
        }
    }
    package com.lambda
    import com.amazonaws.services.lambda.runtime.Context
    import com.amazonaws.services.lambda.runtime.RequestHandler
    import
com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent
    import
com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent
    import com.lambda.data.ExchangeRateResponse
    import io.ktor.client.*
    import io.ktor.client.call.*
    import io.ktor.client.plugins.contentnegotiation.*
    import io.ktor.client.request.*
    import io.ktor.serialization.kotlinx.json.*
    import kotlinx.coroutines.runBlocking
    import kotlinx.serialization.json.Json
    import org.slf4j.Logger
    import org.slf4j.LoggerFactory
    import java.sql.Connection
    import java.sql.DriverManager
    import java.sql.PreparedStatement
    import java.sql.SQLException
    class   DataFetchHandler   :   RequestHandler<APIGatewayProxyRequestEvent,
```

```
APIGatewayProxyResponseEvent> {
          lateinit var client: HttpClient
          private val dbUrl = "jdbc:postgresql://postgres.cl0es064muyk.eu-north-
1.rds.amazonaws.com:5432/postgres"
          private val dbUser = "postgres"
          private val dbPassword = "postgres"
          private           val           logger:           Logger           =
LoggerFactory.getLogger(DataFetchHandler::class.java)
          override fun handleRequest(
              input: APIGatewayProxyRequestEvent?,
              context: Context?
          ): APIGatewayProxyResponseEvent {
              var  response:  ExchangeRateResponse  =  ExchangeRateResponse("",
emptyList())
              client = HttpClient {
                  install(ContentNegotiation) {
                      json(Json {
                          ignoreUnknownKeys = true
                      })
                  }
              }
              runBlocking {
                  try {
                      response                                                =
client.get("https://rest.coinapi.io/v1/exchangerate/USD") {
                          header("X-CoinAPI-Key",      "93D64FED-0C44-4F0F-9499-
5D21B046E108")
                          header("Accept", "text/plain")
                      }.body()
                  } catch (e: Exception) {
                      logger.error("Error: ${e.message}")
                  } finally {
                      client.close()
                  }
              }
              saveToDatabase(response)
              return APIGatewayProxyResponseEvent().apply {
                  body = response.toString()
              }
          }

          private fun saveToDatabase(response: ExchangeRateResponse) {
              var connection: Connection? = null
              var preparedStatement: PreparedStatement? = null

              try {
                  connection    =    DriverManager.getConnection(dbUrl,    dbUser,
dbPassword)

                  // Create table if it does not exist
                  val createTableSQL = """
                  CREATE TABLE IF NOT EXISTS exchange_rates (
                      asset_id_base VARCHAR(255),
                      asset_id_quote VARCHAR(255),
                      rate DOUBLE PRECISION,
                      PRIMARY KEY (asset_id_base, asset_id_quote)
                  )
              """
                  connection.createStatement().execute(createTableSQL)
                  val  insertSQL  =  "INSERT  INTO  exchange_rates  (asset_id_base,
asset_id_quote, rate) VALUES (?, ?, ?) " +
                          "ON CONFLICT (asset_id_base, asset_id_quote) DO UPDATE
SET rate = EXCLUDED.rate"
                  preparedStatement = connection.prepareStatement(insertSQL)
```

```
                for (rate in response.rates) {
                    preparedStatement.setString(1, response.assetIdBase)
                    preparedStatement.setString(2, rate.assetIdQuote)
                    preparedStatement.setDouble(3, rate.rate)
                    preparedStatement.addBatch()
                }

                preparedStatement.executeBatch()

            } catch (e: SQLException) {
                logger.error("Error inserting data into the database: ${e.message}")
            } finally {
                preparedStatement?.close()
                connection?.close()
            }
        }
    }


    @Serializable
    data class ExchangeRateResponse(
        @SerialName("asset_id_base")
        val assetIdBase: String,
        val rates: List<Rate>
    )

    @Serializable
    data class Rate(
        val time: String,
        @SerialName("asset_id_quote")
        val assetIdQuote: String,
        val rate: Double
    )
```