



02739499 - *Information Technology Project*

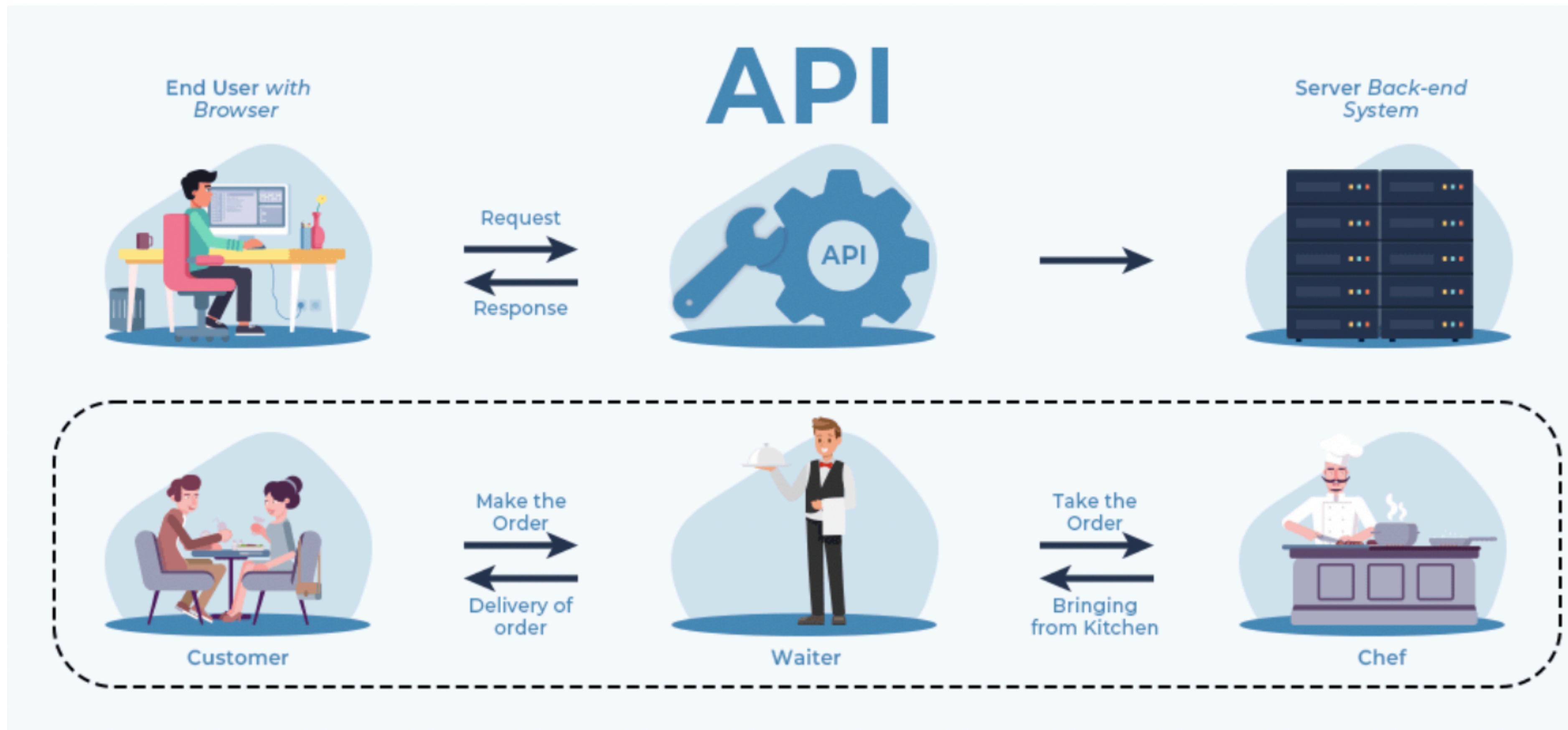
Application Programming Interface - API

Prasertsak U.
Aug, 2567

Outline

- ❖ What is API ?
- ❖ Type of API
- ❖ Workshop on Rest API
 - ❖ REST API Development
 - ❖ API Rate Limiting
 - ❖ Service on HTTPS
 - ❖ API Documentation

What is API?



ภาพจาก: <https://www.geeksforgeeks.org/what-is-an-api/>

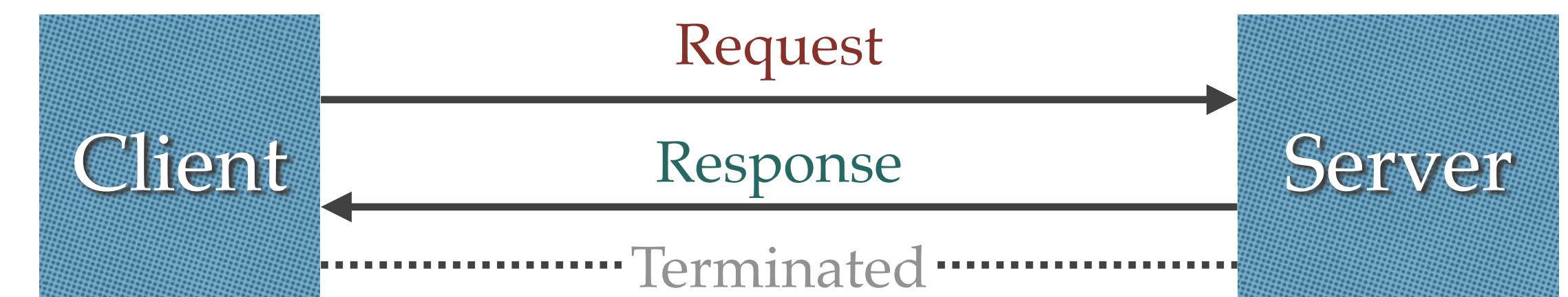
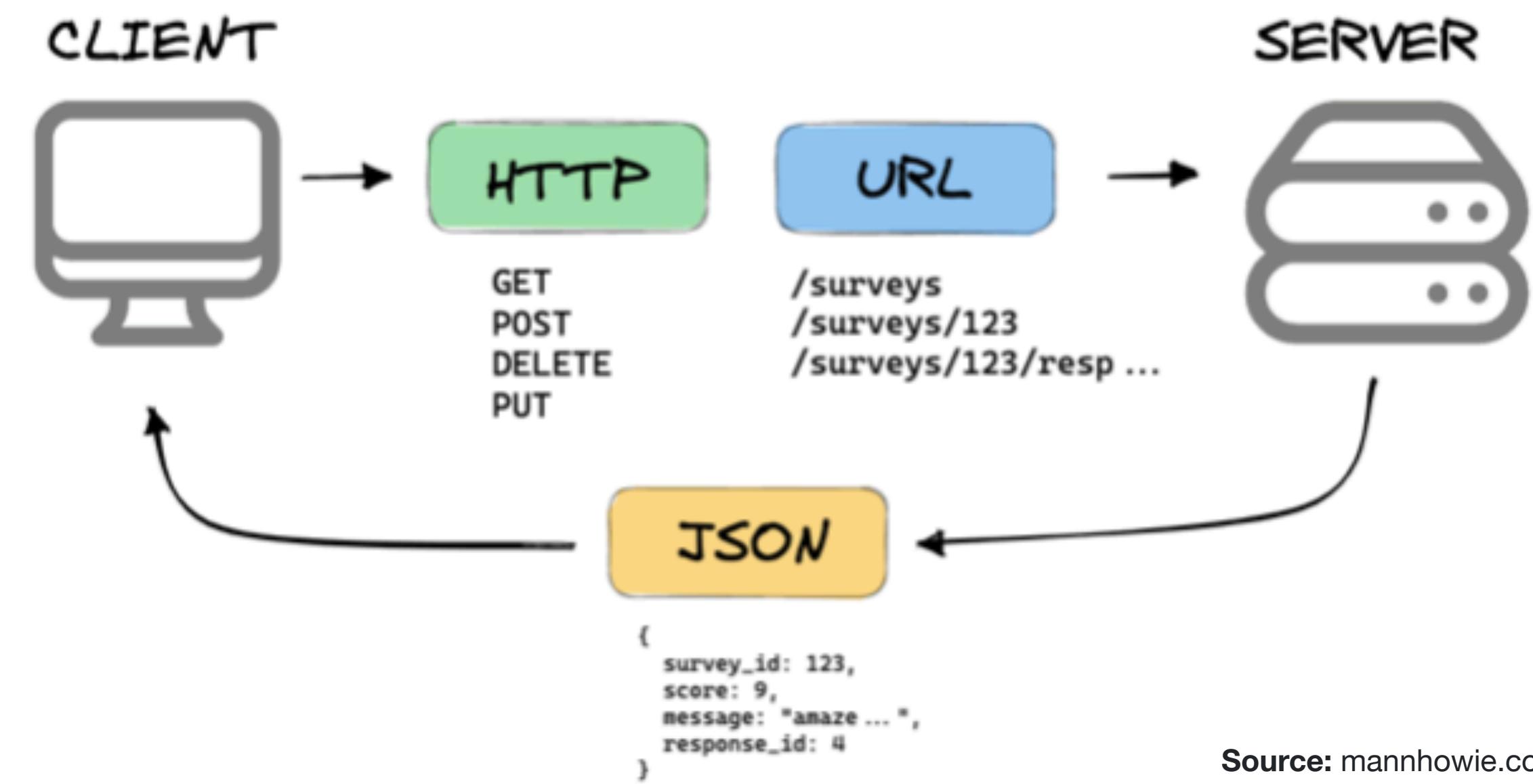
Major Type of API

- ❖ SOAP - **S**imple **O**bject **A**ccess **PA uni-directional protocol**
- ❖ REST API - **R**epresentational **S**tate **T**ransfer : *A uni-directional protocol*
- ❖ WebSocket - *A bi-directional protocol*
- ❖ GraphQL - by Facebook
- ❖ gRPC - by Google

REST API

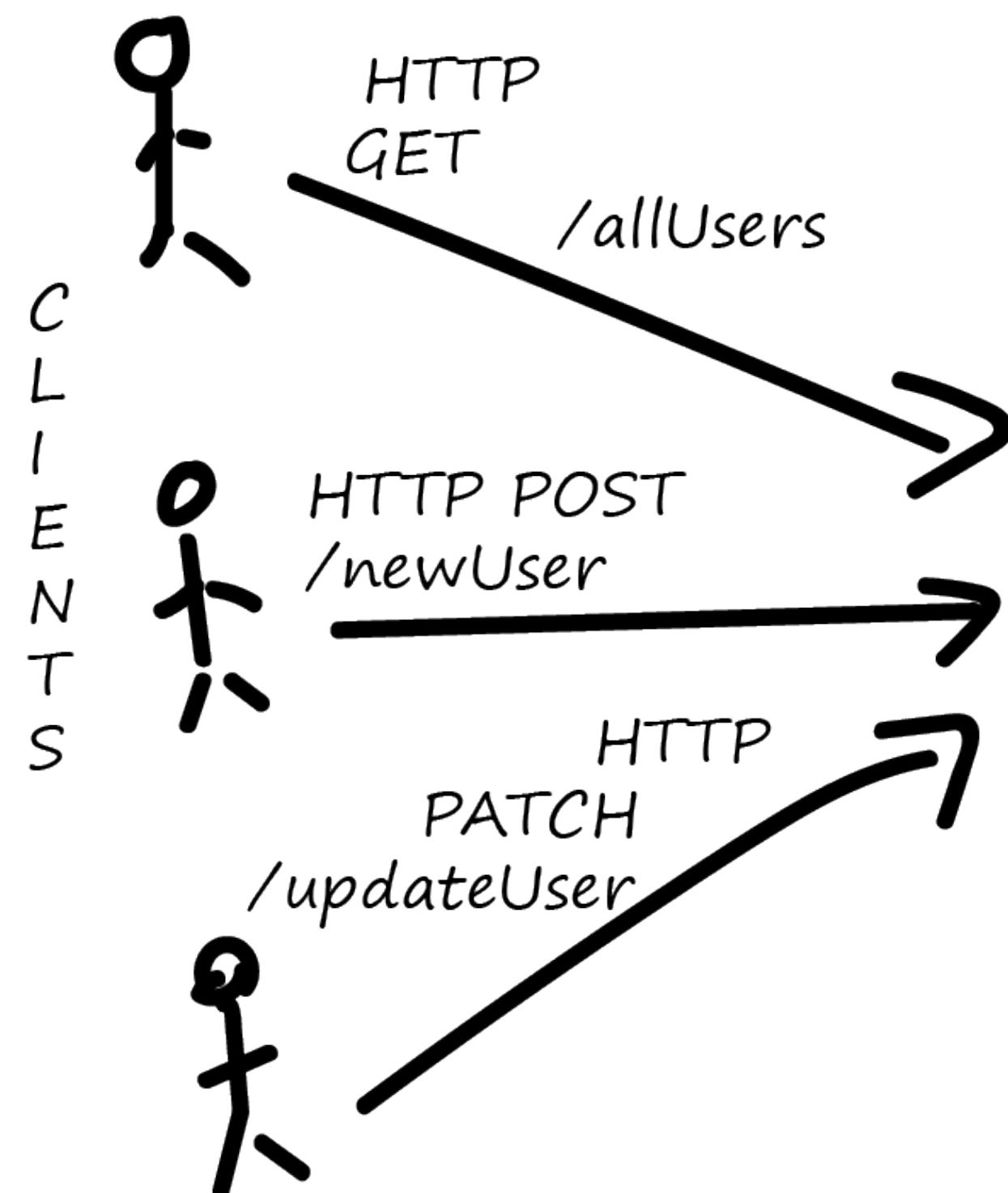
- ❖ ไม่บังคับรูปแบบข้อมูลที่ใช้รับส่งระหว่าง Client กับ Server แต่ส่วนมากนิยมใช้รูปแบบ **JSON**
- ❖ ทำงานบน HTTP Protocol รองรับคำสั่ง GET, POST, PUT, PATCH, DELETE ของ Http Protocol
- ❖ เป็นการสื่อสารแบบ Stateless

WHAT IS A REST API?



REST API

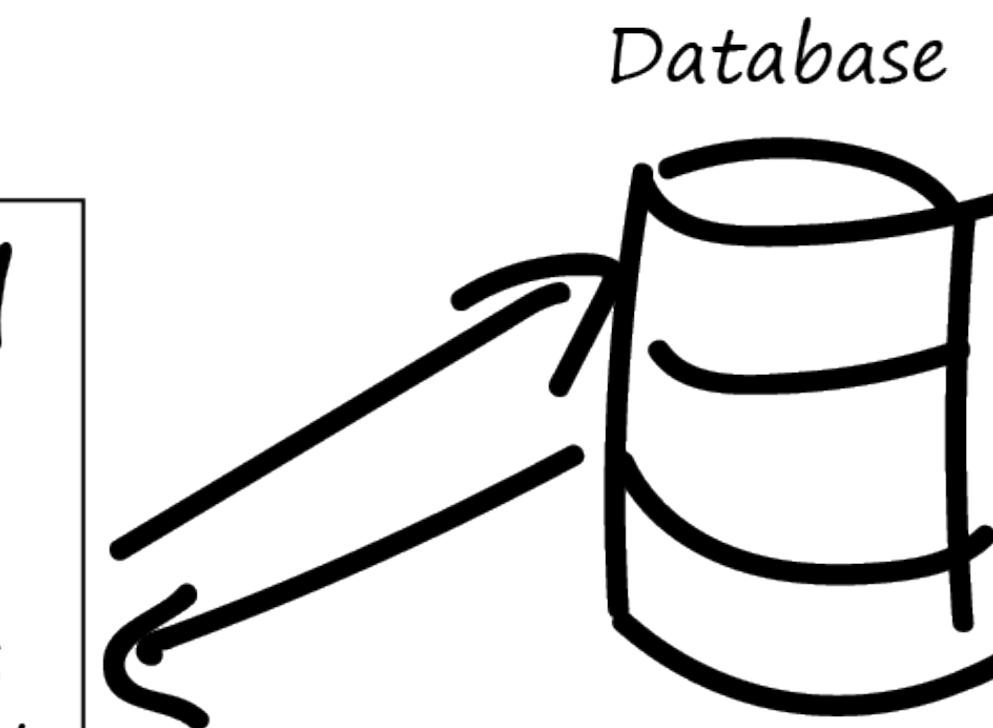
Rest API Basics



Our Clients, send HTTP Requests and wait for responses

Rest API

Receives HTTP requests from Clients and does whatever request needs. i.e create users

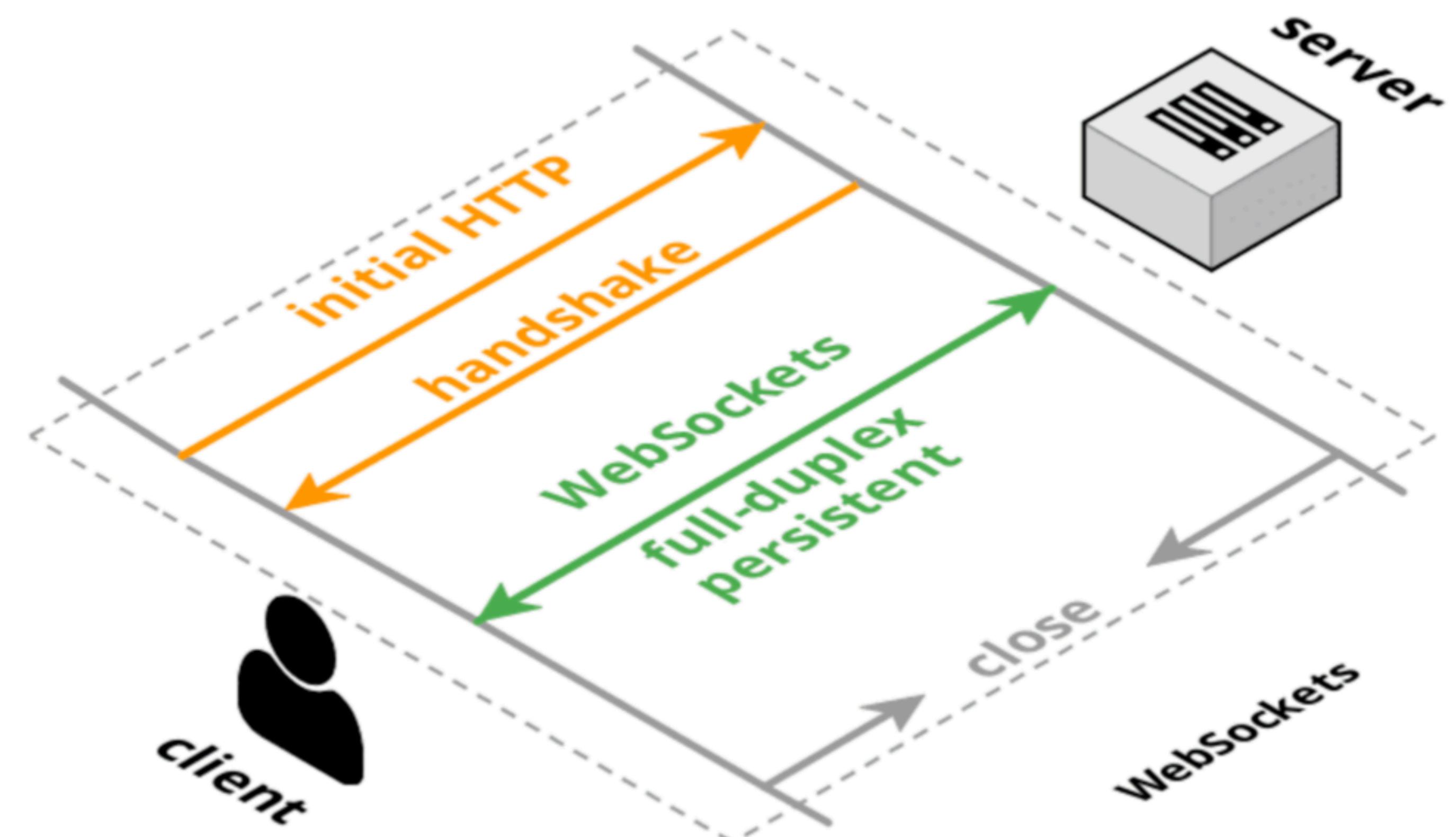


Our Rest API queries the database for what it needs

Response: When the Rest API has what it needs, it sends back a response to the clients. This would typically be in JSON or XML format.

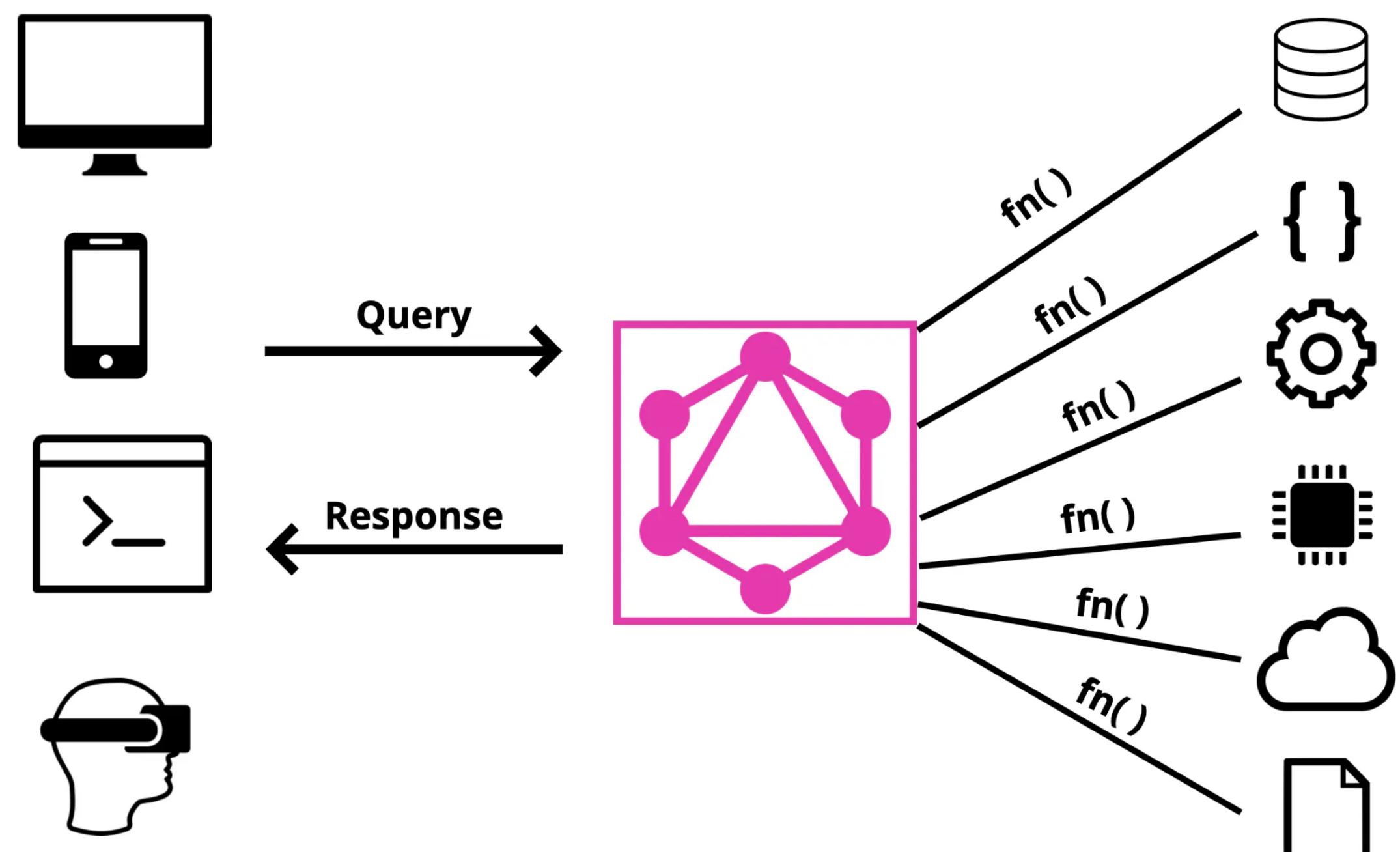
WebSocket

- ❖ เป็นการสื่อสารแบบ 2 ทาง - *bi-directional protocol*
- ❖ ส่วนมากจะเหมาะสมกับการใช้งานประเภท Realtime Applications, Data Streaming, Gaming, Chat เป็นต้น



GraphQL

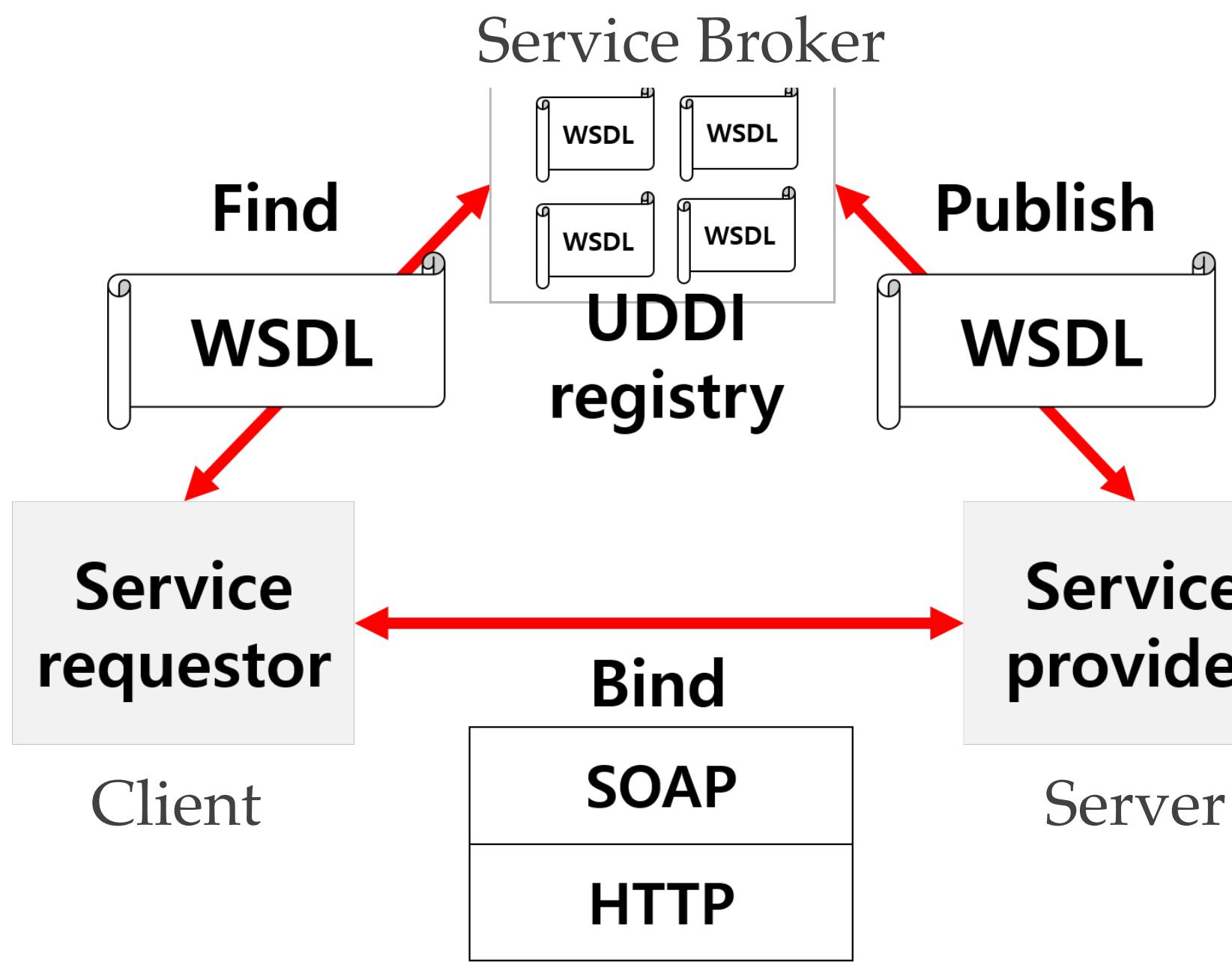
- ❖ มี Endpoint เดียวทำให้ง่ายต่อการอ้างถึง
- ❖ ทำงานผ่าน POST อย่างเดียว และสามารถระบุความต้องการได้ยึดหยุ่นกว่า REST โดยไม่ต้องเปลี่ยน Endpoint
- ❖ Code มีความซับซ้อน เนื่องจากต้องมีการวางแผนหรือเงื่อนไขเพื่อให้จัดการกับ Request ที่แตกต่างกันภายใต้ Endpoint เดียวได้
- ❖ ระยะเวลาในการเรียนรู้สำหรับผู้เริ่มต้น จะใช้เวลามากกว่ารูปแบบอื่น



ภาพจาก <https://lo-victoria.com/graphql-for-beginners-introduction>

SOAP

- ❖ ใช้ XML เป็นรูปแบบในการรับส่งข้อมูล ระหว่าง Client กับ Server
- ❖ ใช้ไฟล์ WSDL ในการอธิบายรายละเอียดของ API



```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" name="TestService" targetNamespace="http://www.examples.com/wsdl/TestService.wsdl">
  <message name="getBMIRequest">
    <part name="weight" type="xsd:float"/>
    <part name="height" type="xsd:float"/>
  </message>
  <message name="getBMIResponse">
    <part name="bmi" type="xsd:float"/>
  </message>
  <portType name="Test_PortType">
    <operation name="calculateBMI">
      <input message="tns:getBMIRequest"/>
      <output message="tns:getBMIResponse"/>
    </operation>
  </portType>
  <binding name="Test_Binding" type="tns:Test_PortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="calculateBMI">
      <soap:operation soapAction="calculateBMI"/>
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:examples:testservice" use="encoded"/>
      </input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:examples:testservice" use="encoded"/>
      </output>
    </operation>
  </binding>
  <service name="BMI_Service">
    <documentation>WSDL File for TestService</documentation>
    <port binding="tns:Test_Binding" name="BMI_Port">
      <soap:address location="http://localhost:9000/bmicalculator/" />
    </port>
  </service>
</definitions>
```

ตัวอย่าง SOAP message

SOAP Envelope
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

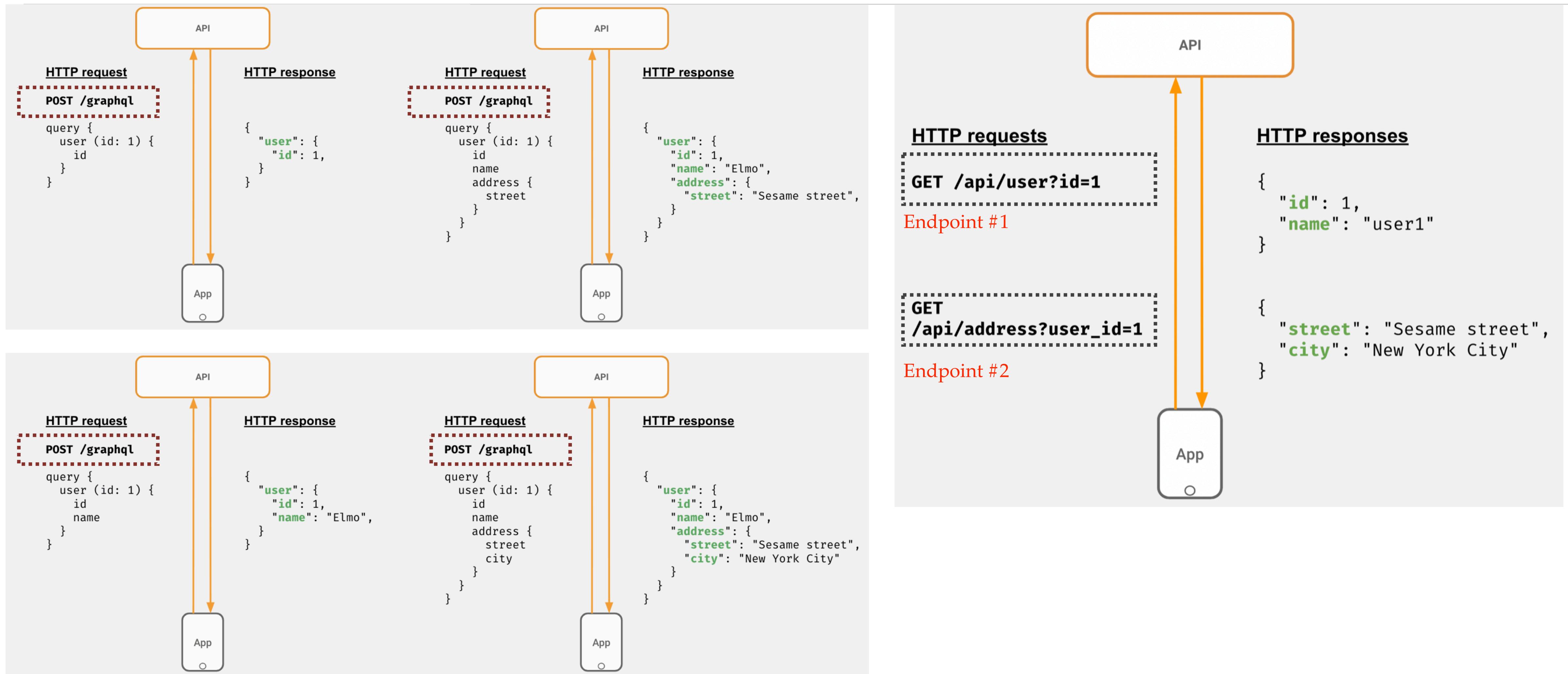
SOAP Header
<soap:Header>
Optional header parts
</soap:Header>

SOAP Body
<soap:Body>
SOAP Message Payload
Optional SOAP Faults
</soap:Body>

</soap:Envelope>

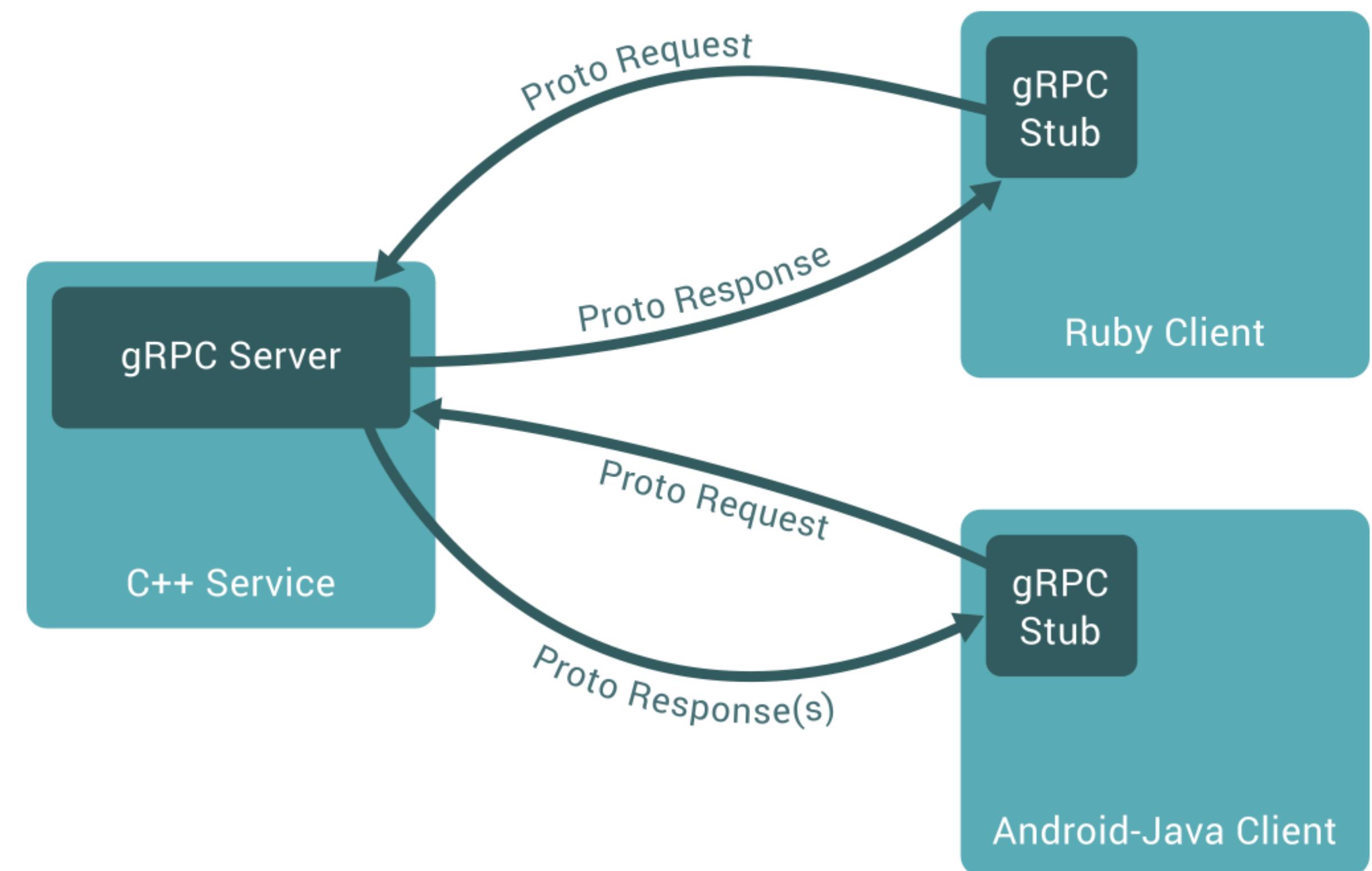
ตัวอย่าง WSDL file

GraphQL vs REST



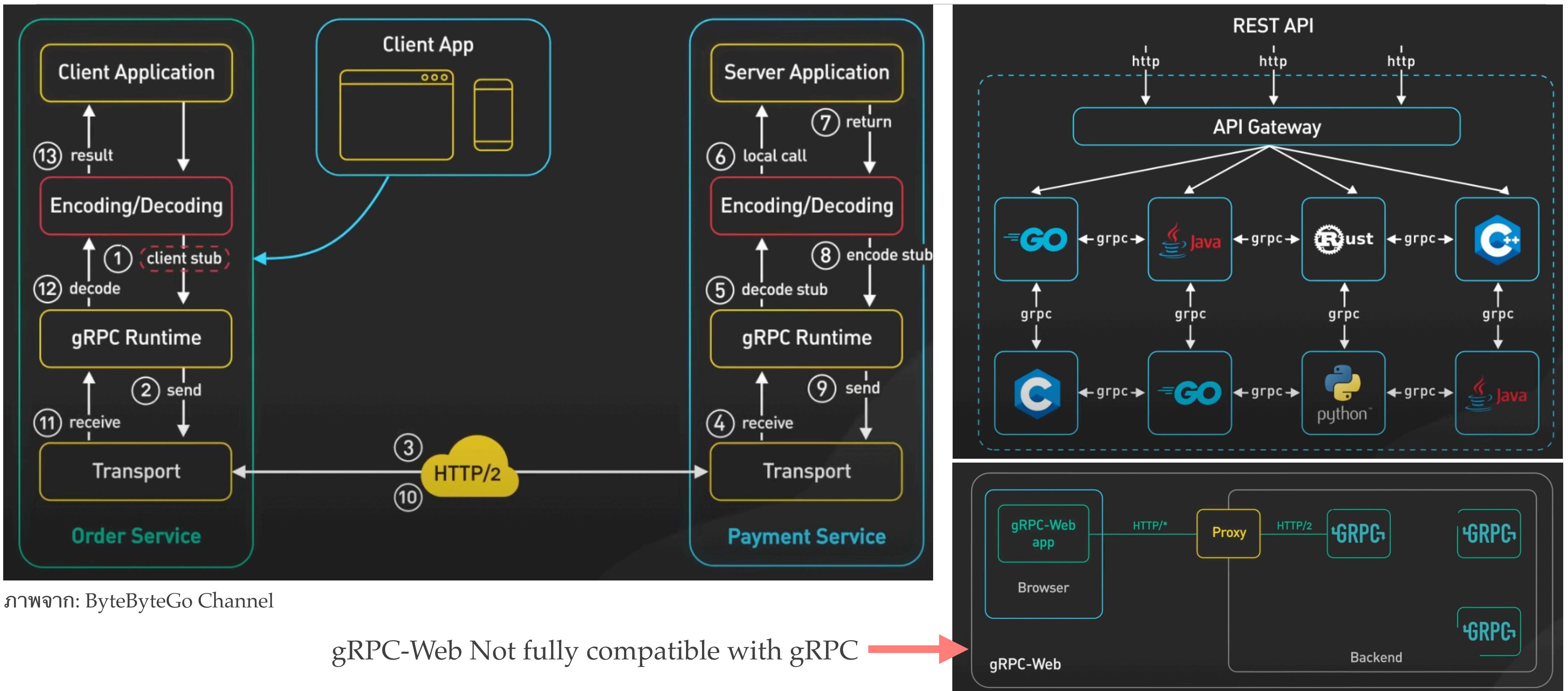
gRPC

- ❖ รองรับ HTTP/2
- ❖ มีขั้นตอนการทำงานคล้ายกับ SOAP โดยเรียกส่วนที่ใช้อธิบายรูปแบบการสื่อสารว่า protocol buffer (.proto)
- ❖ หมายเหตุการใช้สื่อสารระหว่าง Service ด้วยกันเองบนฝั่ง Backend เพราะทำงานได้เร็ว
- ❖ ทำงานเร็วกว่ารูปแบบอื่น เนื่องจากข้อมูลที่รับส่งระหว่าง Client กับ Server จะอยู่ในรูปแบบ Binary ที่ถูกทำให้เป็น Serialized
- ❖ Code มีความซับซ้อน และค่อนข้างยาก



ภาพจาก <https://grpc.io/docs/what-is-grpc/introduction/>

gRPC





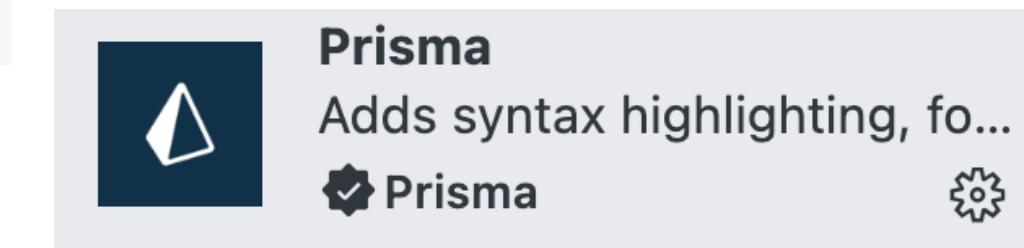
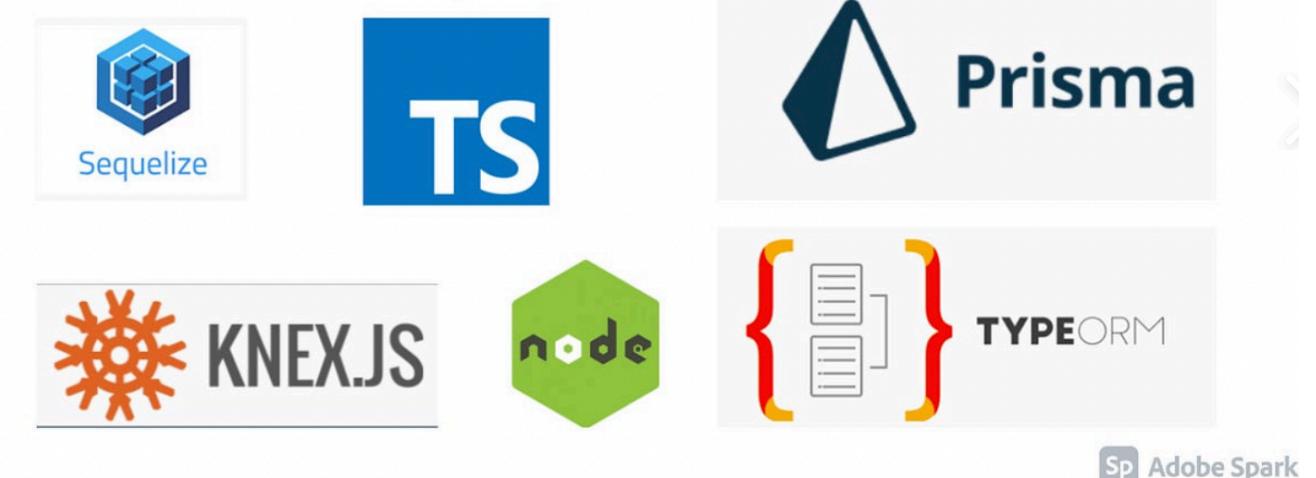
WORKSHOP

Workshop on Rest API using NodeJS

- ❖ เตรียมซอฟต์แวร์ที่จำเป็นสำหรับการพัฒนา API
 - ❖ NodeJS. <https://nodejs.org/en> (ควรเป็น LTS version)
 - ❖ Prisma ORM. - จะติดตั้งไว้ภายใน Project หลังจากสร้าง Node Project เรียบร้อยแล้ว
(อ่านรายละเอียดเพิ่มเติมได้ที่ <https://www.prisma.io/docs/orm/introduction>)
 - ❖ NodeMon.

```
npm install -g nodemon
```
 - ❖ VS-Code - install a Prisma extension
 - ❖ MySQL Database - <https://dev.mysql.com/downloads/mysql/>
 - ❖ Postman - <https://www.postman.com/downloads/>
 - ❖ Web Browser

NODE JS ORM'S SEQUELIZE, TYPEORM, PRISMA

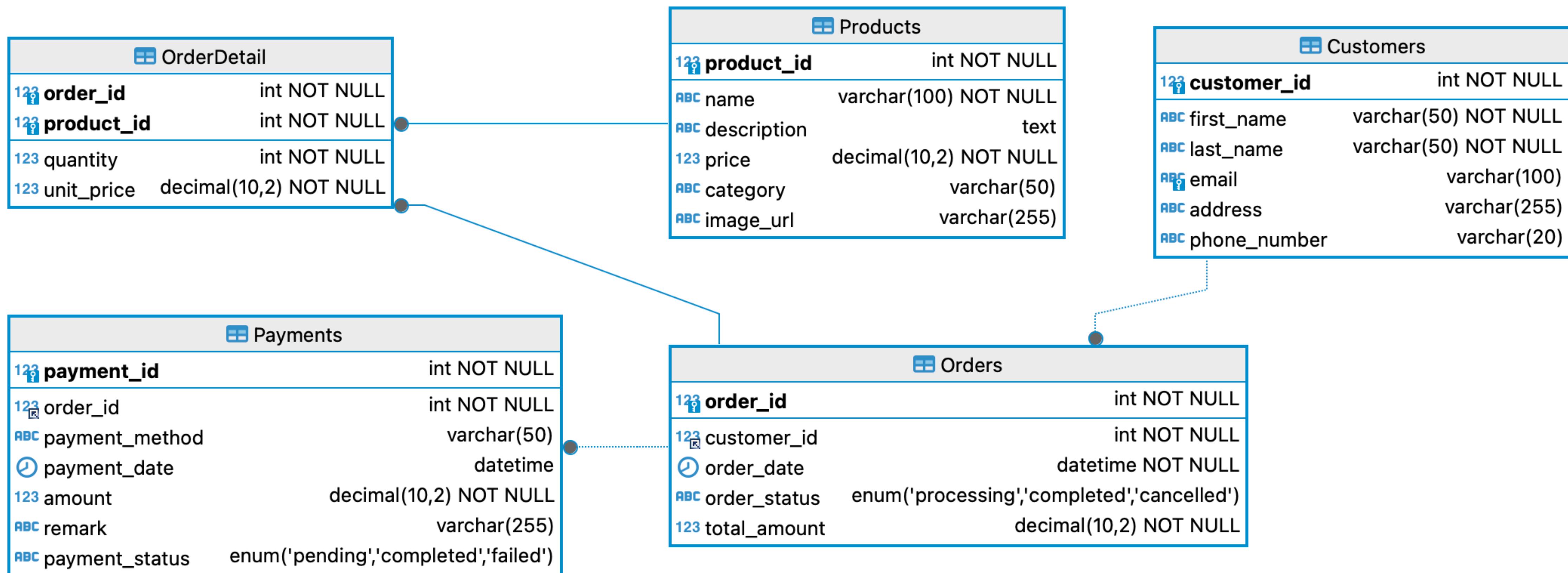


Workshop on Rest API using NodeJS (ต่อ)

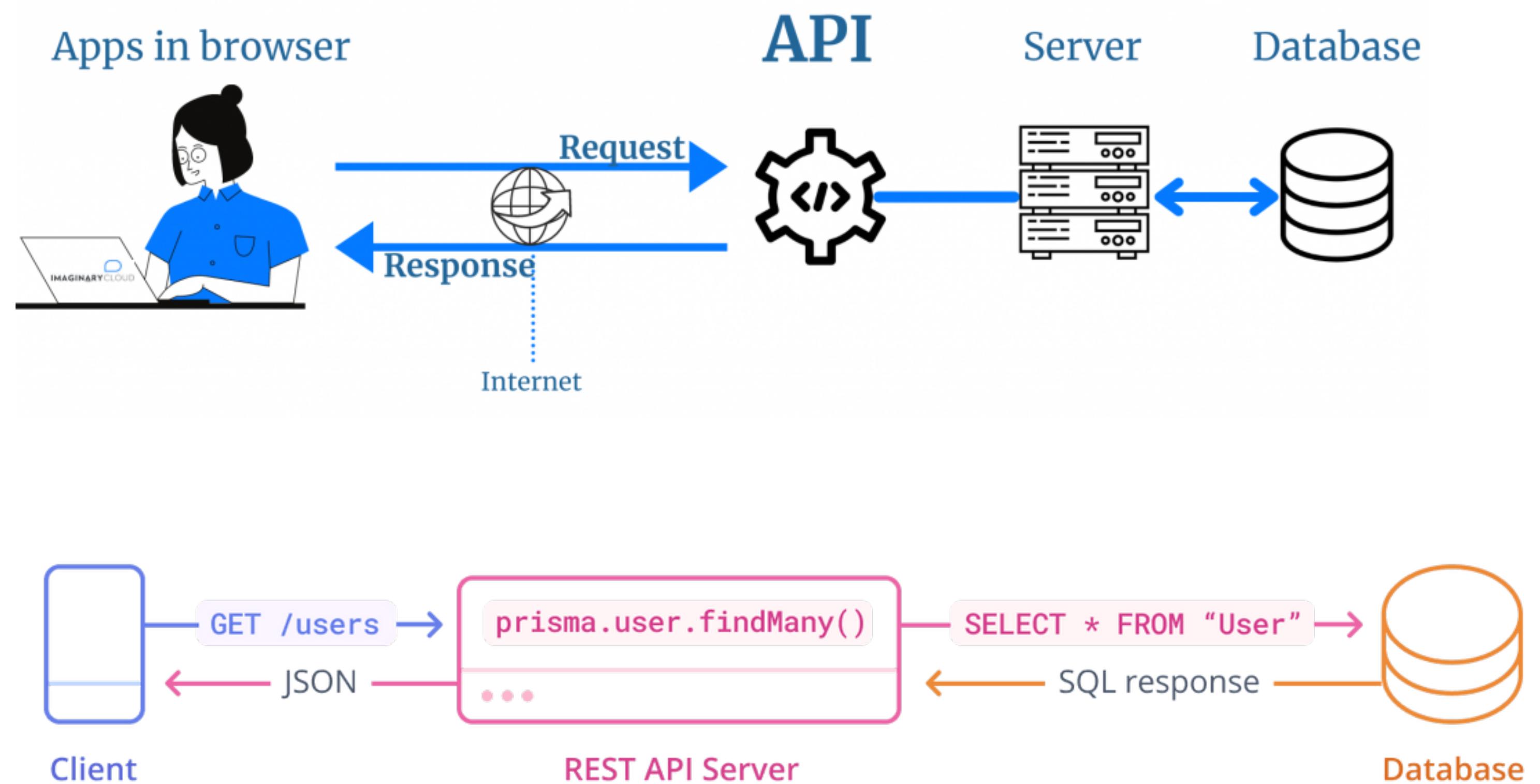
- ❖ เตรียมข้อมูลสำหรับติดตั้งบน MySQL Database
- ❖ ให้ Download โครงสร้างข้อมูลสำหรับ MySQL ได้จาก [ที่นี่](#)
- ❖ ทำการสร้างฐานข้อมูล จาก SQL Script ที่ได้รับจากการ Download โดยใช้ DBeaver หรือ Database Manager ตัวใดก็ได้
- ❖ สร้าง Folder ชื่อว่า *NodeProjects/MiniStoreServices* ไว้ที่ Drive D:\
- ❖ เปิดโปรแกรม VS-Code และทำการ Open Folder : *NodeProjects/MiniStoreServices*

Data Relational Diagram

Database Name: MiniStore



Why ORM ?



An ORM, or Object-Relational Mapper, acts as a *bridge* between your programming language and database

Benefits of using an ORM:

- **Reduced complexity:** Instead of writing complex SQL queries, you interact with data using objects familiar to your programming language.
- **Improved readability:** Your code becomes more readable and easier to maintain.
- **Reduced errors:** ORMs can handle common database tasks like data type conversions and escaping special characters, preventing potential errors in your code.
- **Increased productivity:** With less time spent writing complex SQL queries, you can focus on the logic and functionality of your program.

เริ่มต้นพัฒนา Node Server and Rest API

- ❖ เปิด Terminal ภายใน VS-Code ที่ได้ Open Folder ไว้ก่อนหน้านี้
- ❖ สร้าง Node Project โดยพิมพ์คำสั่งบนเทอร์มินอล: **npm init -y**
- ❖ ติดตั้ง Prisma ORM ภายใน Project ด้วยคำสั่ง: **npm install -g @prisma/client prisma**
- ❖ ติดตั้ง express ภายใน Project ด้วยคำสั่ง: **npm install express**

ตรวจสอบข้อมูลภายในไฟล์ package.json

{ package.json

```
1  {
2    "name": "mystoreservices",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    [REDACTED]
7    "scripts": {
8      "start": "nodemon index.js", ←
9      "test": "echo \"Error: no test specified\" && exit 1"
10     },
11    "keywords": [],
12    "author": "",
13    "license": "ISC",
14    "devDependencies": {
15      "@prisma/client": "^5.11.0",
16      "prisma": "^5.11.0"
17    },
18    "dependencies": {
19      "express": "^4.18.3"
20    }
21 }
```

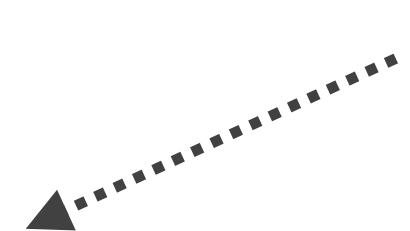
เพิ่มเติมคำสั่ง

สร้างไฟล์ schema.prisma

- ❖ สร้างไฟล์ชื่อว่า schema.prisma ใน project และใส่ข้อมูลเบื้องต้นของ DB ดังนี้

▷ schema.prisma > ...

```
1 generator client {
2   provider = "prisma-client-js"
3 }
4
5 datasource db {
6   provider = "mysql"
7   url      = env("DATABASE_URL")
8 }
```



จะนำข้อมูลที่กำหนดไว้ในไฟล์ .env มาใช้งาน

- ❖ สร้างไฟล์ชื่อว่า .env โดยใส่ข้อมูลเบื้องต้นในการเชื่อมต่อ DB ดังนี้

⚙ .env

```
1 DATABASE_URL=mysql://root:mysqladmin@localhost:3306/MiniStore
```

User: Password

Host: Port /Database Name

สร้างไฟล์ schema.prisma

- ❖ ในที่นี่ได้ทำการสร้าง Database ไว้เรียบร้อยแล้ว ดังนั้นเราจะทำการสร้าง Schema ของ DB โดยอัตโนมัติ เพื่อให้ Prisma รู้จักโครงสร้างของ Database ด้วยคำสั่ง

`prisma db pull`

- ❖ หลังจากคำสั่งทำงานสำเร็จไฟล์ schema.prisma จะมีข้อมูลโครงสร้างของ table ที่อยู่ภายใน DB ชื่อว่า MiniStore ประกอบขึ้น ประกอบด้วย Customers, Products, Orders, OrderDetail, และ Payments
- ❖ จากนั้นให้ทำการสร้าง Prisma Client เพื่อใช้สำหรับ MiniStore DB ภายใต้ Schema ที่ได้นี้โดยใช้คำสั่ง

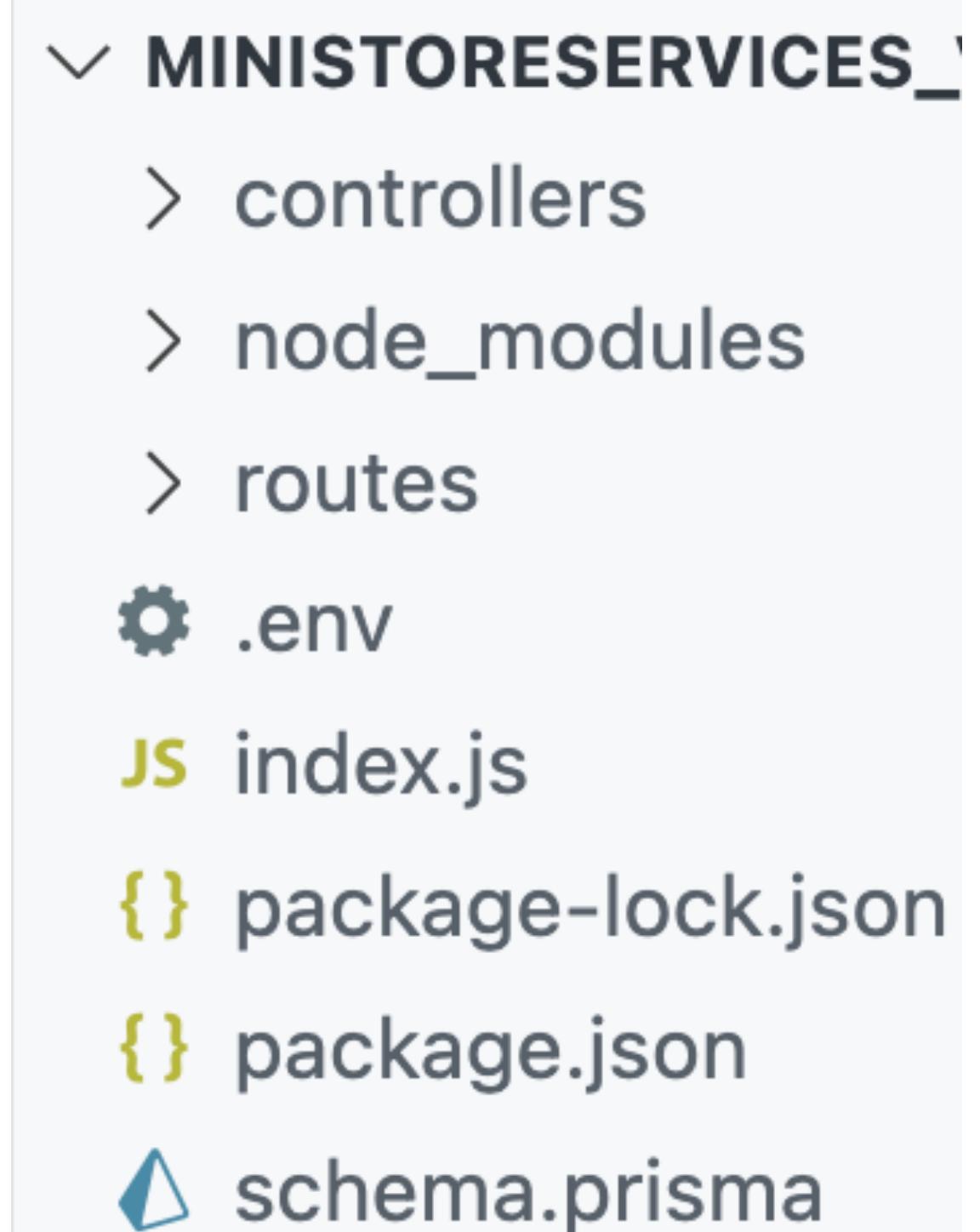
`prisma generate`

ตัวอย่างเนื้อหาในไฟล์ schema.prisma (ส่วนที่สร้างจากคำสั่ง prisma pull db)

```
10 model Customers {
11   customer_id Int      @id @default(autoincrement())
12   first_name  String   @db.VarChar(50)
13   last_name   String   @db.VarChar(50)
14   email       String?  @unique(map: "email") @db.VarChar(100)
15   address     String?  @db.VarChar(200)
16   phone_number String? @db.VarChar(20)
17   Orders      Orders []
18 }
19
20 model OrderDetail {
21   order_id    Int
22   product_id Int
23   quantity    Int
24   unit_price  Decimal @db.Decimal(10, 2)
25   Orders      Orders @relation(fields: [order_id], references: [order_id], on)
26   Products    Products @relation(fields: [product_id], references: [product_id]
27
28   @@id([order_id, product_id])
29   @@index([product_id], map: "product_id")
30 }
31
32 model Orders {
33   order_id    Int      @id @default(autoincrement())
34   customer_id Int
35   order_date   DateTime @default(now()) @db.DateTime(0)
36   order_status Orders_order_status?
37   total_price  Decimal @db.Decimal(10, 2)
38   OrderDetail  OrderDetail[]
39   Customers    Customers @relation(fields: [customer_id],
40   Payments     Payments []
41
42   @@index([customer_id], map: "customer_id")
43 }
44
45 model Payments {
46   payment_id   Int      @id @default(autoincrement())
47   order_id     Int
48   payment_option String?
49   payment_date DateTime @default(now()) @db.DateTime(0)
50   amount       Decimal @db.Decimal(10, 2)
51   remark       String?
52   payment_status Payments_payment_status?
53   Orders       Orders @relation(fields: [order_id], references:
54
55   @@index([order_id], map: "order_id")
56 }
57
58 model Products {
59   product_id   Int      @id @default(autoincrement())
60   name         String   @db.VarChar(100)
61   description  String? @db.Text
62   price        Decimal @db.Decimal(10, 2)
63   category     String? @db.VarChar(50)
64   image_url   String? @db.VarChar(255)
65   OrderDetail  OrderDetail[]
66 }
67
68 enum Orders_order_status {
69   processing
70   completed
71   cancelled
72 }
73
74 enum Payments_payment_status {
75   processing
76   completed
77   failed
78 }
```

เริ่มสร้าง API

- ❖ ทำการสร้าง Folder ชื่อว่า *controllers* และ *routes* ภายใน project
- ❖ ทำการสร้าง text ไฟล์ชื่อว่า *customers.js* ภายใน folder ชื่อ controllers
- ❖ เนื้อหาภายในไฟล์ *customers.js* จะประกอบด้วย API function ทั้งหมดที่ใช้ร่วมกับ ตาราง *customers* ในฐานข้อมูล



Customers API

- ❖ Post - ใช้เพิ่มข้อมูลใหม่
- ❖ Put - ใช้ update ข้อมูลเดิม
- ❖ Delete - ใช้ลบข้อมูลเดิม
- ❖ Get all - ใช้ดึงข้อมูลทั้งหมด
- ❖ Get only one by id - ใช้ดึงข้อมูลด้วย ID
- ❖ Get by search term - ใช้ค้นหาข้อมูลด้วยคำค้นอื่น ๆ เช่น เบอร์โทรศัพท์ อีเมล เป็นต้น

```
controllers > JS customers.js > ...
1  const { PrismaClient } = require('@prisma/client')
2  const prisma = new PrismaClient();
3
4  // insert one customer
5  > const createCustomer = async (req, res) => {
6    ...
7  };
8
9  // update one customer
10 > const updateCustomer = async (req, res) => {
11   ...
12 };
13
14 // delete customer by customer_id
15 > const deleteCustomer = async (req, res) => {
16   ...
17 };
18
19 // get all customers
20 > const getCustomers = async (req, res) => {
21   ...
22 };
23
24 // get only one customer by customer_id
25 > const getCustomer = async (req, res) => {
26   ...
27 };
28
29 // search any customer by name
30 > const getCustomersByTerm = async (req, res) => {
31   ...
32 };
33
34 module.exports = {
35   createCustomer, getCustomer, getCustomers,
36   updateCustomer, deleteCustomer, getCustomersByTerm
37 };
38
```

Customers API

- ❖ ตัวอย่าง Code เพิ่มเติมใน API ส่วนของ *Post* เพื่อเพิ่มข้อมูลลูกค้า มีดังนี้

```
4 // insert one customer
5 const createCustomer = async (req, res) => {
6   const { customer_id, first_name, last_name, address, email, phone_number } = req.body;
7   try {
8     const cust = await prisma.customers.create({
9       data: {
10         customer_id,
11         first_name,
12         last_name,
13         address,
14         email,
15         phone_number
16       }
17     });
18     res.status(200).json(cust);
19   } catch (err) {
20     res.status(500).json(err);
21   }
22 };
```

Customers API

- ❖ ตัวอย่าง Code ส่วนของ *Put* เพื่อใช้สำหรับแก้ไขข้อมูลลูกค้า มีดังนี้

```
22 // update one customer
23 const updateCustomer = async (req, res) => {
24   const { id, first_name, last_name, address, email, phone_number } = req.body;
25   try {
26     const cust = await prisma.customers.update({
27       data: {
28         first_name,
29         last_name,
30         address,
31         email,
32         phone_number
33       },
34       where: { customer_id: Number(id) }
35     });
36     res.status(200).json(cust);
37   } catch (err) {
38     res.status(500).json(err);
39   }
40 };
```

Customers API

- ❖ ตัวอย่าง Code ส่วนของ *Delete* เพื่อใช้สำหรับลบข้อมูลลูกค้า มีดังนี้

```
41 // delete customer by customer_id
42 const deleteCustomer = async (req, res) => {
43   const id = req.params.id;
44   try {
45     const cust = await prisma.customers.delete({
46       where: {
47         customer_id: Number(id),
48       },
49     })
50     res.status(200).json(cust)
51   } catch (err) {
52     res.status(500).json(err);
53   }
54 }
```

Customers API

- ❖ ตัวอย่าง Code ส่วนของ *Get* เพื่อเรียกดูข้อมูลลูกค้า มีดังนี้ (กรณีดึงข้อมูลทั้งหมด)

```
55 // get all customers
56 const getCustomers = async (req, res) => {
57   const custs = await prisma.customers.findMany()
58   res.json(custs)
59 };
```

Customers API

- ❖ ตัวอย่าง Code ส่วนของ *Get* เพื่อเรียกดูข้อมูลลูกค้าเป็นราย id

```
60 // get only one customer by customer_id
61 const getCustomer = async (req, res) => {
62   const id = req.params.id;
63   try {
64     const cust = await prisma.customers.findUnique({
65       where: { customer_id: Number(id) },
66     });
67     if (!cust) {
68       res.status(404).json({ 'message': 'Customer not found!' });
69     } else {
70       res.status(200).json(cust);
71     }
72   } catch (err) {
73     res.status(500).json(err);
74   }
75 };
```

Customers API

- ❖ ตัวอย่าง Code ส่วนของ *Get* เพื่อเรียกดูข้อมูลลูกค้าตามคำค้นที่หลากหลาย
- ❖ ในตัวอย่างผู้เรียกใช้จะสามารถระบุคำที่ต้องการค้นเป็นข้อมูลใดก็ได้อよ่างไร ได้อย่างหนึ่ง ประกอบด้วย ชื่อ หรือ อีเมล์
- ❖ การค้นจะเข้าไปเปรียบเทียบค่าตามฟิลด์ที่ระบุทีละฟิลด์ หากมีข้อมูลสองคลองที่ฟิลด์ใด ก็จะถือว่าพบข้อมูลและส่งค่ากลับไปยังผู้เรียก หากไม่พบข้อมูลจากฟิลด์ทั้งหมด จะส่งค่ากลับว่าไม่พบข้อมูล

```
77 const getCustomersByTerm = async (req, res) => {
78   const searchString = req.params.term;
79   try {
80     const custs = await prisma.customers.findMany({
81       where: {
82         OR: [
83           {
84             first_name: {
85               contains: searchString
86             }
87           },
88           {
89             email: {
90               contains: searchString
91             }
92           }
93         ],
94       },
95     });
96     if (!custs || custs.length == 0) {
97       res.status(404).json({ 'message': 'Customer not found!' });
98     } else {
99       res.status(200).json(custs);
100     }
101   } catch (err) {
102     res.status(500).json(err);
103   }
104 }
```

Contain = พبคำค้นเพียงบางส่วนก็ถือว่าพบข้อมูล

สร้าง API Router เพื่อเรียกใช้ Customers Services

- ❖ สร้างไฟล์ชื่อว่า api.js ไว้ใน Folder ชื่อว่า Routes และทำการเขียนโปรแกรม เพื่อให้ Router ซึ่งไปยัง Services ที่ต้องการ ดังต่อไปนี้

routes > **JS** api.js > ...

```
1 const express = require('express');
2 const router = express.Router();
3 const customerController = require('../controllers/customers');

4

5 router.post('/customers', customerController.createCustomer);
6 router.put('/customers', customerController.updateCustomer);
7 router.delete('/customers/:id', customerController.deleteCustomer);
8 router.get('/customers/:id', customerController.getCustomer);
9 router.get('/customers/q/:term', customerController.getCustomersByTerm);
10 router.get('/customers', customerController.getCustomers);

11

12 module.exports = router;
13
```

สร้าง Server เพื่อเปิดให้บริการ API Services

- ❖ แก้ไขโปรแกรมภาษาในไฟล์ index.js ดังต่อไปนี้

JS index.js > ...

```
1  const express = require('express');
2  const apiRouter = require('./routes/api');
3
4  const app = express();
5
6  // express version 4.16.0+ has built-in JSON parsing capabilities
7  // using below statement for json body parser
8  app.use(express.json());
9  app.use('/api/v1', apiRouter);
10
11 const port = 8800;
12
13 app.listen(port, () => {
14   console.log("Server listening on port:" + port);
15 }) ;
```

Summary (Customers API)

- ❖ สรุป API ทั้งหมดที่ได้สร้างสำหรับจัดการกับข้อมูล Customers ประกอบด้วย 6 APIs
 - ❖ เพิ่ม
 - ❖ แก้ไข
 - ❖ ลบ
 - ❖ เรียกดูทั้งหมด
 - ❖ เรียกดูเฉพาะ ID ที่ระบุ
 - ❖ ค้นหาข้อมูลตามค่าค้นต่อไปนี้ได้แก่ ชื่อ และ อีเมล์ เป็นต้น

JS index.js > ...

```
1 const express = require('express');
2 const apiRouter = require('./routes/api');
3
4 const app = express();
5
6 // express version 4.16.0+ has built-in JSON parsing capabilities
7 // using below statement for json body parser
8 app.use(express.json());
9 app.use('/api/v1', apiRouter);
10
11 const port = 8800;
12
13 app.listen(port, () => {
14   console.log("Server listening on port:" + port);
15 })
```

ตัวอย่าง url ที่ต้องเรียกใช้

http://localhost:8800/api/v1/customers

ชื่อของ service ที่ระบุไว้
ในโปรแกรม

ทดสอบ API โดยใช้ Postman

- ❖ ทำการรัน Server โดยใช้คำสั่ง *npm start* บน Terminal ของ VSCode
- ❖ เปิดโปรแกรม Postman และทดสอบการเรียกใช้การเพิ่มข้อมูลผ่าน Post

The screenshot shows the Postman application interface. At the top, there's a status bar with a pen icon, the URL 'POST http://localhost:8800/api/v1/customers', and a '+' button. Below the status bar, the main window has a header with 'HTTP' and the URL 'http://localhost:8800/api/v1/customers'. On the right side of the header are 'Save' and 'Send' buttons. The main area contains a form with the method 'POST' selected, the URL 'http://localhost:8800/api/v1/customers', and a 'Body' tab selected. Under the 'Body' tab, the 'JSON' option is chosen. In the JSON editor below, there is a code block containing the following JSON data:

```
1 {  
2   "first_name": "CustomerName1",  
3   "last_name": "Surname1",  
4   "email": "customer1@gmail.com",  
5   "address": "N/A",  
6   "phone_number": "0891111111"  
7 }
```

The screenshot shows the VSCode terminal window with the title bar 'PROBLEMS', 'OUTPUT', 'TERMINAL' (which is underlined), 'PORTS', and 'DEBUG CONSOLE'. The terminal output shows the command 'npm start' being run, followed by the output of nodemon starting the server on port 8800.

```
ku@kus-mbp-2 MiniStoreServices % npm start  
> mystoreservices@1.0.0 start  
> nodemon index.js  
  
[nodemon] 3.1.0  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): **  
[nodemon] watching extensions: js,mjs,cjs,json  
[nodemon] starting `node index.js`  
Server listening on port:8800
```

*ให้ใช้ Postman ทดสอบ API ส่วนที่เหลือให้ครบถ้วน

API Versioning

- ❖ เมื่อมีความต้องการแก้ไข API เดิม โดยอาจจะเป็นการเพิ่มความสามารถใหม่ๆ ขึ้น หรือต้องการเปลี่ยนแปลงการเรียกใช้งานอย่าง แต่ API เดิมยังคงมีผู้ใช้งานอยู่ จะทำอย่างไร เพื่อไม่ให้เกิดผลกระทบกับผู้ใช้เดิม
- ❖ แนวทางที่นิยมใช้คือการสร้าง API ใหม่โดยระบุเลข Version ไว้ในส่วนของ Endpoint โดยผู้ใช้สามารถเลือกได้ว่าจะเรียกใช้ Version ใด หากต้องการใช้ของเดิมก็ยังสามารถใช้ได้อยู่โดยไม่ต้องแก้ไขโปรแกรม แต่หากต้องการใช้งานตัวใหม่ ก็จะต้องยอมแก้ไขโปรแกรมเพื่อเปลี่ยนมาเรียกใช้ Endpoint ตัวใหม่ เป็นต้น

ตัวอย่าง API Versioning

- ❖ สมมติว่ามีการเพิ่มเติมความสามารถใหม่ให้กับ Service การค้นหาเดิมที่ชื่อ `getCustomerByTerm` โดยให้สามารถค้นหาจากนามสกุล และเบอร์โทรศัพท์ได้ด้วย โดยผู้ใช้เดิมจะต้องไม่ได้รับผลกระทบ มีแนวทางดังนี้
- ❖ สร้างโปรแกรมใหม่ชื่อว่า `customersv2.js` ภายใต้โฟลเดอร์ `controllers`
- ❖ เขียนโปรแกรมใหม่โดยใช้ชื่อ `function` เดิม ดัง Slide ถัดไป

ตัวอย่าง โปรแกรมภาษา Node.js สำหรับ API ดูแลลูกค้าในไฟล์ customersv2.js

```
controllers > JS customersv2.js > ...
1  const { PrismaClient } = require('@prisma/client')
2  const prisma = new PrismaClient();
3
4  // search any customer by name
5  const getCustomersByTerm = async (req, res) => {
6    const searchString = req.params.term;
7    try {
8      const custs = await prisma.customers.findMany({
9        where: {
10          OR: [
11            {
12              first_name: {
13                contains: searchString
14              }
15            },
16            {
17              last_name: {
18                contains: searchString
19              }
20            },
21            {
22              email: {
23                contains: searchString
24              }
25            },
26            {
27              phone_number: {
28                contains: searchString
29              }
30            }
31          ],
32        }
33      });
34      if (!custs || custs.length == 0) {
35        res.status(404).json({ 'message': 'Customer not found!' });
36      } else {
37        res.status(200).json(custs);
38      }
39    } catch (err) {
40      res.status(500).json(err);
41    }
42  };

```

ตัวอย่าง API Versioning (ต่อ)

- ❖ สร้างไฟล์ใหม่ชื่อว่า apiv2.js ภายในโฟลเดอร์ Routes

routes > **JS** apiv2.js > ...

```
1 const express = require('express');
2 const router = express.Router();
3 const customerControllerv2 = require('../controllers/customersv2');
4
5 router.get('/customers/q/:term', customerControllerv2.getCustomersByTerm);
6
7 module.exports = router;
```

ตัวอย่าง API Versioning (ต่อ)

- ❖ แก้ไขไฟล์ index.js เดิมให้สามารถเรียกใช้บริการที่ปรับปรุงใหม่

JS index.js > ...

```
1  const express = require('express');
2  const apiRouter = require('./routes/api');
3  const apiv2Router = require('./routes/apiv2');

4
5  const app = express();

6
7  // express version 4.16.0+ has built-in JSON parsing capabilities
8  // using below statement for json body parser
9  app.use(express.json());
10 app.use('/api/v1', apiRouter);
11 app.use('/api/v2', apiv2Router);

12
13 const port = 8800;

14
15 app.listen(port, () => {
16   console.log("Server listening on port:" + port);
17 }) ;
```

ทดสอบ

- ❖ เมื่อเรียกใช้ Endpoint เดิมจะไม่สามารถหาข้อมูลจาก เบอร์โทรศัพท์ หรือ นามสกุลได้

```
http://localhost:8800/api/v1/customers/q/08922
```

- ❖ เมื่อเรียกใช้ Endpoint ใหม่จะสามารถหาข้อมูลได้เพิ่มเติม จากเบอร์โทรศัพท์ หรือนามสกุล

```
http://localhost:8800/api/v2/customers/q/08922
```

API Rate Limiting

- ❖ เมื่อมีการเรียกใช้ API ติด ๆ กันมากเกินไปอาจส่งผลให้เครื่อง Server ที่ให้บริการ และเครื่องที่รับระบบฐานข้อมูลทำงานหนัก อาจส่งผลไม่ดีต่อระบบ โดยรวม เราจะสามารถบรรเทาปัญหานี้ได้อย่างไรบ้าง
- ❖ **จำกัดจำนวนการเข้าถึงด้วยการกำหนดไว้ในโปรแกรมของ Server**
- ❖ **จำกัดจำนวนการเข้าถึง API โดยใช้วิธีการจัดการของ Server เช่น การสร้าง Connection Pool เป็นต้น**
- ❖ เพิ่มเครื่องให้บริการและใช้แนวคิดของ Load Balancing เข้ามาช่วย
- ❖ อื่น ๆ

API Rate Limiting

- ❖ การจำกัดการเข้าถึงด้วยวิธีการกำหนดไว้ในโปรแกรมของ Server
- ❖ ทำการติดตั้ง Library ชื่อว่า *express-rate-limit* เพิ่มเติม

```
18 "dependencies": {  
19   "express": "^4.18.3",  
20   "express-rate-limit": "^7.2.0",
```

ภายในไฟล์ *package.json*

- ❖ สร้าง option ของการจำกัดเพิ่มเติม เช่น

```
const rateLimit = require('express-rate-limit');  
const apiLimiter = rateLimit({  
  windowMs: 1000*60*3,    // 3 minutes  
  max: 10,  
  message: 'Too many requests, please try again after 3 minutes!'  
});
```

ตัวอย่าง *Limit* ที่ระบุด้วย *Code* นี้หมายถึง ยอมให้มีการเรียกใช้ *API* บน *Server* นี้จำนวนสูงสุดไม่เกิน 10 ครั้งภายใน 3 นาที หากเกินกว่าที่กำหนดไว้ ผู้เรียกจะต้องรอ อีก 3 นาทีจึงจะสามารถเรียกใช้ *API* ได้ เป็นต้น

API Rate Limiting (ต่อ)

- ❖ การนำไปใช้ ให้แก่ไขเพิ่มเติมไฟล์ api.js ที่อยู่ในโฟลเดอร์ Routes ดังนี้

```
1 const express = require('express');
2 const rateLimit = require('express-rate-limit');
3 const apiLimiter = rateLimit({
4   windowMs: 1000*60*3,    // 3 minutes
5   max: 10,
6   message: 'Too many requests, please try again after 3 minutes!'
7 });
8 const router = express.Router();
9 const customerController = require('../controllers/customers');
10
11 router.post('/customers', apiLimiter, customerController.createCustomer);
12 router.put('/customers', apiLimiter, customerController.updateCustomer);
13 router.delete('/customers/:id', apiLimiter, customerController.deleteCustomer);
14 router.get('/customers/:id', customerController.getCustomer);
15 router.get('/customers/q/:term', apiLimiter, customerController.getCustomersByTerm);
16 router.get('/customers', apiLimiter, customerController.getCustomers);
17
18 module.exports = router;
```

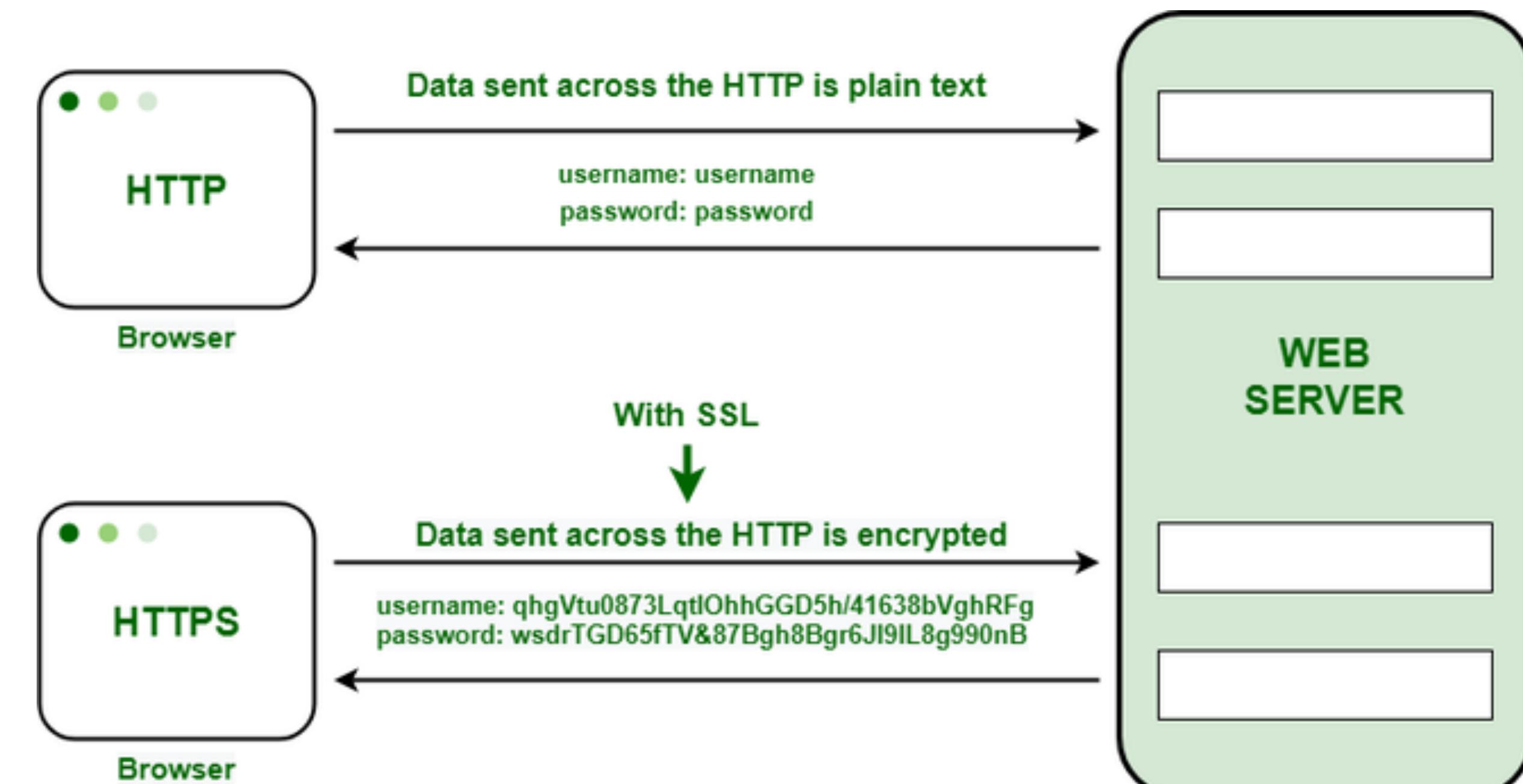
นำ apiLimiter ไปใส่เพิ่มเติมใน endpoint ที่เราต้องการจำกัดจำนวนครั้งในการเข้าถึง หากไม่ได้กำหนด apiLimiter ผู้ใช้จะสามารถเรียกใช้ endpoint นั้นได้ไม่จำกัด

ในตัวอย่างนี้ ไม่ได้ใส่ apiLimiter ให้กับ api การเรียกดูข้อมูลของลูกค้าแบบระบุ id ซึ่ง endpoint นี้จะเรียกใช้อย่างไรก็ได้ไม่ถูกจำกัด

ทดสอบ

API Services บน HTTPS

- ❖ การให้บริการ API บน HTTPS: เพื่อสร้างการเข้ามายังการสื่อสารที่ปลอดภัยโดยจัดให้มีการเข้ารหัสระหว่างการรับส่งข้อมูล
- ❖ มีการใช้ใบรับรองอิเล็กทรอนิกส์ เพื่อยืนยันตัวตนและเข้ารหัส
- ❖ ใบรับรองที่นำเข้าเชื่อถือจะออกโดย certificate authority (CA) เป็นหน่วยงานในการออก ใบรับรองอิเล็กทรอนิกส์ (Certificate) เพื่อใช้ระบุตัวบุคคล หรือองค์กร โดยผู้ให้บริการออกใบรับรองจะทำการตรวจสอบข้อมูลของเจ้าของใบรับรองและจะทำการรับรองกุญแจสาธารณะ (Public Key) กับข้อมูลดังกล่าวที่ได้ผ่านการตรวจสอบอย่างน่าเชื่อถือ ออกเป็นใบรับรองอิเล็กทรอนิกส์ เพื่อใช้ในการทำธุกรรมทางอิเล็กทรอนิกส์

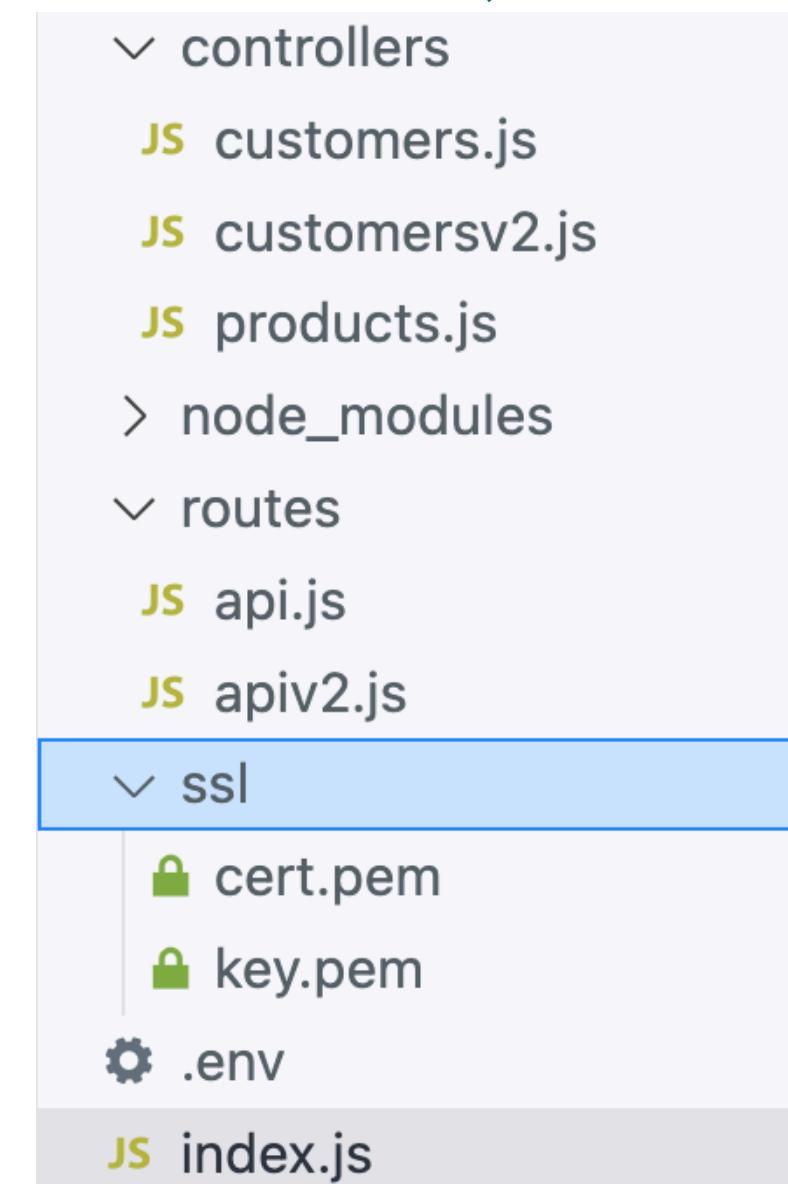


ภาพจาก <https://www.geeksforgeeks.org>

API Services บน HTTPS

- ❖ แนวทางการให้บริการ API บน HTTPS
- ❖ ทำการสร้างใบรับรองเพื่อใช้ในการยืนยันตัวตนและการเข้ารหัส ในที่นี้จะขอใช้การสร้างใบรับรองด้วยตนเองเพื่อความสะดวกในการทดลอง โดยใช้เครื่องมือ *openssl*
- ❖ คำสั่งที่ใช้ในการสร้างใบรับรองแสดงในรอบสีแดง เมื่อสร้างเสร็จเรียบร้อย ให้นำไฟล์ key.pem และ cert.pem ไปวางไว้ในโฟลเดอร์ ssl ภายใต้ project (ให้สร้างโฟลเดอร์ชื่อว่า ssl ขึ้นมาใหม่)

```
9 // การสร้างใบรับรอง ให้ใช้ command line ดังนี้
10 /**
11 openssl genrsa -out key.pem 2048
12 openssl req -new -key key.pem -out csr.pem
13 openssl x509 -req -days 365 -in csr.pem -signkey key.pem -out cert.pem
14 */
```



API Services บน HTTPS

- ❖ แก้ไข code เพิ่มเติม เพื่อใช้งาน key และใบรับรองที่สร้างขึ้น
- ❖ ทำการสร้างบริการให้ server สามารถเรียกใช้ได้ทั้ง http และ https ด้วยการแยก port ในการให้บริการ ..หากต้องการใช้งาน https เพียงอย่างเดียว ก็ไม่จำเป็นต้องเปิด port สำหรับ http

js index.js > ...

```
1 const express = require('express');
2 const apiRouter = require('./routes/api');
3 const apiv2Router = require('./routes/apiv2');
4 const https = require('https');
5 const fs = require('fs');
6 const app = express();

7
8 // express version 4.16.0+ has built-in JSON parsing capabilities
9 // using below statement for json body parser
10 app.use(express.json());
11 app.use('/api/v1', apiRouter);
12 app.use('/api/v2', apiv2Router);

13
14 const ssl_options = {
15   key: fs.readFileSync('./ssl/key.pem'),
16   cert: fs.readFileSync('./ssl/cert.pem')
17 }

18 const port = 8800;
19 const secure_port = 8443;
20 app.listen(port, () => {
21   console.log("Server listening on port:" + port);
22 });
23 https.createServer(ssl_options, app).listen(secure_port, () => {
24   console.log("HTTPS Server listening on port:" + secure_port);
25 });


```

ทดสอบ

การสร้าง API Documentation

- ❖ แนวทางการสร้างเอกสารเพื่อใช้อธิบายการเรียกใช้ API โดยใช้เครื่องมือชื่อว่า Swagger

The screenshot shows the Swagger UI interface for the MiniStore API. At the top, it displays the title "MiniStore API" with version "1.0.0" and "OAS 3.0". Below the title, there is a subtitle "Learning to build Rest API using NodeJS". A "Servers" dropdown menu is set to "http://localhost:8800 - My API through HTTP". The main content area is divided into sections: "Customers" and "Schemas". The "Customers" section contains two GET requests: one for "/api/v1/customers" (Get All Customers) and another for "/api/v1/customers/{id}" (Get Customer by ID). The "Schemas" section shows a JSON schema for the "Customer" type, which includes fields: customer_id, first_name, last_name, address, email, and phone_number.

```
Customer <pre>{</pre><pre>    customer_id</pre><pre>    first_name</pre><pre>    last_name</pre><pre>    address</pre><pre>    email</pre><pre>    phone_number</pre><pre>}</pre>
```

การใช้งาน Swagger

- ❖ เราใช้ Swagger ในการทำ Document ของ API เพื่ออธิบายถึงวิธีการเรียกใช้งาน รวมทั้งสามารถใช้ทดสอบ API นั้น ๆ ได้ด้วย
- ❖ สำหรับ NodeJS เมื่อเราต้องการใช้งาน Swagger จะต้องติดตั้งไลบรารีเพิ่มเติม 2 ตัวได้แก่ swagger-ui-express และ swagger-jsdoc
- ❖ การติดตั้งสามารถใช้คำสั่ง npm ในการติดตั้ง ดังนี้

```
> npm install swagger-ui-express swagger-jsdoc
```

เริ่มการใช้งาน Swagger

- ❖ สร้างไฟล์ชื่อ `swagger.js` เพื่อกำหนด spec และ options ต่าง ๆ ของโปรเจค
- ❖ ในตัวอย่างนี้จะสร้างไว้ระดับเดียวกับ `index.js`
- ❖ รายละเอียดภายในไฟล์ `swagger.js` แสดงด้านขวา

```
> controllers  
> node_modules  
✓ routes  
  JS api.js  
  JS apiv2.js  
> ssl  
  .env  
 JS index.js  
 {} package-lock.json  
 {} package.json  
 🔧 schema.prisma  
 JS swagger.js
```

ระบุ path ของ api ให้สอดคล้อง กับโครงสร้าง โปรเจคของเรา

```
JS swagger.js > ...  
1  const swaggerUI = require('swagger-ui-express');  
2  const swaggerJSDoc = require('swagger-jsdoc');  
3  
4  // set swagger options  
5  const swaggerOptions = {  
6    definition: {  
7      openapi: '3.0.0', // Specify the OpenAPI version  
8      info: {  
9        title: 'MiniStore API', // API's title  
10       version: '1.0.0', // API's version  
11       description: 'Learning to build Rest API using NodeJS',  
12     },  
13     servers: [ // Add server details  
14       { url: 'http://localhost:8800',  
15         description: "My API through HTTP"  
16       }, // server URL  
17       { url: 'https://localhost:8443',  
18         description: "My API through HTTPS"  
19       }, // server URL  
20     ],  
21   },  
22   apis: ['./routes/*.js'], // Path to all API definitions  
23 };  
24 const swaggerSpecs = swaggerJSDoc(swaggerOptions);  
25  
26 module.exports = { swaggerSpecs, swaggerUI };
```

เริ่มการใช้งาน Swagger (ต่อ)

- ❖ นำ swagger.js ไปใช้ โดยระบุไว้ใน index.js ดังนี้

JS index.js > ...

```
1  const express = require('express');
2  const cors = require('cors');
3  const { swaggerSpecs, swaggerUI } = require('./swagger');
4  const apiRouter = require('./routes/api');
5  const apiv2Router = require('./routes/apiv2');
6  const https = require('https');
7  const fs = require('fs');
8  const app = express();
9
10 // express version 4.16.0+ has built-in JSON parsing capabilities
11 // using below statement for json body parser
12 app.use(express.json());
13 app.use(cors());
14 app.use('/api/v1', apiRouter);
15 app.use('/api/v2', apiv2Router);
16 app.use("/api-docs", swaggerUI.serve, swaggerUI.setup(swaggerSpecs));
17
```

เริ่มการใช้งาน Swagger (ต่อ)

- ❖ ใส่คำอธิบาย ด้วยโครงสร้างที่ swagger กำหนดไว้ในไฟล์ .js ที่ต้องการทำ document ในที่นี่จะขอยกตัวอย่างไฟล์ api.js ในส่วนของ api การอ่านข้อมูล customers จาก DB

```
/**  
 * @swagger  
 * /api/v1/customers:  
 *   get:  
 *     summary: Get All Customers  
 *     tags: [Customers]  
 *     description: Returns a list of all customers in the database.  
 *     responses:  
 *       200:  
 *         description: A list of customers.  
 *         content:  
 *           application/json:  
 *             schema:  
 *               type: array  
 *               items:  
 *                 $ref: '#/components/schemas/Customer'  
 *       500:  
 *         description: Internal server error.  
 */
```

โครงสร้างของ comment นี้เป็นรูปแบบของ yaml ซึ่งค่อนข้างเข้มงวด โดยเฉพาะการเย็บองต่าง ๆ หากผิดรูปแบบจะไม่สามารถแสดง API Document ได้

```
/*  
 * @swagger  
 * components:  
 *   schemas:  
 *     Customer:  
 *       type: object  
 *       properties:  
 *         customer_id:  
 *           type: integer  
 *           description: The unique identifier of the customer.  
 *         first_name:  
 *           type: string  
 *           description: The customer's firstname.  
 *         last_name:  
 *           type: string  
 *           description: The customer's lastname.  
 *         address:  
 *           type: string  
 *           description: The customer's address.  
 *         email:  
 *           type: string  
 *           description: The customer's email (unique).  
 *         phone_number:  
 *           type: string  
 *           description: The customer's phone number.  
 *         required:  
 *           - none  
 */
```

เริ่มการใช้งาน Swagger (ต่อ)

- ❖ ใส่คำอธิบาย เพิ่มเติมต่อจากเดิมภายในไฟล์ api.js สำหรับการเพิ่มข้อมูล customer โดยใช้ POST

```
/**  
 * @swagger  
 * /api/v1/customers:  
 *   post:  
 *     summary: Create a new Customer  
 *     tags: [Customers]  
 *     description: create a new customer on database  
 *     requestBody:  
 *       required: true  
 *       content:  
 *         application/json:  
 *           schema:  
 *             $ref: '#/components/schemas/Customer'  
 *     responses:  
 *       200:  
 *         description: Customer object created.  
 *         content:  
 *           application/json:  
 *             schema:  
 *               $ref: '#/components/schemas/Customer'  
 *       500:  
 *         description: Internal server error.  
 */
```

• • •

ทดสอบ

- หลังจากเปิด server เรียบร้อยแล้ว เราสามารถทดสอบการสร้าง API Documents ได้โดยเปิด Browser และไปที่ url ที่กำหนดไว้ใน option คือ

<http://localhost:8800/api-docs>

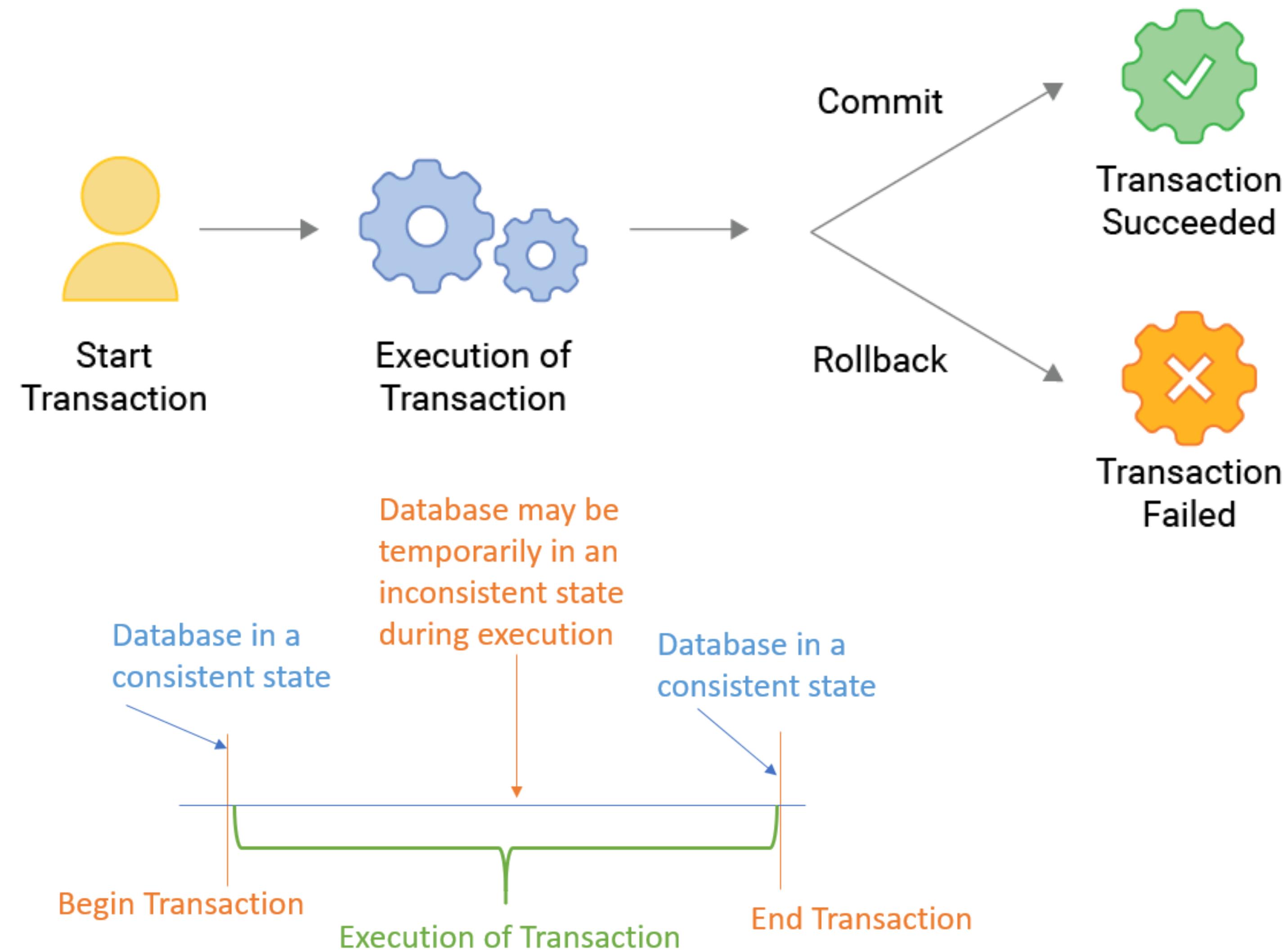
หรือ

<https://localhost:8443/api-docs>

หากเปิดด้วย Safari และไม่แสดงข้อมูล
ได ๆ ให้เปลี่ยนไปใช้ Chrome หรือ
Edge และลองใหม่อีกครั้ง

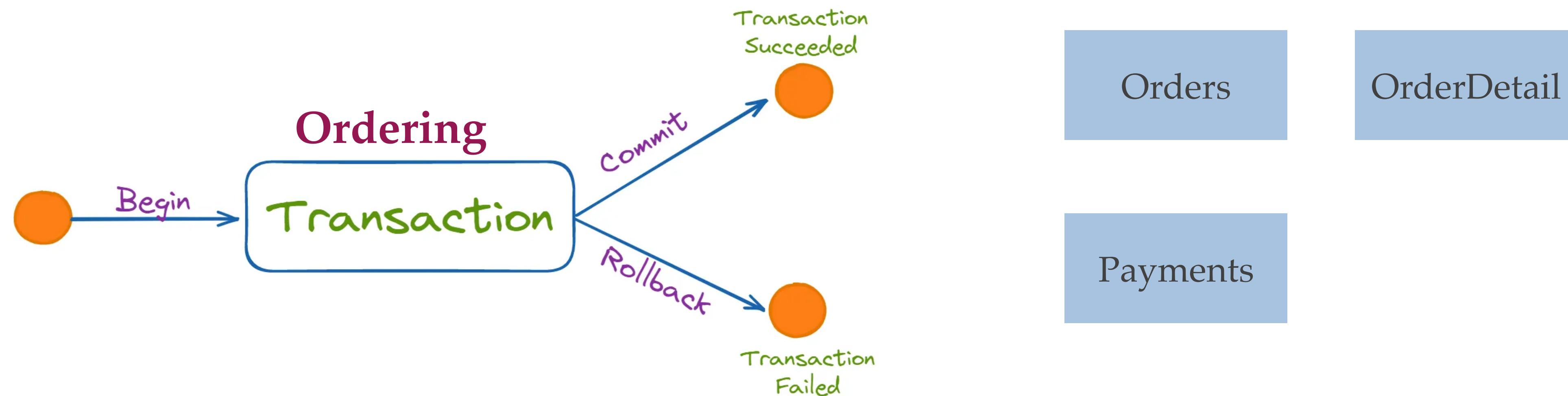
The screenshot shows the Swagger UI interface for the MiniStore API. At the top, it displays the title "MiniStore API 1.0.0 OAS 3.0" and a subtitle "Learning to build Rest API using NodeJS". Below this, there is a dropdown menu labeled "Servers" containing the URL "http://localhost:8800 - My API through HTTP". The main content area is divided into sections: "Customers" and "Schemas". The "Customers" section contains two GET requests: "/api/v1/customers" (Get All Customers) and "/api/v1/customers/{id}" (Get Customer by ID). The "Schemas" section shows a JSON schema for the Customer model, which includes fields like customer_id, first_name, last_name, address, email, and phone_number.

การใช้หลักการ Transactional ในการทำงานภายใน ได้ Prisma



การใช้งาน Transaction ด้วย Prisma

- ❖ **ตัวอย่าง:** เราจะเลือกใช้วิธีการนี้กับ API สำหรับการสั่งสินค้า ซึ่งจะต้องมีการบันทึกข้อมูลลงในตารางมากกว่า 1 ตาราง ได้แก่ orders, orderdetail และ payments ดังนั้นเราจะต้องมั่นใจว่าข้อมูลสามารถบันทึกลงในตารางต่างๆ ที่เกี่ยวข้องได้ครบถ้วน จึงจะยอมรับว่าการทำรายการสำเร็จ หากขาดตกไปในขั้นตอนใดก็ตามจะต้องย้อนกลับสิ่งที่ได้ทำไปแล้วทั้งหมด และแจ้งเตือนการทำรายการไม่สำเร็จ



การใช้งาน Transaction ด้วย Prisma (ต่อ)

- ❖ สมมติว่าหน้า Frontend ส่งข้อมูลการสั่งสินค้ามาให้กับ API มีโครงสร้างข้อมูล ดังต่อไปนี้ ซึ่งประกอบด้วย ข้อมูลผู้ซื้อ ข้อมูลสินค้า จำนวนการสั่ง ราคาต่อหน่วย และจำนวนเงินรวม

```
{  
  "orderId": 1001,  
  "customerId": 100,  
  "orderDate": "2024-05-31",  
  "items": [  
    {  
      "productId": 111,  
      "quantity": 2,  
      "unitPrice": 250.00  
    },  
    {  
      "productId": 222,  
      "quantity": 1  
      "unitPrice": 58.50  
    }  
  ],  
  "paymentMethod": "Credit Card",  
  "totalAmount": 558.50  
}
```

การใช้งาน Transaction ด้วย Prisma (ต่อ)

- ❖ สร้างไฟล์ชื่อว่า orders.js ในโฟล์เดอร์ controllers
- ❖ สร้าง api function เพื่อรองรับการส่งสินค้า ดังนี้

```
controllers > JS orders.js > ...
1  const { PrismaClient } = require('@prisma/client')
2  const prisma = new PrismaClient();
3
4  const createOrder = async (req, res) => {
5      const { orderId, customerId, orderDate, items, paymentMethod, totalAmount } = req.body;
6
7      if (!customerId || !items || !orderId || !paymentMethod || !totalAmount) {
8          return res.status(400).json({ message: 'Invalid order data' });
9      }
10     try {
11         const [order, payment] = await prisma.$transaction([
12             prisma.orders.create({ ... }),
13             prisma.payments.create({ ... })
14         ]);
15         console.log('Transaction committed:', { order, payment });
16         res.status(200).json('Order created successfully.');
17     } catch (err) {
18         console.error('Failed to create orders:', err.message);
19         res.status(500).json(err);
20     }
21
22     module.exports = { createOrder };
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

รายละเอียดโปรแกรมส่วนนี้อยู่หน้าถัดไป

การใช้งาน Transaction ด้วย Prisma (ต่อ)

controllers > **JS** orders.js > ...

```
1  const { PrismaClient } = require('@prisma/client')
2  const prisma = new PrismaClient();
3
4  const createOrder = async (req, res) => {
5      const { orderId, customerId, orderDate, items, paymentMethod, totalAmount } = req.body;
6
7      if (!customerId || !items || !orderId || !paymentMethod || !totalAmount) {
8          return res.status(400).json({ message: 'Invalid order data' });
9      }
10     try {
11         const [order, payment] = await prisma.$transaction([
12             prisma.orders.create({
13                 data: {
14                     order_id: orderId,
15                     customer_id: customerId,
16                     order_date: new Date(orderDate),
17                     order_status: 'processing',
18                     total_amount: totalAmount,
19                     OrderDetail: {
20                         create: items.map((item) => ({
21                             product_id: item.productId,
22                             quantity: item.quantity,
23                             unit_price: item.unitPrice,
24                         })),
25                     },
26                 },
27             },
28         );
29         prisma.payments.create({
30             data: {
31                 order_id: orderId,
32                 amount: totalAmount,
33                 payment_method: paymentMethod,
34                 payment_status: 'pending',
35                 amount: totalAmount
36             },
37         });
38         console.log('Transaction committed:', { order, payment });
39         res.status(200).json('Order created successfully.');
40     } catch (err) {
41         console.error('Failed to create orders:', err.message);
42         res.status(500).json(err);
43     }
44 };
45
46 module.exports = { createOrder };
47
```

