



Neo4j NoSQL Graph Database

เสนอ

ผู้ช่วยศาสตราจารย์ ดร.วราภรณ์ วิyananท์

โดย

SELECT \* FROM ใจເຮືອ

กลุ่มผู้เรียน B01

นายจิรภูริช อุคงคາ 66102010233

นายญาณภัทร ปานเกشم 66102010236

นายรักษ์ศานต์ จันทร์ 66102010244

โครงการนี้เป็นส่วนหนึ่งของการศึกษารายวิชา คพ242 ระบบฐานข้อมูล

มหาวิทยาลัยศรีนครินทรวิโรฒ

ภาคการศึกษาที่ 2 ปีการศึกษา 2567

## สารบัญ

เนื้อหา	หน้า
<b>สารบัญ</b>	<b>ก</b>
ที่มาและความสำคัญ	1
วัตถุประสงค์	1
ขั้นตอนการติดตั้งระบบ Neo4j Database ลงบนอุปกรณ์ของคุณและวิธีการใช้งานเบื้องต้น	2
รายละเอียด CRUD Operation ของ Neo4j NoSQL Graph Database	15
คำอธิบายเกี่ยวกับ Use Case	25
สรุปผลการดำเนินงาน	35
<b>บรรณานุกรม</b>	<b>ข</b>

## ที่มาและความสำคัญ

ในปัจจุบัน การแนะนำสินค้าให้ตรงกับความต้องการของผู้ใช้งานแต่ละคนถือเป็นหนึ่งในองค์ประกอบสำคัญของเว็บไซต์เชิงพาณิชย์ เพื่อเพิ่มโอกาสในการตัดสินใจซื้อและยกระดับประสบการณ์ของผู้ใช้งาน โครงการนี้จึงมีวัตถุประสงค์เพื่อพัฒนาระบบแนะนำสินค้าที่สามารถปรับเปลี่ยนได้ตามพฤติกรรมของผู้ใช้ โดยเลือกใช้ Neo4j ซึ่งเป็นฐานข้อมูลแบบ NoSQL Graph Database ที่มีความเหมาะสมในการจัดการข้อมูลที่มีความเชื่อมโยงกันอย่างซับซ้อน

การนำโครงสร้างข้อมูลแบบ Node และ Relationship (Edge) มาใช้ ช่วยให้ระบบสามารถเรียกค้นข้อมูลได้อย่างยืดหยุ่นและมีประสิทธิภาพ ตอบสนองความต้องการของเว็บไซต์ที่มีการเชื่อมโยงข้อมูลหลายมิติได้เป็นอย่างดี

โครงการนี้จึงนับเป็นแนวทางหนึ่งที่แสดงให้เห็นถึงศักยภาพของการใช้ฐานข้อมูลกราฟในการพัฒนาระบบแนะนำสินค้า และสามารถนำไปต่อยอดหรือพัฒนาเพิ่มเติมได้ในอนาคต ทั้งยังเป็นประโยชน์ต่อผู้ที่สนใจศึกษาหรือประยุกต์ใช้เทคโนโลยีด้านฐานข้อมูลกราฟในงานพัฒนาเว็บไซต์และระบบเชิงพาณิชย์ต่อไป

## วัตถุประสงค์

- เพื่อพัฒนาระบบแนะนำสินค้าบนเว็บไซต์ที่สามารถปรับเปลี่ยนคำแนะนำให้เหมาะสมกับพฤติกรรมและความสนใจของผู้ใช้งานแต่ละบุคคล
- เพื่อศึกษาและประยุกต์ใช้ฐานข้อมูลแบบกราฟ (Graph Database) โดยใช้ Neo4j ในการจัดเก็บและบริหารข้อมูลที่มีความสัมพันธ์เชื่อมโยงระหว่างกัน

## ขั้นตอนการติดตั้งระบบ Neo4j Database ลงบนอุปกรณ์ของคุณและวิธีการใช้งานเบื้องต้น

### 1. ติดตั้ง Docker Desktop

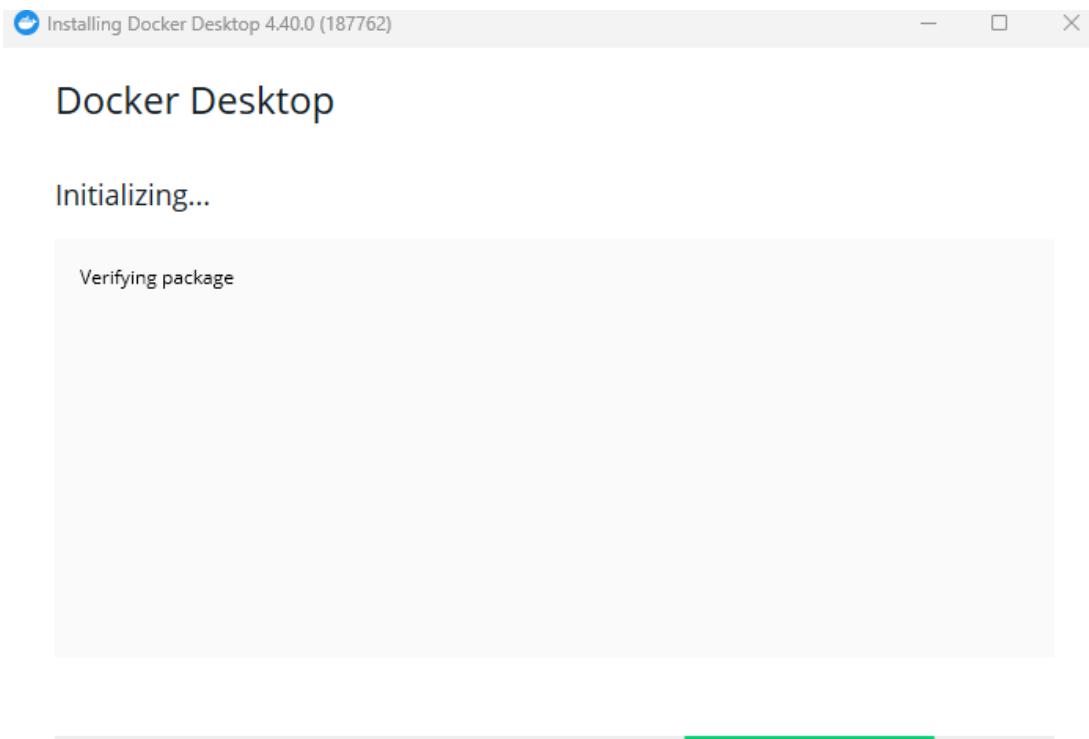
#### a. ดาวน์โหลด Docker Desktop จาก Website

<https://www.docker.com/products/docker-desktop/>

The image contains two screenshots of the Docker documentation website:

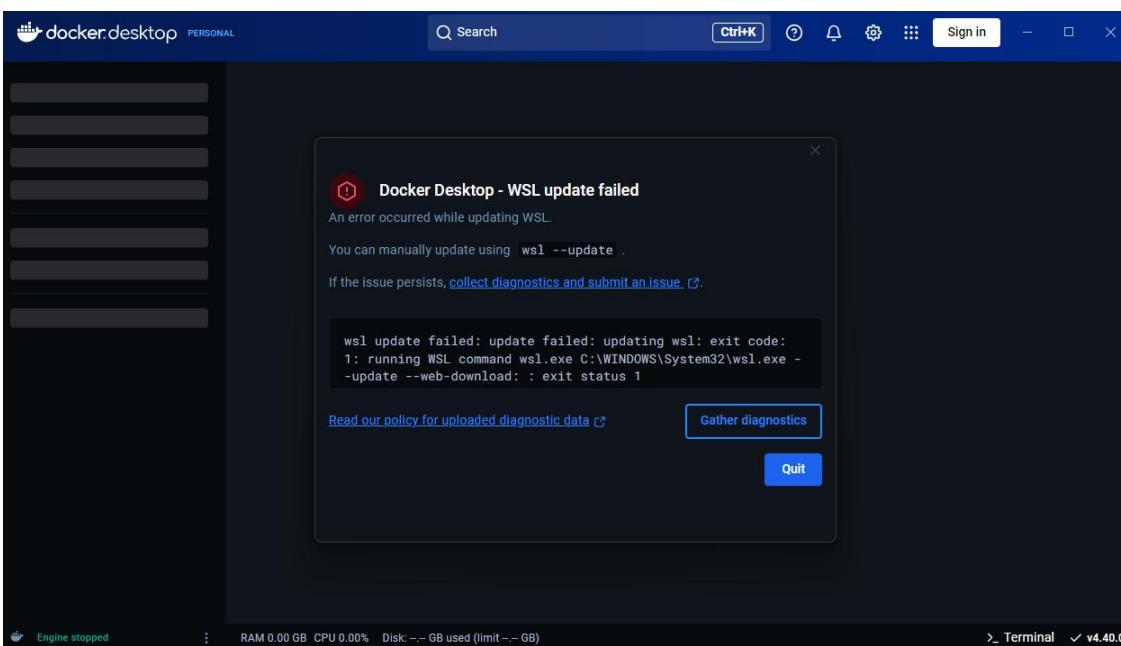
- Screenshot 1 (Top): Docker Compose Page**
  - URL: <https://docs.docker.com/compose/>
  - Section: Docker Compose
  - Content: Describes Docker Compose as a tool for defining and running multi-container applications. It includes sections on Why use Compose?, How Compose works, Quickstart, View the release notes, and Explore the Compose file reference.
  - Callout: A red box highlights the "Install Compose" section, which provides instructions on how to install Docker Compose.
- Screenshot 2 (Bottom): Install Docker Desktop on Windows Page**
  - URL: <https://docs.docker.com/desktop/windows/install/>
  - Section: Install Docker Desktop on Windows
  - Content: Provides download links, system requirements, and step-by-step installation instructions for Docker Desktop on Windows. It includes sections on Docker Desktop terms, System requirements, and a Tip about Hyper-V or WSL.
  - Callout: A red box highlights the "Docker Desktop for Windows - x86\_64" download link.

b. รันไฟล์ติดตั้ง Docker Desktop Installer.exe ขั้นตอนนี้อาจจะใช้เวลานานสักพัก



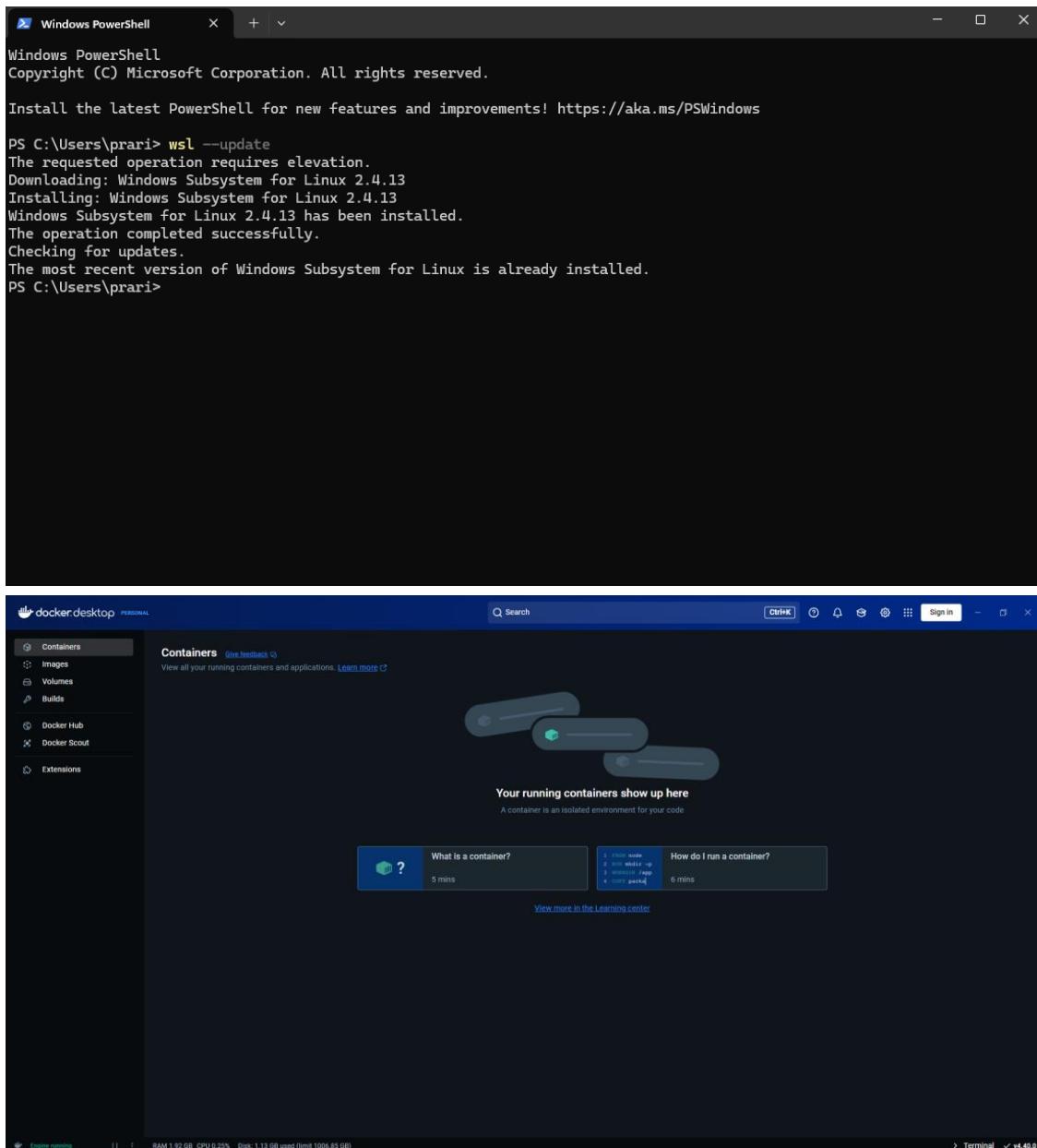
c. หากเกิดข้อผิดพลาดเมื่อเข้า Docker Desktop ครั้งแรก (หากไม่เกิดข้อผิดพลาด สามารถข้ามไปยังขั้นตอนถัดไปได้ทันที)

i. ปิดโปรแกรม



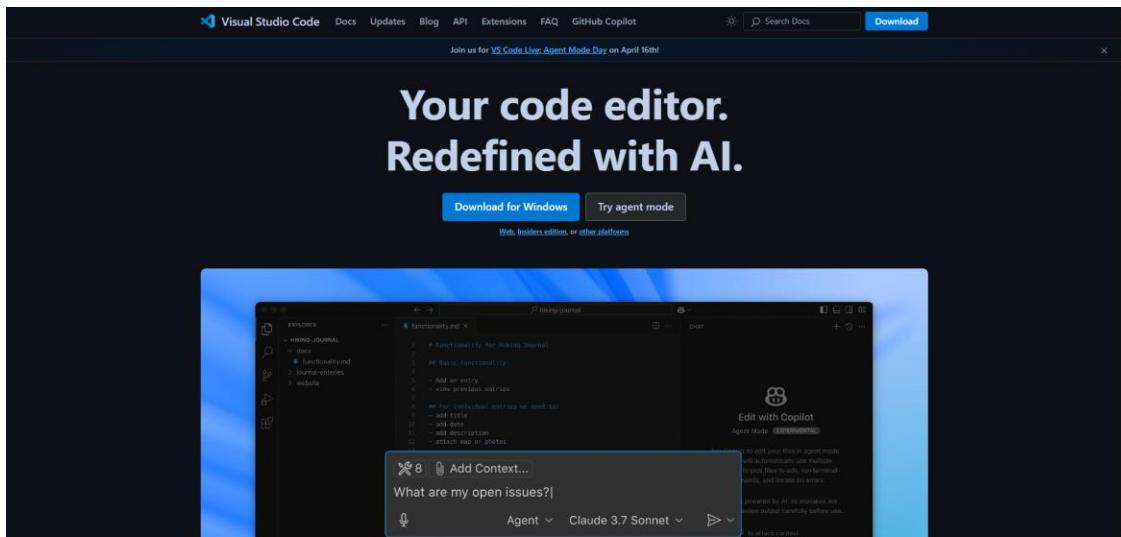
ii. เข้าโปรแกรม Terminal => Windows PowerShell และรันคำสั่ง

**wsl –update** จากนั้นเปิดโปรแกรม Docker Desktop ใหม่อีกครั้ง



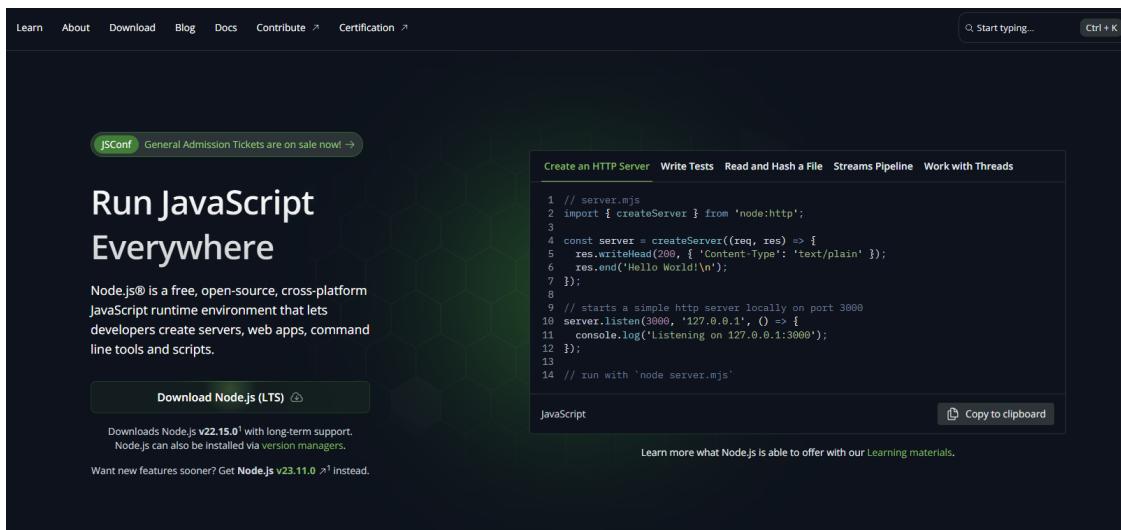
## 2. ติดตั้ง Visual Studio Code

ดาวน์โหลด Visual Studio Code จาก Website: <https://code.visualstudio.com/>



## 3. ติดตั้ง Node.js

ดาวน์โหลด Node.js จาก Website: <https://nodejs.org/en>



#### 4. ติดตั้งระบบฐานข้อมูลลงบนเครื่อง โดยใช้ Docker Desktop และ Visual Studio Code

- สร้างโปรเจค Folder ใหม่ใน Visual Studio Code
- สร้างไฟล์ชื่อ docker-compose.yml ในระดับ Root และเขียนกำหนดค่า Config ต่าง ๆ

ดังนี้

```
services:
  neo4j:
    image: neo4j:latest
    container_name: neo4j-container
    volumes:
      - ./neo4j_database/neo4j_data:/data
      - ./neo4j_database/neo4j_logs:/logs
      - ./neo4j_database/neo4j_plugins:/plugins
    environment:
      - NEO4J_AUTH=neo4j/Neo4j12345*
    ports:
      - "7474:7474"
      - "7687:7687"
    restart: always
```

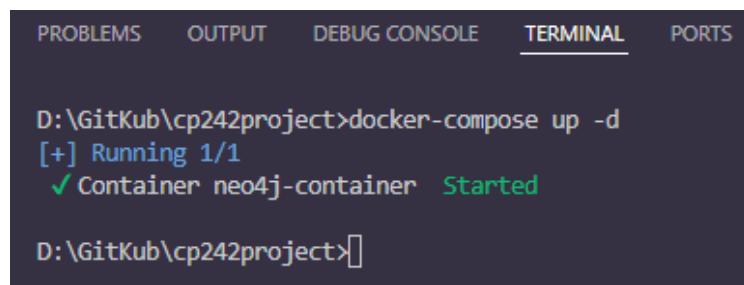
ในตัวอย่างนี้มีการตั้งให้เรียกใช้ Image Neo4j version ล่าสุด ตั้งชื่อ Container ว่า neo4j-container เลือกเก็บข้อมูลไว้ใน Folder neo4j\_database ตั้ง Username คือ neo4j และ Password คือ Neo4j12345\* และกำหนด Ports สำหรับเข้าใช้ Neo4jBrowser คือ 7474 และ Port สำหรับเชื่อมต่อ กับ Node.js คือ 7687 สามารถเปลี่ยนได้ตามความเหมาะสม

- ต้องมั่นใจว่า Path อยู่ใน Folder โปรเจคนั้น และรันคำสั่ง สามารถเลือกใช้ Command

Prompt หรือ PowerShell ได้

```
docker-compose up -d
```

ผลลัพธ์ที่ได้



```
D:\GitKub\cp242project>docker-compose up -d
[+] Running 1/1
✓ Container neo4j-container Started

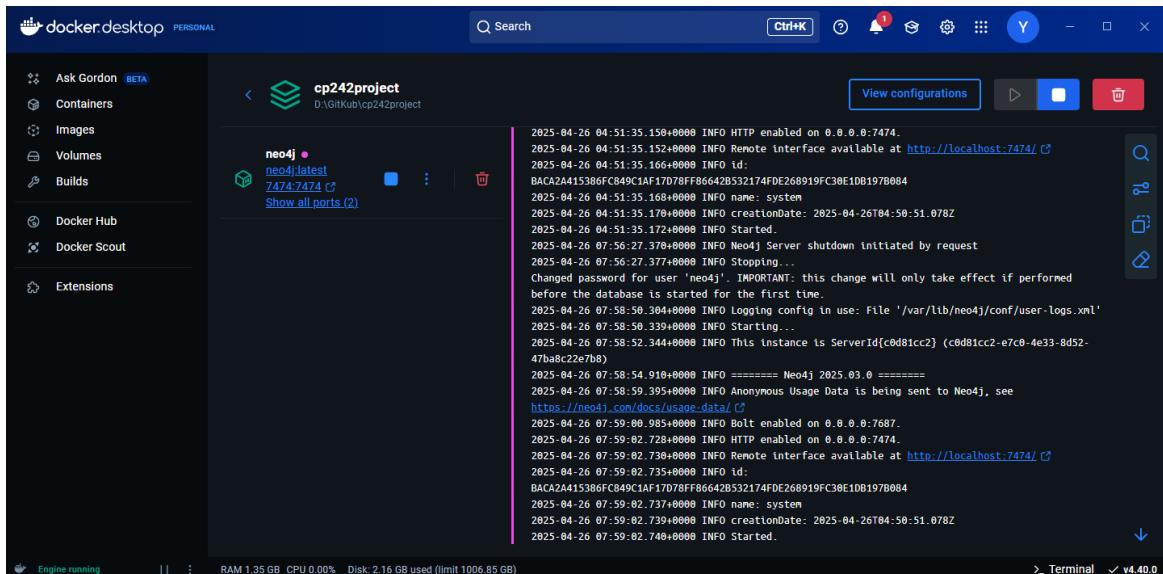
D:\GitKub\cp242project>[]
```

หมายเหตุ ในครั้งแรกที่รันจะต้องใช้เวลาในการโหลด Image สักพัก แต่หากรันครั้งที่สอง จะรันได้เร็วขึ้น

#### d. คำสั่งพื้นฐานสำหรับการจัดการ Docker ใน Terminal

```
docker-compose up -d // Start all containers
docker-compose stop // Stop all containers
docker ps // Show all process about containers
docker-compose down // Stop and remove all containers
```

#### e. สามารถดูรายละเอียดเพิ่มเติมเกี่ยวกับ Docker ได้ที่ Docker Desktop



## 5. การเชื่อมต่อ Node.js ให้สามารถทำงานร่วมกับ Neo4j Database ที่อยู่บน Docker

### แบบเบื้องต้น

- ที่ Terminal ภายใต้ Folder Project รันคำสั่ง โดยรันทีละบรรทัด

```
npm init
npm install express neo4j-driver nodemon body-parser
```

- สร้างไฟล์ชื่อ server.js และเขียนโค้ดด้านล่างนี้

```
const express = require('express');
const bodyParser = require('body-parser');
const neo4j = require('neo4j-driver');

const app = express();
app.use(bodyParser.json());

const driver = neo4j.driver(
  'bolt://localhost:your_port',
  neo4j.auth.basic('your_username', 'your_password')
);
const session = driver.session();

app.get('/', async (req, res) => {
  try {
    await session.run('RETURN 1');
    res.send('Connected to Neo4j successfully!');
  } catch (error) {
    res.status(500).send(`Failed to connect to Neo4j: ${error.message}`);
  }
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on
http://localhost:${PORT}`);
});
```

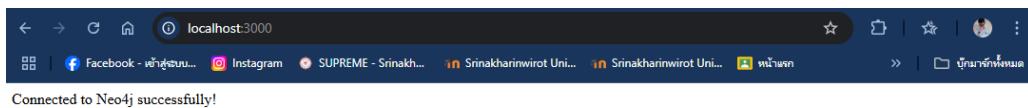
หมายเหตุ Port, Username และ Password ให้เลือกใช้ตามไฟล์ docker-compose.yml ที่

ได้กำหนดค่าเอาไว้

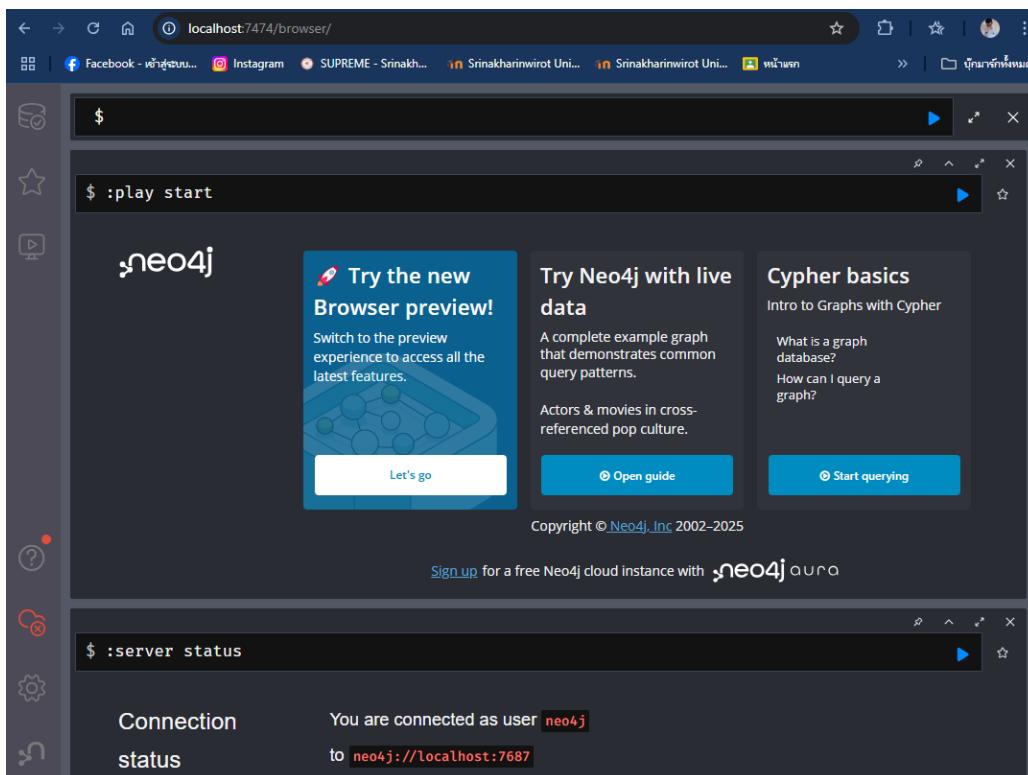
เมื่อกำหนดค่าเรียบร้อยแล้วให้รันนี้ใน Terminal

```
nodemon server.js
```

## ผลลัพธ์



ไปที่ localhost:7474/browser จะเจอกับหน้า Interface สำหรับ Neo4j



หมายเหตุ ในครั้งแรกต้องล็อคอินด้วย Username และ Password ที่ได้ตั้งเอาไว้ และต้องเปิดใช้งาน Docker แล้วเท่านั้น

## 6. ทดสอบคำสั่ง CRUD เป็งต้นด้วย Neo4j Driver จาก Node.js และ Postman

- ติดตั้ง Postman จาก Website: <https://www.postman.com/>
- ที่ Postman ให้เลือกใช้ URL: localhost:3000/ ที่ได้กำหนดไว้ใน Express.js
- ทดสอบระบบ Create
  - เพิ่มโค้ดส่วนนี้เข้าไปใน server.js

```
app.post('/nodes', async (req, res) => {
  const { label, properties } = req.body;
  try {
    const result = await session.run(
      `CREATE (n:${label} ${props}) RETURN n`,
      { props: properties }
    );
    res.json(result.records[0].get('n').properties);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

- ไปที่ Postman เลือก POST ไปที่ localhost:3000/nodes และทดสอบ
- เลือก Body => Raw => JSON

The screenshot shows the Postman interface with a POST request to `http://localhost:3000/nodes`. The 'Body' tab is selected, and the 'JSON' option is highlighted with a red box. The request body is a JSON object:

```
{
  "label": "Person",
  "properties": {
    "id": "u1",
    "name": "Alice",
    "age": 25
  }
}
```

### iii. ผลลัพธ์

The screenshot shows the Neo4j browser interface. On the left, there's a sidebar titled 'Database Information' with sections for 'Use database' (set to 'neo4j'), 'Node labels' (with 'Person' highlighted in red), 'Relationship types' (with 'BOUGHT' and 'FROM'), and 'Property keys'. On the right, a central panel displays a graph with a single red node labeled 'Alice'. To the right of the graph, a table shows the node's properties:

Property	Value
<elementId>	4:a2cc2abe-cd48-43b5-9713-c12a55b4c5cf:23
<id>	23
age	25.0
id	u1
name	Alice

### d. ทดสอบระบบ READ

i. เพิ่มโค้ดส่วนนี้เข้าไปใน server.js

```
app.get('/nodes/:label', async (req, res) => {
  const label = req.params.label;
  try {
    const result = await session.run(`MATCH (n:${label}) RETURN n`);
    const nodes = result.records.map(record =>
      record.get('n').properties);
    res.json(nodes);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

ii. ไปที่ Postman เลือก GET ไปที่ localhost:3000/nodes/:label และ

ทดสอบ ใส่ชื่อ label ที่ต้องการ ในตัวอย่างนี้คือ Person

### iii. ผลลัพธ์

The screenshot shows a Postman request to `http://localhost:3000/nodes/Person`. The request method is `GET`. The `Body` tab contains the following JSON:

```

1
2   "label": "Person",
3   "properties": {
4     "id": "u1",
5     "name": "Alice",
6     "age": 25
7   }
8

```

The response tab shows a `200 OK` status with `195 ms` duration and `272 B` size. The response body is identical to the request body.

### e. ทดสอบระบบ UPDATE

#### i. เพิ่มโค้ดส่วนนี้เข้าไปใน server.js

```

app.put('/nodes/:label/:id', async (req, res) => {
  const { label, id } = req.params;
  const { properties } = req.body;
  try {
    const result = await session.run(
      `MATCH (n:${label} {id: $id})
        SET n += $props
        RETURN n`,
      { id, props: properties }
    );
    res.json(result.records[0].get('n').properties);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

```

- ii. ไปที่ Postman เลือก PUT ไปที่ localhost:3000/nodes/Person/:id และ  
ในตัวอย่างนี้คือ nodes/Person/u1

iii. ผลลัพธ์

f. ทดสอบระบบ Delete

- i. เพิ่มโค้ดส่วนนี้เข้าไปใน server.js

```
app.delete('/nodes/:label/:id', async (req, res) => {
  const { label, id } = req.params;
  try {
    await session.run(
      `MATCH (n:${label} {id: $id}) DETACH DELETE n`,
      { id }
    );
    res.json({ message: 'Node deleted' });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

- ii. ไปที่ Postman เลือก DELETE ไปที่ localhost:3000/nodes/Person/:id  
และ ในตัวอย่างนี้คือ nodes/Person/u1

### iii. ผลลัพธ์

The screenshot shows the Postman interface. The URL in the address bar is `http://localhost:3000/nodes/Person/u1`. The method dropdown is set to `DELETE`. The Headers tab shows `(8)`. The Body tab is selected and contains the following JSON payload:

```

1  {
2   | "properties": [ { "age": 26 } ]
3 }

```

The response tab shows the following JSON message:

```

1  {
2   | "message": "Node deleted"
3 }

```

## 7. การเรียกใช้ Cypher-Shell Neo4j ผ่าน Docker

- a. รันคำสั่งผ่าน Terminal เราสามารถใช้คำสั่ง CRUD ผ่าน Cypher-Shell ได้

```
docker exec -it <container-name> cypher-shell -u <username>
-p <password>
```

ผลลัพธ์

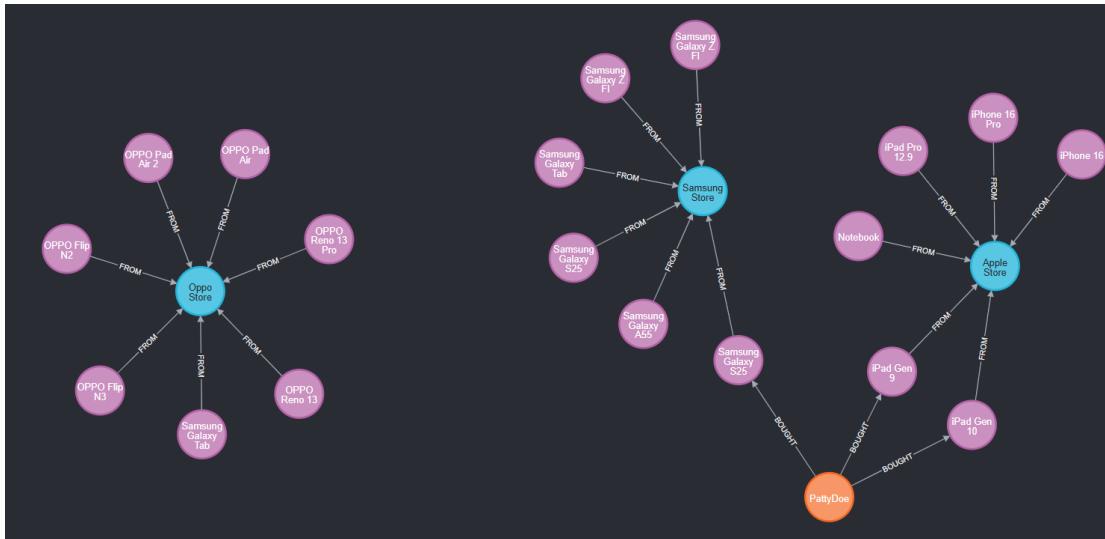
```
neo4j@neo4j:~> MATCH (a:Account) return a LIMIT 5;
+-----+
| a
+-----+
| (:Account {password: "123", role: "Admin", id: "1745643139978", email: "admin@example.com", username: "admin"}) |
| (:Account {password: "123", role: "User", id: "1745648681081", email: "patty.doe@example.com", username: "PattyDoe"}) |
+-----+
2 rows
ready to start consuming query after 44 ms, results consumed after another 1 ms
neo4j@neo4j:~> []
```

หมายเหตุ ทุกรวงที่รันคำสั่งต้องลงท้าย ; ทุกรวง โดยที่ Docker ต้องเปิดใช้งานอยู่ และคำสั่ง

ออกคือ

```
:quit
```

## รายละเอียด CRUD Operation ของ Neo4j NoSQL Graph Database



สำหรับการ Execute คำสั่ง Cypher-Query เราสามารถเลือก Execute ผ่าน Neo4j Browser ได้หรือสามารถเลือก Execute ผ่าน Cypher-Shell ใน Docker ได้เช่นกัน โดยในตัวอย่างนี้เราจะใช้ Neo4j Browser เป็นหลัก

จากการข้างต้น เราจะสังเกตเห็นว่า Graph นั้นมีสิ่งที่เรียกว่า Node และ Edge ในการเชื่อมต่อแต่ละ Node เข้าด้วยกัน โดยเราจะสาธิตกระบวนการ CRUD กับ Node และ Edge

### 1. CRUD Operation สำหรับ Node

#### a. CREATE Operation

```
CREATE (node:Label {property: <your_property>});
```

Execute ผ่าน Neo4j Browser

```
neo4j$ CREATE (p:Product {productName: 'iPhone 15', productPrice: 15000, productTags: 'SmartPhone'});
Added 1 label, created 1 node, set 3 properties, completed after 11 ms.
```

## Execute ฝ่าย Cypher-Chell ใน Docker

```
neo4j@neo4j:~$ CREATE (p:Product {productName: 'iPhone 17', productPrice: 25000, productTags: 'SmartPhone'});
0 rows
ready to start consuming query after 15 ms, results consumed after another 0 ms
Added 1 nodes, Set 3 properties, Added 1 labels
neo4j@neo4j:~$
```

### b. READ Operation

```
MATCH (node:Label) RETURN node LIMIT <number>;
```

Execute ฝ่าย Neo4j Browser

## Execute ฝ่าย Cypher-Chell ใน Docker

```
neo4j@neo4j:~$ MATCH (node:Shop) RETURN node LIMIT 25;
+-----+
| node
+-----+
| {:Shop {shopName: "Samsung Store", shopPhone: "0998881248", id: "1745643181263", shopAddress: "114 Sukhumvit 23, Bangkok 10110, Thailand."}}
| {:Shop {shopName: "Apple Store", shopPhone: "0998887777", id: "1745643212838", shopAddress: "789 Green Lane, Eco Town"}}
| {:Shop {shopName: "Oppo Store", shopPhone: "0998887755", id: "1745643228818", shopAddress: "123 Main Street, Cityville"}}
| {:Shop {shopName: "SMUShop", shopPhone: "0999999999", id: "1745648165481", shopAddress: "114 Sukhumvit 23, Bangkok 10110, Thailand."}}
+-----+
4 rows
ready to start consuming query after 43 ms, results consumed after another 1 ms
neo4j@neo4j:~$
```

### c. Update Operation

```
MATCH (node:Label {property: <your_property>})
SET node.property = <new_value>;
```

Execute ផែន Neo4j Browser

Before

The screenshot shows the Neo4j Browser interface with two main sections: a graph view and a details view.

**Graph View:** A circular diagram representing a node named "iPhone 17". The node has several outgoing edges, some of which are highlighted with icons like a lock and a gear.

**Details View:** A panel on the right showing "Node properties" for a "Product" node.

Property	Value
<elementId>	4:a2cc2abe-cd48-43b5-9713-c12a55b4c5cf:26
<id>	26
productName	iPhone 17
productPrice	25000
productTags	SmartPhone

**Code Panel:** Below the details view, there is a code panel showing the Cypher query used to update the node.

```
1 MATCH (node:Product {productName: 'iPhone 17'})
2 SET node.productPrice = 15000;
```

The output of the query is displayed below the code panel:

Set 1 property, completed after 62 ms.

After

The screenshot shows the Neo4j Browser interface. On the left, there's a sidebar with tabs: Graph (selected), Table, Text, and Code. The main area displays a circular node for 'iPhone 17'. The node has three outgoing edges: one with a power icon, one with a camera icon, and one with a gear icon. To the right of the node is a panel titled 'Node properties' for a 'Product' node. The properties listed are:

- <element> 4:a2cc2abe-cd48-
- <id> 43b5-9713-c12a55b4c5cf:26
- <id> 26
- product** iPhone 17
- Name
- productP** 15000
- rice**
- productT** SmartPhone
- ags

The properties 'productP', 'rice', and 'productT' are highlighted with a red border.

d. Delete Operation

```
MATCH (node:Label {property: <your_property>})
DELETE node;
```

Execute ผ่าน Neo4j Browser

The screenshot shows the Neo4j Browser interface after a delete operation. The main query in the top bar is:

```
neo4j$ MATCH (node:Product {productName: "iPhone 17"}) DELETE node;
```

The results pane at the bottom shows the message: "Deleted 1 node, completed after 22 ms." This message is also displayed in the status bar at the bottom of the browser window.

Execute ผ่าน Cypher-Chell ใน Docker

```
neo4j@neo4j:~$ MATCH (node:Product {productName: "iPhone 15"}) DELETE node;
0 rows
ready to start consuming query after 19 ms, results consumed after another 0 ms
Deleted 1 nodes
neo4j@neo4j:~$
```

## 2. CRUD Operation สำหรับ Edge

### a. CREATE Edge Operation

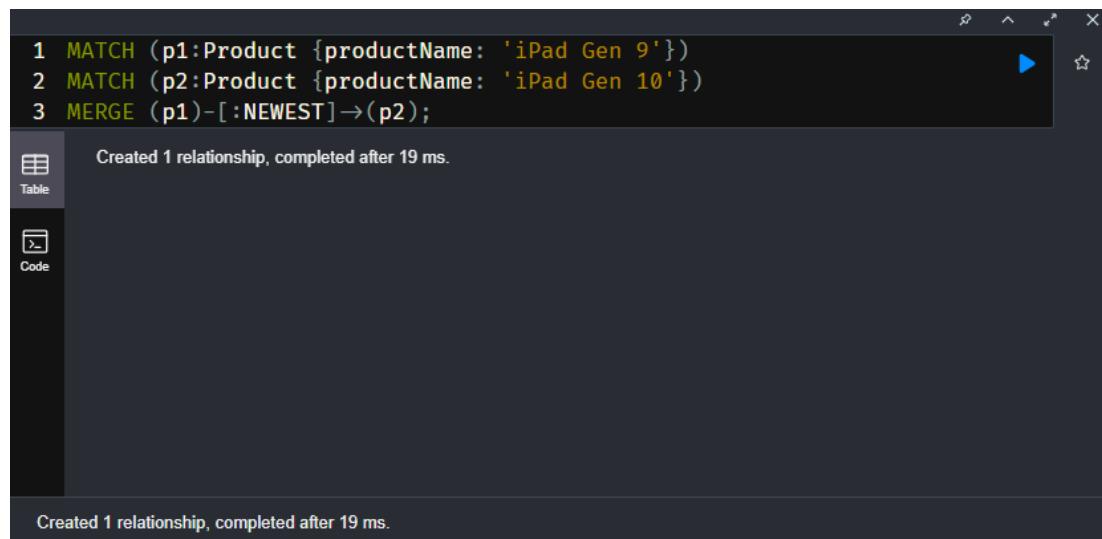
สร้าง Relationship จาก Node A ไปยัง Node B

```
MATCH (a:LabelA {propertyKey: 'valueA' })
MATCH (b:LabelB {propertyKey: 'valueB' })
MERGE (a)-[:RELATIONSHIP_TYPE]->(b);
```

สร้าง Relationship จาก Node B กลับมายัง Node A

```
MATCH (a:LabelA {propertyKey: 'valueA' })
MATCH (b:LabelB {propertyKey: 'valueB' })
MERGE (b)-[:RELATIONSHIP_TYPE]->(a);
```

Execute ผ่าน Neo4j Browser



The screenshot shows the Neo4j Browser interface with a dark theme. In the top-left code editor, there is a Cypher query:

```
1 MATCH (p1:Product {productName: 'iPad Gen 9'})
2 MATCH (p2:Product {productName: 'iPad Gen 10'})
3 MERGE (p1)-[:NEWEST]->(p2);
```

In the bottom-right status bar, the message "Created 1 relationship, completed after 19 ms." is displayed. On the left side, there is a sidebar with two tabs: "Table" and "Code".

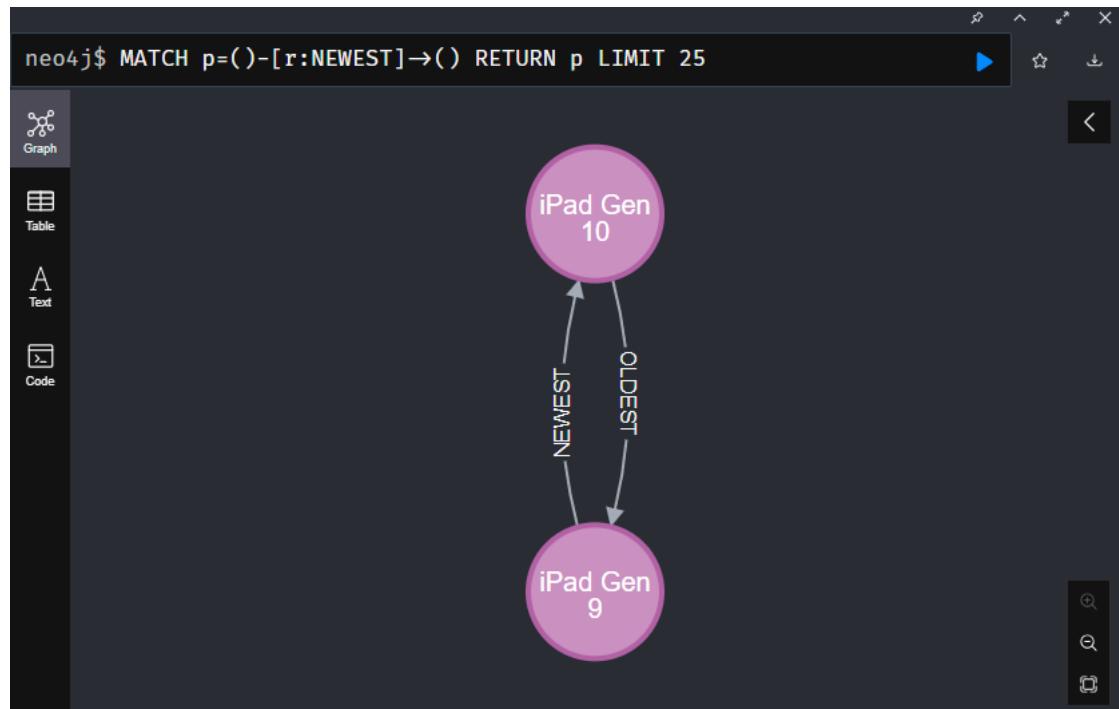
```

1 MATCH (p1:Product {productName: 'iPad Gen 9'})
2 MATCH (p2:Product {productName: 'iPad Gen 10'})
3 MERGE (p2)-[:OLDEST]→(p1);

Created 1 relationship, completed after 68 ms.

```

ผลลัพธ์



### b. READ Edge Operation

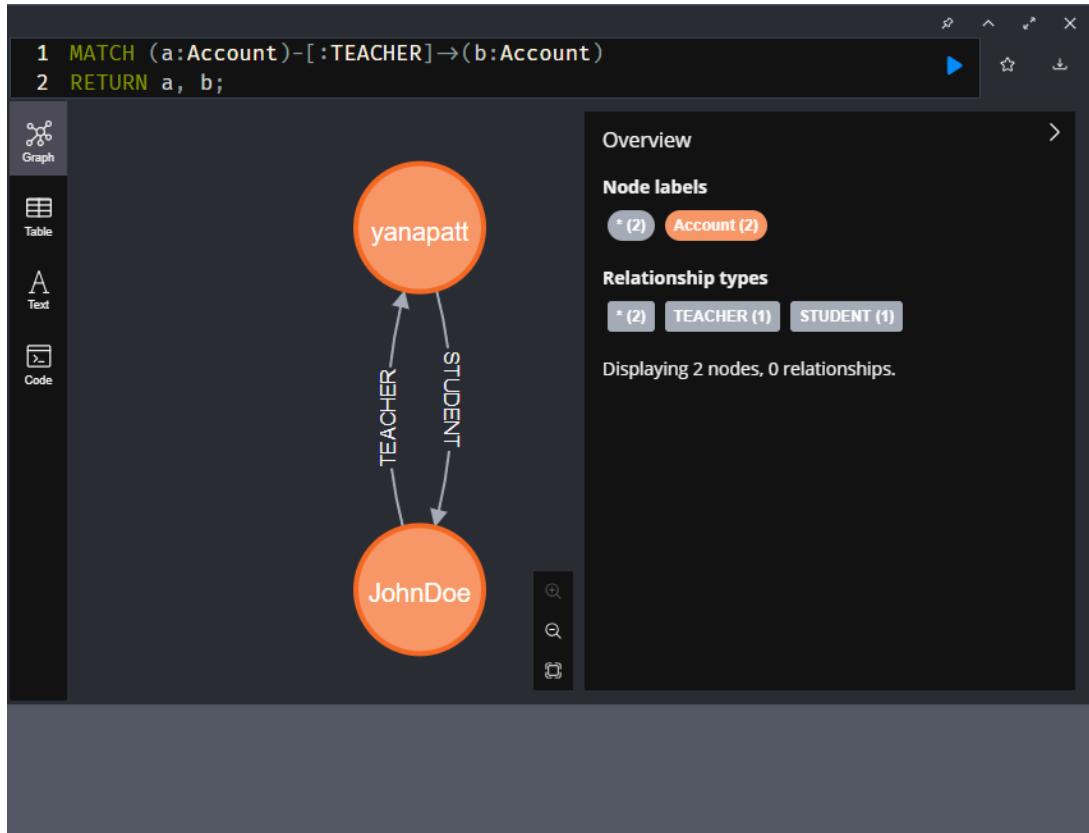
อ่านค่า Node ที่มีความสัมพันธ์กัน

```

MATCH (a:LabelA) - [:RELATIONSHIP_TYPE] -> (b:LabelB)
RETURN a, b;

```

Execute ฝึก Neo4j Browser



Execute ฝึก Cypher-Chell ใน Docker

```
ready to start consuming query after 1 ms, results consumed after another 1 ms
neo4j@neo4j:~$ MATCH (a:Account)-[:TEACHER]->(b:Account)
      RETURN a, b;
+-----+-----+
| a | b |
+-----+-----+
| { :Account {password: "123", role: "User", id: "1745670144931", email: "john.doe@example.com", username: "JohnDoe"} } | { :Account {password: "123", role: "User", id: "1745670138311", email: "yanapatt@example.com", username: "yanapatt"} } |
+-----+-----+
1 row
ready to start consuming query after 8 ms, results consumed after another 0 ms
neo4j@neo4j:~$
```

### c. UPDATE Edge Operation

อัพเดทค่า Property ภายใน Edge

```
MATCH (a:LabelA) - [r:RELATIONSHIP_TYPE] -> (b:LabelB)
SET r.propertyKey = newValue
RETURN r;
```

Execute ฝ่าย Neo4j Browser

```
neo4j$ MATCH (a1:Account)-[r:STUDENT]->(a2:Account)
WHERE a1.username = 'yanapatt' AND a2.username = 'JohnDoe'
SET r.since = 2018
RETURN r;
```

The screenshot shows the Neo4j Browser interface. On the left, there is a sidebar with four tabs: Table, Text (which is selected), and Code. The main area displays the following Cypher query:

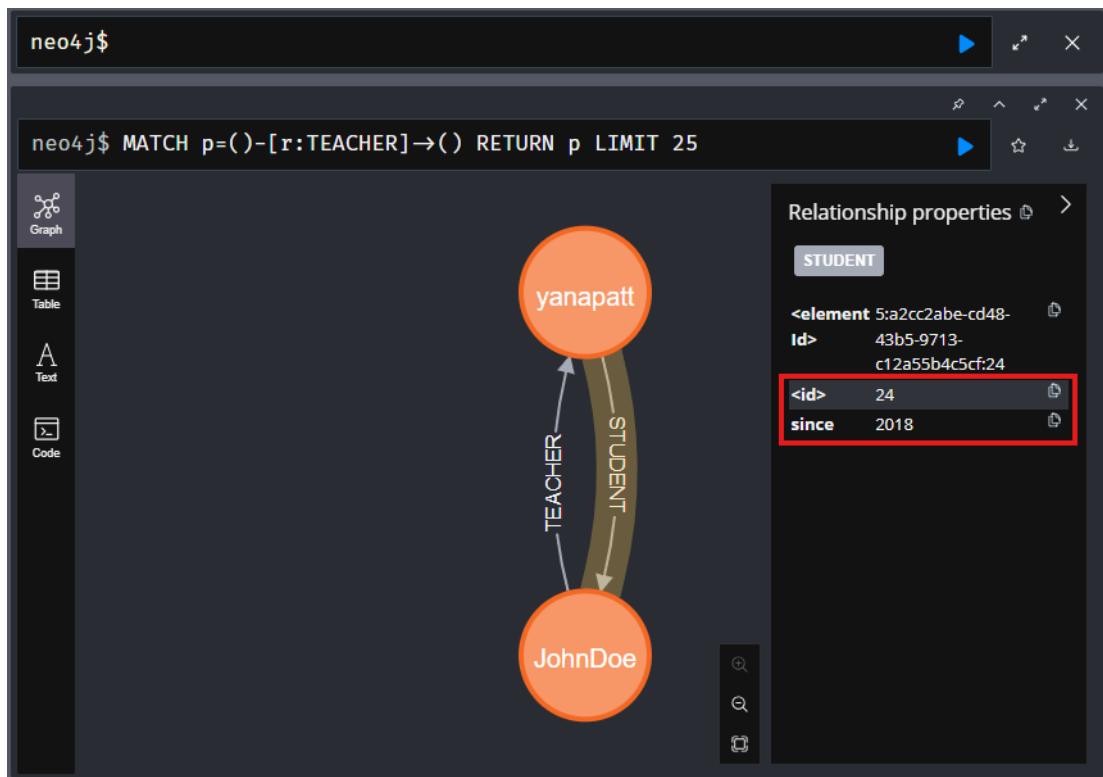
```
1 MATCH (a1:Account)-[r:STUDENT]->(a2:Account)
2 WHERE a1.username = 'yanapatt' AND a2.username = 'JohnDoe'
3 SET r.since = 2018
4 RETURN r;
```

Below the query, the results are shown in a table format:

r
[:STUDENT {since: 2018}]

At the bottom of the interface, there is a "MAX COLUMN WIDTH:" slider.

ผลลัพธ์



#### d. DELETE Edge Operation

ลบ Edge ระหว่าง Node 2 nodes

```
MATCH (a)-[r:RELATIONSHIP_TYPE] -> (b)
DELETE r;
```

Execute ผ่าน Neo4j Browser

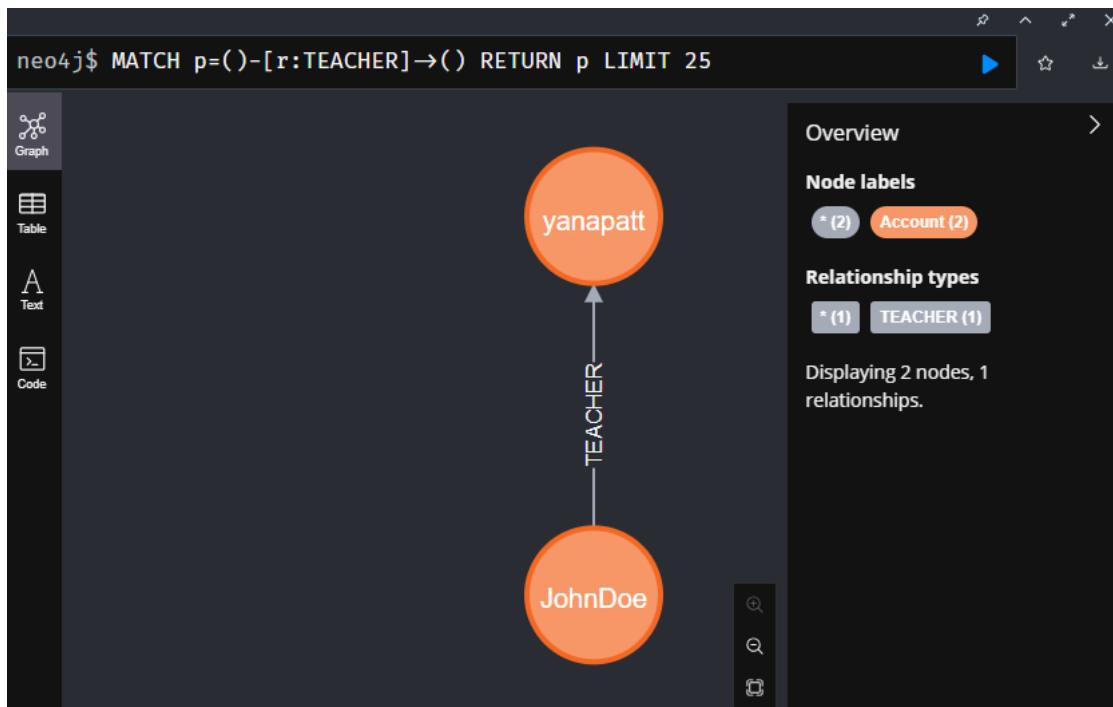
The screenshot shows the Neo4j Browser interface. In the top-left corner, it says "neo4j\$". Below the title bar, there are two code blocks. The first block contains:

```
1 MATCH (a)-[r:STUDENT]->(b)
2 DELETE r;
```

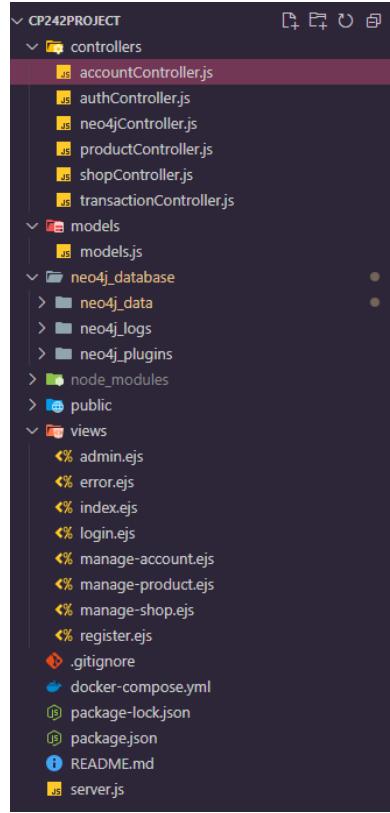
After executing this query, the message "Deleted 1 relationship, completed after 8 ms." appears in the results area.

At the bottom of the browser window, another message "Deleted 1 relationship, completed after 8 ms." is displayed.

ผลลัพธ์



## โครงสร้าง Folder Project



## คำอธิบายเกี่ยวกับ Use Case

The screenshot shows a web application interface for a shop. At the top, there's a navigation bar with "Neo4j Shop App" on the left and "Logout" on the right. Below the navigation bar, there are three main sections:

- Purchase Summary by Category.** A search bar with "SmartPhone" and a blue "Search" button (with a '2' notification). Below it is another search bar with "Samsung Store" and a green "Search" button (with a '2' notification).
- Purchase Summary by Shop.**
- Recommend products for you.** This section displays four recommended products in cards:
  - Samsung Galaxy S25 Ultra**: Price ₧45000, Shop: Samsung Store Count: 16. Tags: Popular, SmartPhone. Button: Buy.
  - OPPO Reno 13 Pro**: Price ₧50000, Shop: Oppo Store Count: 16. Tags: New, SmartPhone. Button: Buy.
  - iPhone 16**: Price ₧25000, Shop: Apple Store Count: 36. Tags: Popular, SmartPhone. Button: Buy.
  - Samsung Galaxy A55**: Price ₧12500, Shop: Samsung Store Count: 42. Tags: Popular, SmartPhone. Button: Buy.

Website นี้เป็น Website เกี่ยวกับการขายสินค้า โดยจะมีร้านค้า สินค้า และผู้ซื้อ ซึ่งสินค้า จะมาจากร้านค้าแต่ละร้าน โดยสินค้าจะมีหมวดหมู่เป็นของตนเอง การทำงานของ Website นี้จะค่อยๆ ตรวจสอบว่า ผู้ซื้อ มีการซื้อของสินค้าประเภทไหนและสินค้าที่มาจากร้านไหนมากที่สุด สิ่งที่จะวัดความซื่อสัตย์ได้นั่นคือ จำนวนการสั่งซื้อสินค้าของลูกค้าแต่ละคน

ด้วยเหตุนี้เราจึงเลือกใช้ Neo4j NoSQL Graph Database เนื่องจากมีความเหมาะสมกับการทำงานแบบระบบแนะนำสินค้า โดยเราจะเลือกเก็บข้อมูล สินค้า ร้านค้า และผู้ซื้อ เป็น Node และความสัมพันธ์ เป็น Edge ซึ่งเราได้ออกแบบ Node และ Edge ตั้งนี้

## คำอธิบาย Node

- Account

```
{
  "id": ,
  "username": ,
  "email": ,
  "role": ,
  "password": 
}
```

Account ประกอบไปด้วย Properties id, username, email, role และ password

- Product

```
{
  "id": ,
  "productDescription": ,
  "shopId": ,
  "productCategory": ,
  "productCounts": ,
  "productTags": ,
  "productName": ,
  "productPrice": ,
  "productImageUrl": 
}
```

Product ประกอบไปด้วย Properties id, productDescription, shopId, productCategory, productCounts, productTage, productName, productPrice และ productImageUrl

- Shop

```
{
  "id": ,
  "shopAddress": ,
  "shopName": ,
  "shopPhone": 
}
```

Product ประกอบไปด้วย Properties id, shopAddress, shopName และ shopPhone

## คำอธิบาย Edge (Relationship)



Account – [ :BOUGHT ] -> Product

Product – [ :FROM ] -> Shop

โดย Relationship BOUGHT ระหว่าง Account และ Product จะมี Property Count ในการนับจำนวนครั้งที่ ผู้ซื้อ กดซื้อสินค้าในแต่ละครั้ง ป้องกันการสร้าง Relationship ซ้ำโดยไม่จำเป็น

เมื่อผู้ใช้สมัครและล็อกอินเข้าสู่ Website ในเริ่มต้นจะยังไม่มีสินค้าแนะนำ เนื่องจากยังไม่มีการสร้าง Relationship ระหว่าง Account และ Product

## Register

Full Name

JohnDoe

Email Address

john.doe@example.com

Password

...

Confirm Password

...

Role

User

[Register](#)

[Already have an account? Login](#)

Neo4j Shop App

[Logout](#)

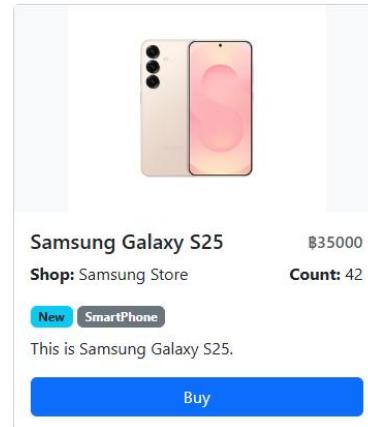
## Products



**Samsung Galaxy S25 Ultra** ₦45000  
**Shop:** Samsung Store **Count:** 14

[Popular](#) [SmartPhone](#)  
 This is Samsung Galaxy S25 Ultra.

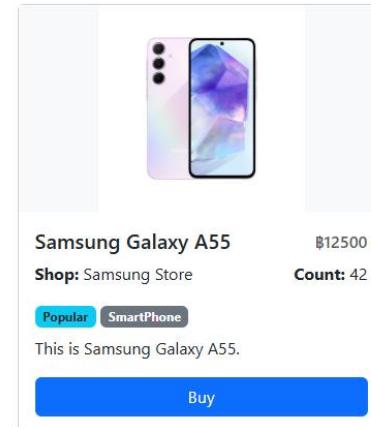
[Buy](#)



**Samsung Galaxy S25** ₦35000  
**Shop:** Samsung Store **Count:** 42

[New](#) [SmartPhone](#)  
 This is Samsung Galaxy S25.

[Buy](#)



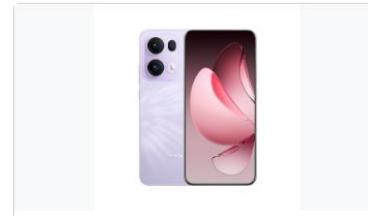
**Samsung Galaxy A55** ₦12500  
**Shop:** Samsung Store **Count:** 42

[Popular](#) [SmartPhone](#)  
 This is Samsung Galaxy A55.

[Buy](#)



**OPPO Reno 13** ₦33500



**OPPO Reno 13 Pro** ₦50000

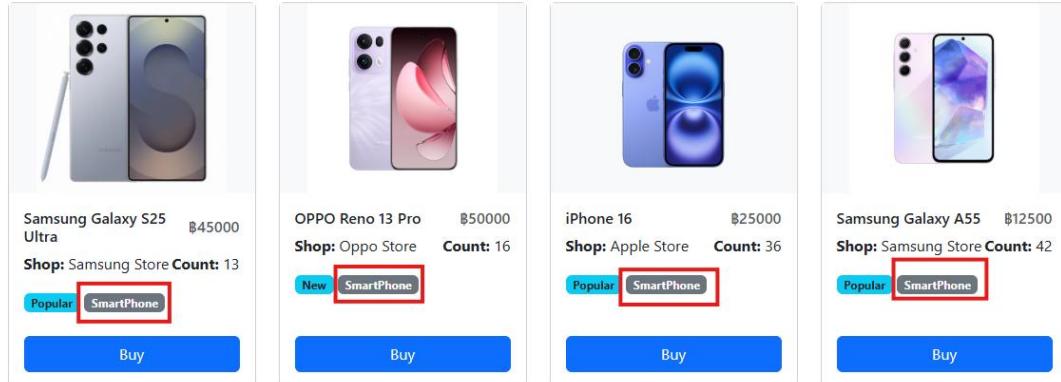


**Samsung Galaxy Z Flip 6** ₦72500

ทันทีที่ผู้ใช้กดซื้อสินค้า ระบบจะมีการสร้าง Relationship ระหว่าง Account และ Product การแนะนำสินค้าจะเกิดขึ้น โดยแบ่งได้ 2 แบบ

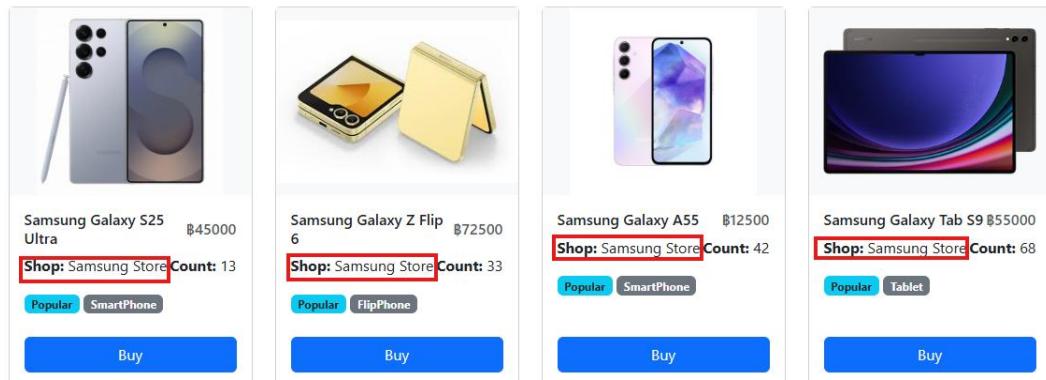
- แบบที่ 1 แนะนำสินค้าที่มี Category เดียวกัน

#### Recommend products for you.



- แบบที่ 2 แนะนำสินค้าที่มีจาก Shop เดียวกัน

#### Recommend shops for you.



โดยใช้ Count ที่เป็น Properties บน Relationship ระหว่าง Account และ Product ในการพิจารณาเลือกแนะนำสินค้า ซึ่งจะพิจารณาเลือกสินค้าที่มี Count สูงที่สุด

Purchase recorded successfully!

X

### Purchase Summary by Category.

FlipPhone

2

SmartPhone

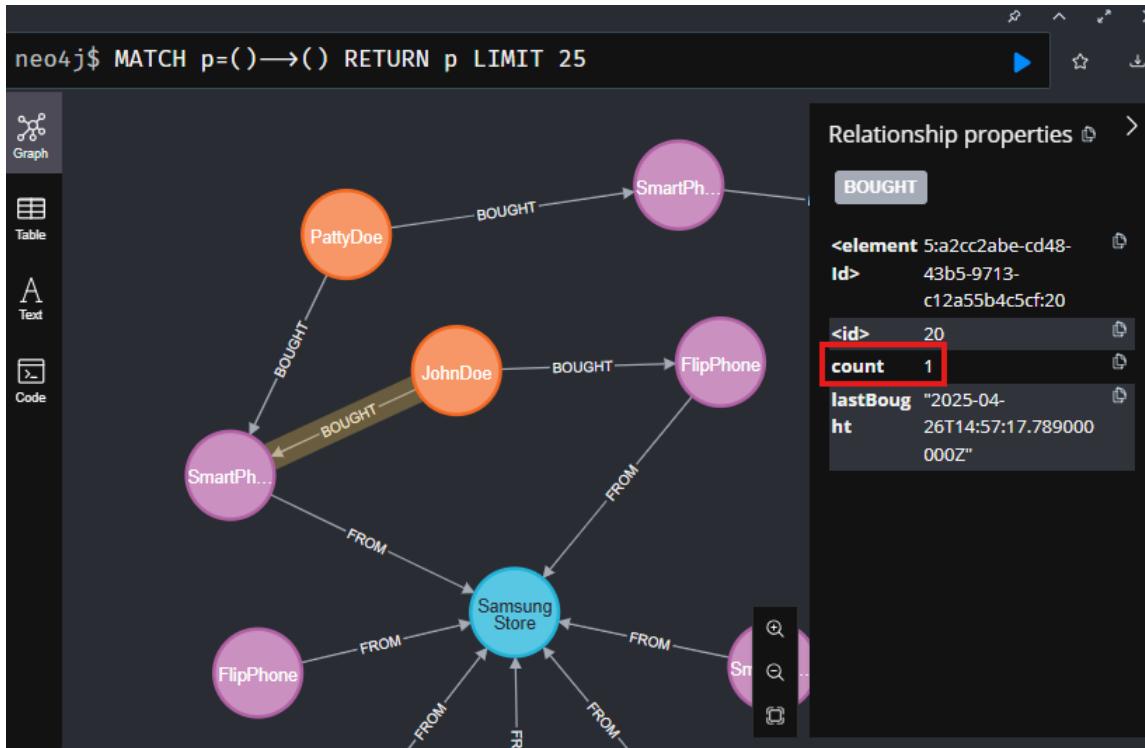
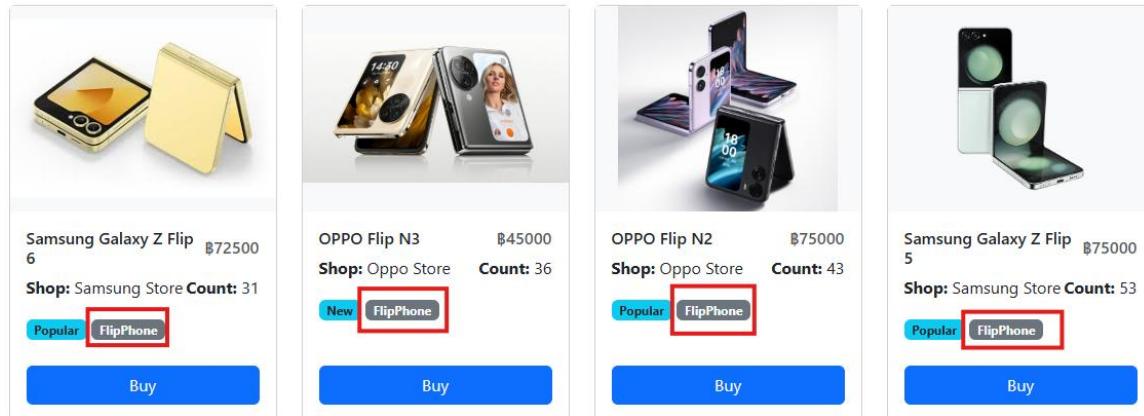
1

### Purchase Summary by Shop.

Samsung Store

3

### Recommend products for you.



## คำอธิบายเกี่ยวกับการทำงานของ Application

ที่ models.js

```
const neo4j = require('neo4j-driver');

const driver = neo4j.driver(
  'bolt://localhost:7687',
  neo4j.auth.basic('neo4j', 'Neo4j12345*')
);

module.exports = driver;
```

มีการเรียกใช้ neo4j-driver ที่เป็น Package จาก Node.js ในการติดต่อกับ Neo4j โดยภายใต้จะมี การกำหนด Port การเชื่อมต่อ และ Authenticate ตามที่ได้ตั้งไว้ใน docker-compose.yml

ที่ neo4jController.js (โค้ดบางส่วน)

```
const driver = require('../models/models');

exports.authNeo4j = async (req, res, next) => {
  const session = driver.session();
  try {
    await session.run('RETURN 1');
    res.render('index', { status: 'Connected to Neo4j successfully!' });
  } catch (err) {
    next(err);
  } finally {
    await session.close();
  }
}

// CREATE
exports.createNode = async (label, properties) => {
  const session = driver.session();
  try {
    const cypherQuery = `CREATE (n:${label})
${properties} RETURN n`;
    await session.run(cypherQuery, { properties });
  } finally {
    await session.close();
  }
}
```

```

};

// READ
exports.readNodes = async (label) => {
    const session = driver.session();
    try {
        const result = await session.run(`MATCH
(n:${label}) RETURN n`);
        return result.records.map(record =>
record.get('n').properties);
    } finally {
        await session.close();
    }
};

```

เป็นส่วนที่ใช้ในการทำ CRUD Operation ใน Neo4j โดยจะมีการเตรียม Cypher Query ล่วงหน้า  
เพื่อรอ Execute คำสั่ง

ที่ productController.js (โค้ดบางส่วน)

```

const neo4jController = require('./neo4jController');

exports.showManageAccountPage = async (req, res, next) => {
    try {
        const accounts = await
neo4jController.readNodes('Account');

        const accountEmail = req.params.email;
        let account = null;
        let isEdit = false;

        if (accountEmail) {
            account = accounts.find(a => a.email ===
accountEmail);
            isEdit = true;
        }

        res.render('manage-account', {
            message: req.query.message || null,
            accounts,
            isEdit,
            account
        });
    } catch (err) {

```

```

        next(err);
    }
};

// สำหรับ login: ที่ account โดย email และ password
exports.findAccountByEmailAndPassword = async (email,
password) => {
    const accounts = await
neo4jController.readNodes('Account');
    return accounts.find(acc => acc.email === email &&
acc.password === password);
};

```

เปรียบเสมือนตัวกลางในการรับ Request จากผู้ใช้ เพื่อให้ neo4jController Execute Cypher Query และส่งต่อผลลัพธ์จากการเรียก Method ใน neo4jController กลับไปให้หน้า ejs เพื่อแสดงข้อมูลประกอบการแสดงผล User Interface โดยจะมี accountController (จัดการเกี่ยวกับบัญชี), transactionController (จัดการเกี่ยวกับการซื้อสินค้า) และ shopController (จัดการเกี่ยวกับร้านค้า) ที่มีหน้าที่เฉพาะเจาะจงในแต่ละงาน แต่มีหลักการทำงานที่คล้ายคลึงกัน

ที่ authController.js (โค้ดบางส่วน)

```

const neo4jController = require('./neo4jController');

exports.showManageAccountPage = async (req, res, next) => {
    try {
        const accounts = await
neo4jController.readNodes('Account');

        const accountEmail = req.params.email;
        let account = null;
        let isEdit = false;

        if (accountEmail) {
            account = accounts.find(a => a.email ===
accountEmail);
            isEdit = true;
        }

        res.render('manage-account', {
            message: req.query.message || null,
            accounts,

```

```

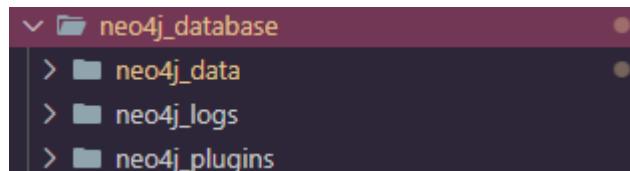
        isEdit,
        account
    });
} catch (err) {
    next(err);
}
};

// สำหรับ login: หา account โดย email และ password
exports.findAccountByEmailAndPassword = async (email,
password) => {
    const accounts = await
neo4jController.readNodes('Account');
    return accounts.find(acc => acc.email === email &&
acc.password === password);
};

```

มีหน้าที่ในการ Verify ความถูกต้องของบัญชีผู้ใช้ก่อนเข้าสู่ระบบ และจัดการบัญชีของ Account เพื่อจำแนก User ปกติและ Admin ส่งผลให้ 2 บทบาทนี้จะเข้าถึงฟังก์ชันต่าง ๆ ในเว็บไซต์ได้แตกต่างกัน

ข้อมูล Database ทั้งหมดที่อยู่ท่ามกลาง จะถูกจัดเก็บไว้ใน Folder ภายใต้ Folder Project



## สรุปผลการดำเนินงาน

โครงการนี้มีวัตถุประสงค์เพื่อพัฒนาระบบแนะนำสินค้าต่าง ๆ บนเว็บไซต์ให้แก่ผู้ใช้งาน โดยเป็นการแนะนำที่สอดคล้องกับผู้ใช้งานแต่ละคน ซึ่งการใช้ Neo4j NoSQL Graph Database ถือได้ว่ามีประสิทธิภาพยอดเยี่ยมและเป็นอีกหนึ่งทางเลือกในการจัดการหรือบริหารข้อมูลสำหรับการทำงานเว็บไซต์ในลักษณะนี้ การมี Node และ Edge ทำให้การดึงข้อมูลต่าง ๆ มีความยืดหยุ่นมากยิ่งขึ้น อย่างไรก็ตามในระหว่างการพัฒนาเว็บไซต์ก็ได้พบกับอุปสรรค-many และเราได้คาดหวังว่าโครงการนี้จะเป็นประโยชน์แก่ผู้ที่ต้องการศึกษาและนำโครงการไปต่อยอดให้ดีขึ้นต่อไป

## บรรณานุกรม

[k]code. (2565, กุมภาพันธ์). **Neo4j Docker quick start.** สืบค้นเมื่อวันที่ 24 เมษายน 2568.

จาก <https://www.youtube.com/watch?v=kyfcr5UPIqw&t=154s>

Neo4jOfficial. (2568). **Neo4j Documentation.** สืบค้นเมื่อวันที่ 24 เมษายน 2568. จาก

<https://neo4j.com/-docs/>