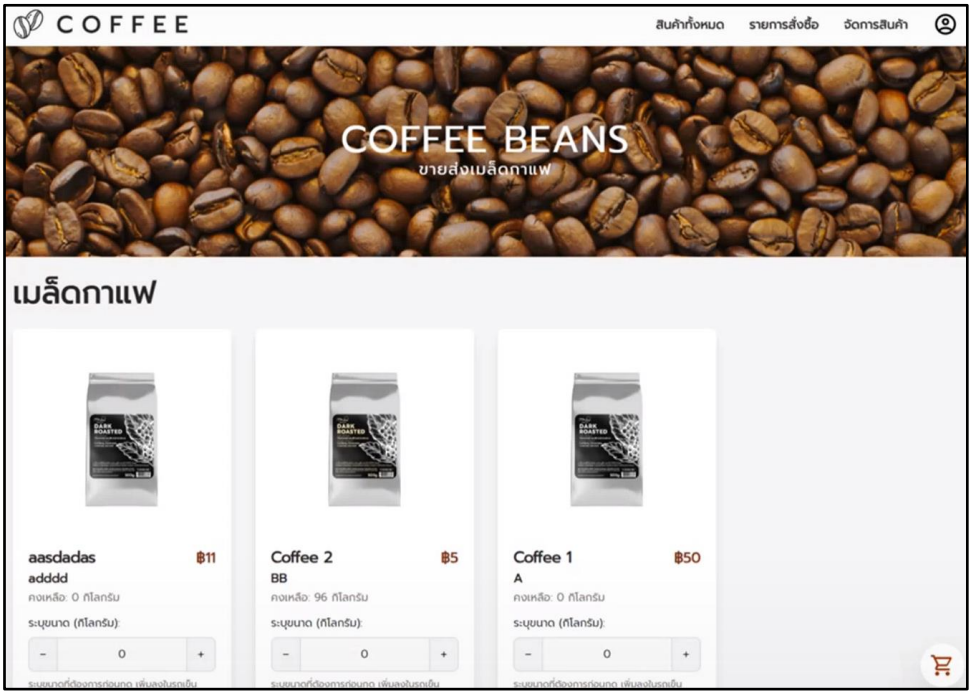
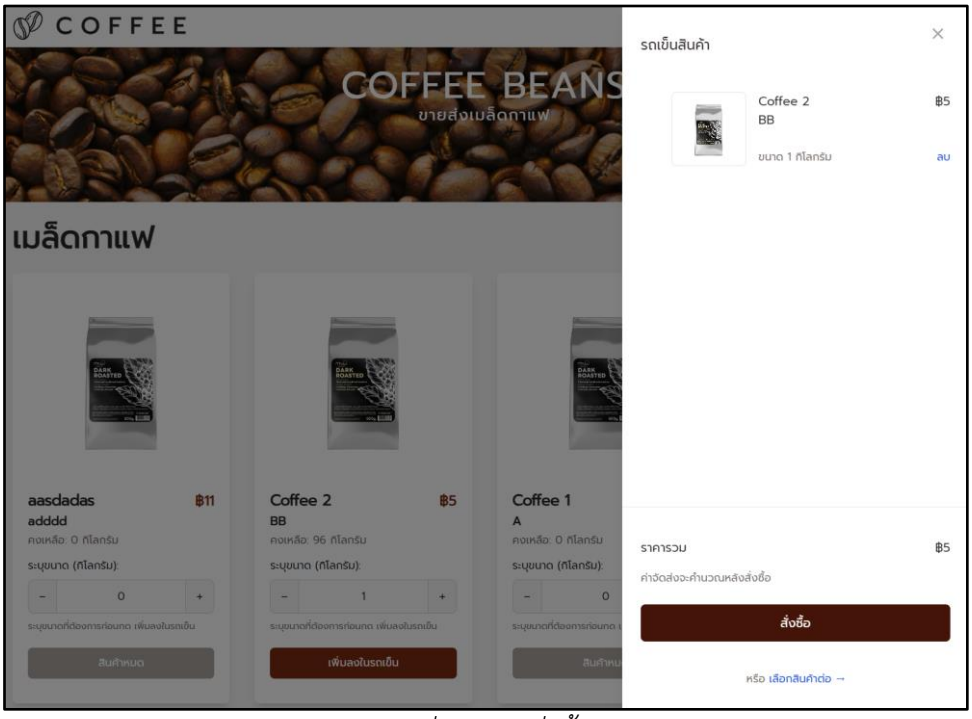
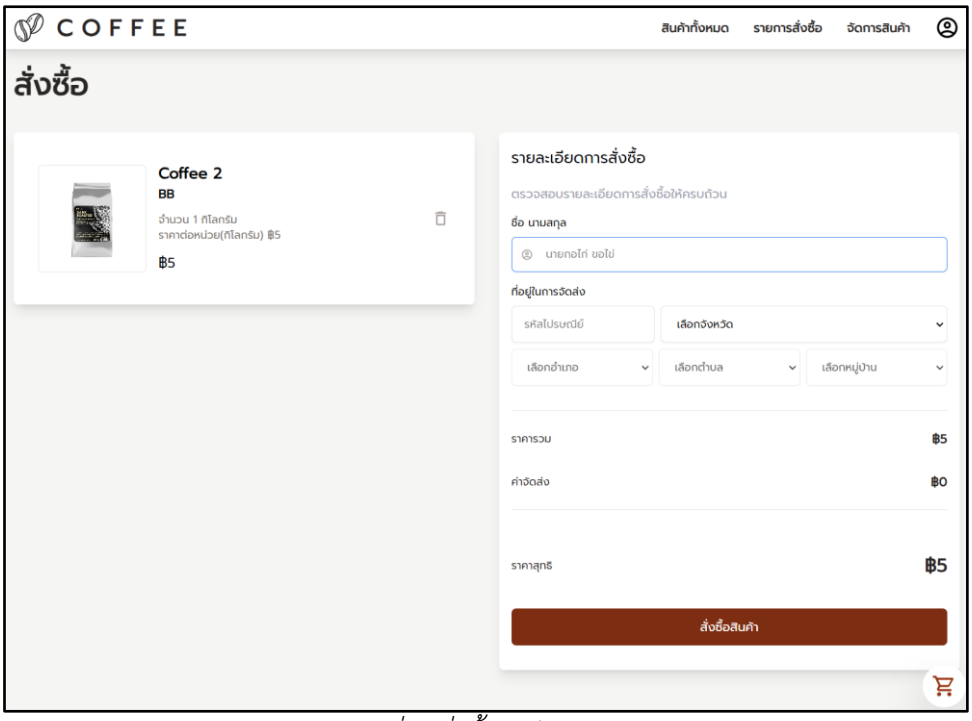
# ใบงาน



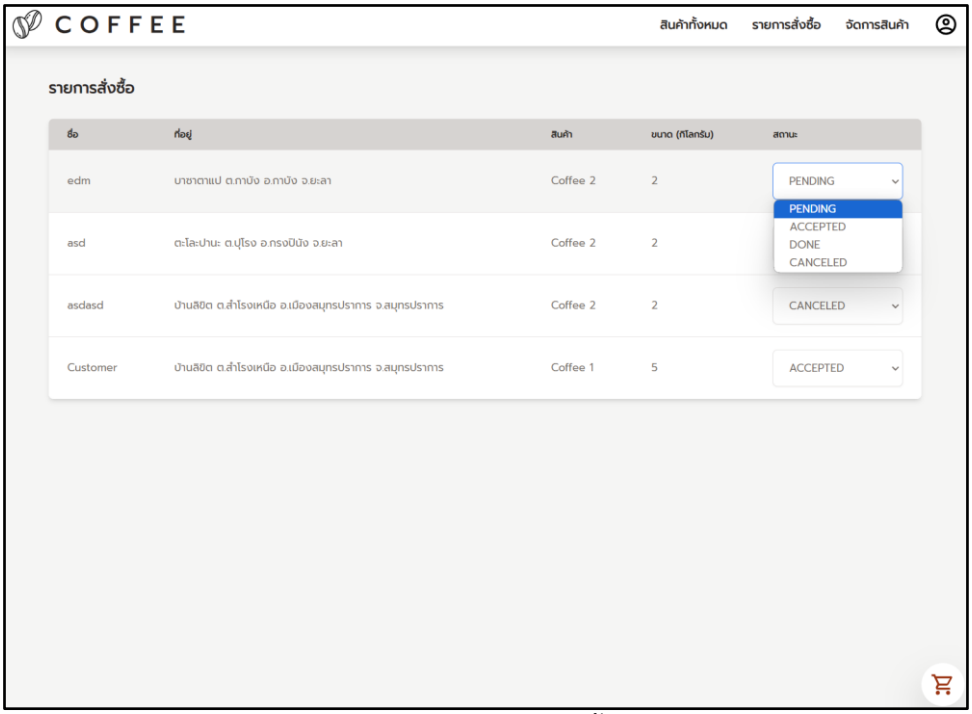

*รูปที่ 1 ตัวอย่างแอปพลิเคชันสั่งกาแฟ*




*รูปที่ 2 ส่งคำสั่งซื้อ*

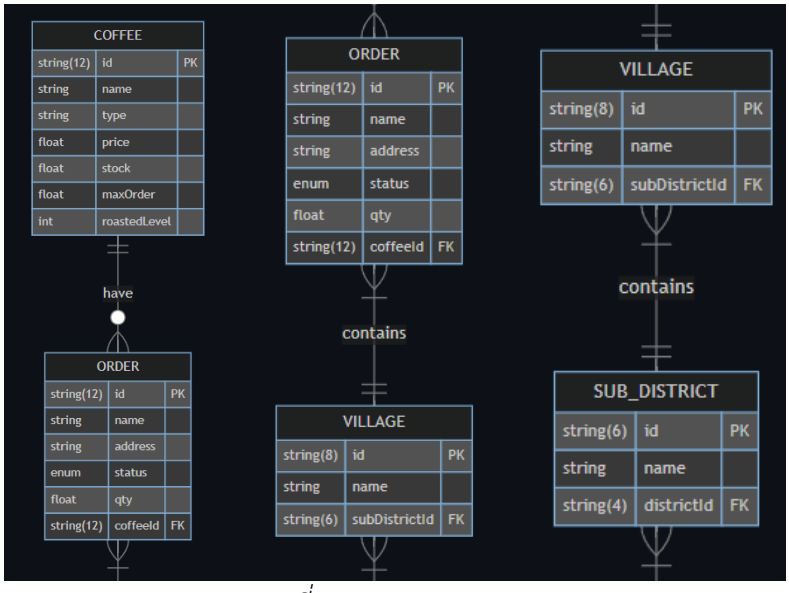



*รูปที่ 3 สั่งซื้อเมล็ดกาแฟ*

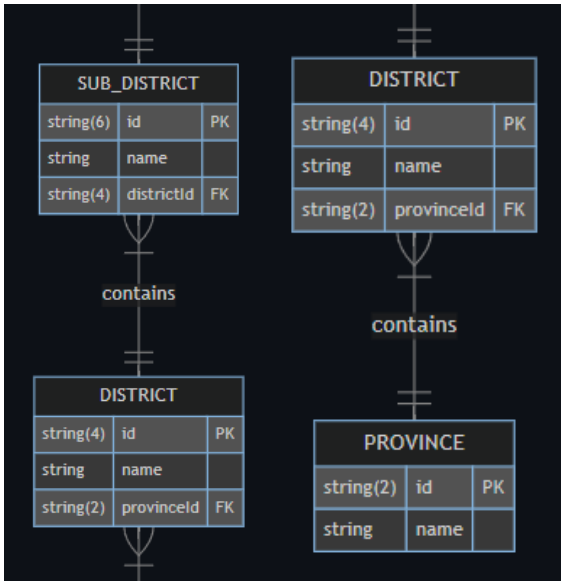



*รูปที่ 4 จัดการคำสั่งซื้อ*

ใบงานนี้จัดทำเพื่อให้เข้าใจหลักการทำงานของ Front-End, Back-End และการใช้งาน PrismaORM และควบคุม Racing Conditions ในขณะติดต่อกับฐานข้อมูลโดยใช้ Transaction เพื่อให้ข้อมูลสอดคล้องกัน เมื่อมีใช้งานพร้อมกัน โดยระบบจะรองรับระบบ เพิ่ม ลบ แก้ไข เมล็ดกาแฟ ส่งคำสั่งซื้อและจัดการคำสั่งซื้อ และใช้ Web Socket เพื่อให้สามารถแสดงผลการเปลี่ยนแปลงของข้อมูลให้เป็นปัจจุบัน โดยมี ER-Diagram ดังนี้




*รูปที่ 5 ER-Diagram (1)*




*รูปที่ 6 ER-Diagram (2)*

## เครื่องมือที่ใช้ในการดำเนินงาน

1. Database Server (MariaDB หรือ MySQL)
2. Node.js LTS, Deno, Bun
3. IDE หรือ Editor (VSCode, JetBrains IDE, Neovim w/ LSP)
4. Git CLI
5. PNPM (ไม่บังคับ - ติดตั้งได้โดย npm i -g pnpm หรือใช้ npm)

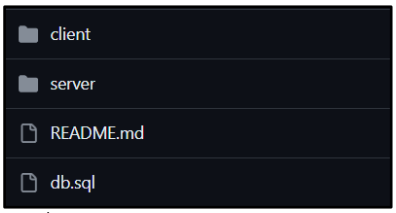
## วิธีการดำเนินงาน

1. ติดตั้งฐานข้อมูล โดยใน Windows สามารถใช้ XAMPP ได้ และในส่วนของ Linux เช่น Alpine, Ubuntu, Fedora หรือ OpenSUSE ก็สามารถติดตั้งได้โดยใช้ Package Manager ของ Distribution นั้นๆ เช่น APK, APT, DNF หรือ ZYPPER

```
$ sudo apt update
$ sudo apt install mariadb-server
$ sudo mysql_secure_installation
```

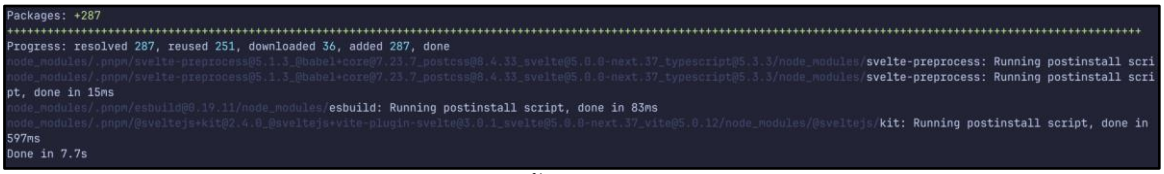
*รูปที่ 7 ตัวอย่างการติดตั้งโดยใช้ APT บน Debain-Based Distro (Debian / Ubuntu)*

2. โคลน GitHub Project โดยใช้ Git



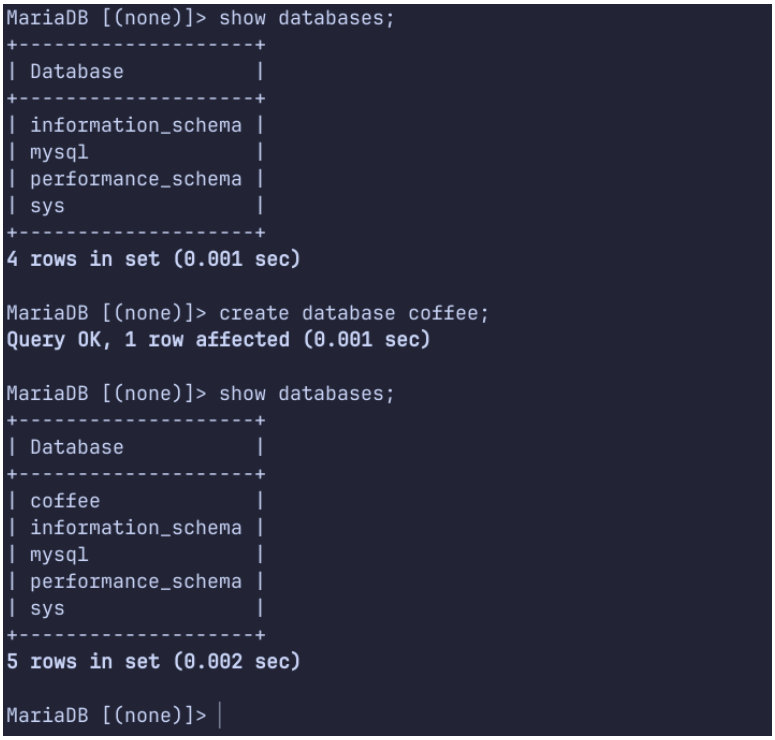

*รูปที่ 8 แฟ้มและไฟล์ภายใน Project*

3. ภายใน client และ server ให้ใช้ NPM หรือ PNPM ติดตั้ง Package ที่จำเป็นให้เรียบร้อยโดยใช้ npm i หรือ pnpm i

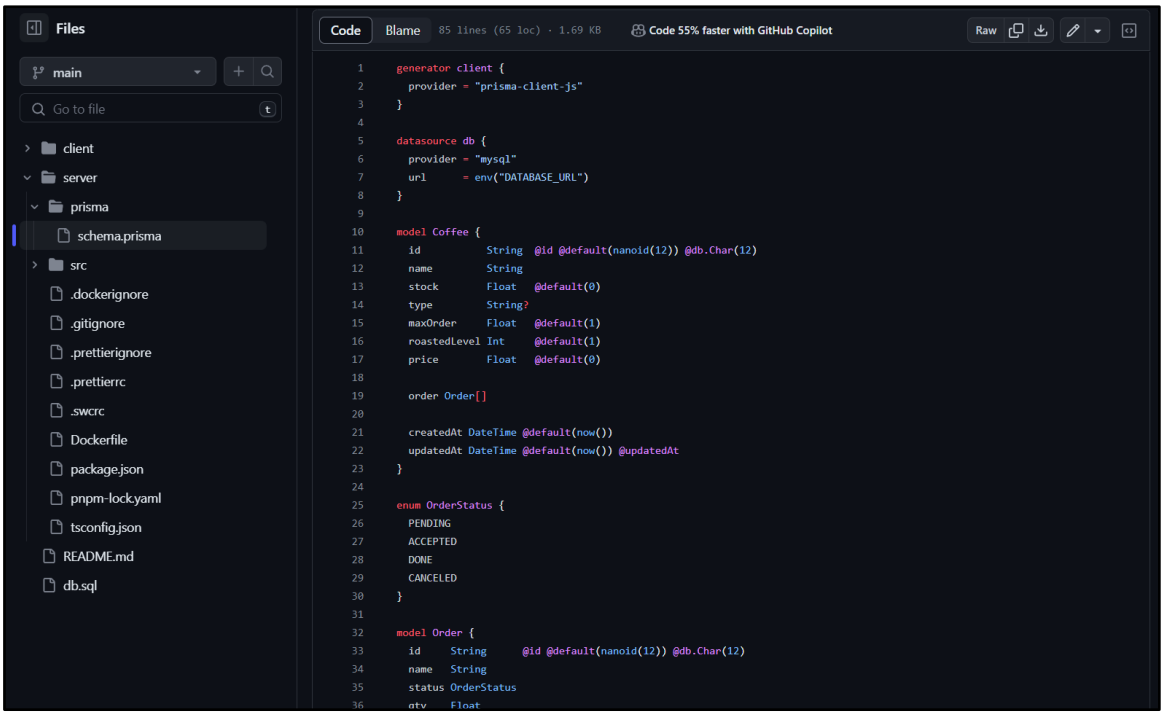



*รูปที่ 9 ติดตั้ง Package (PNPM)*

4. ทำการสร้างฐานข้อมูลเพื่อเตรียมโครงสร้างและนำข้อมูลเข้าระบบ ในตัวอย่างใช้ CLI ในการจัดการฐานข้อมูล สามารถเลือกใช้โปรแกรมใดก็ได้ เช่น Beekeeper Studio Community Edition



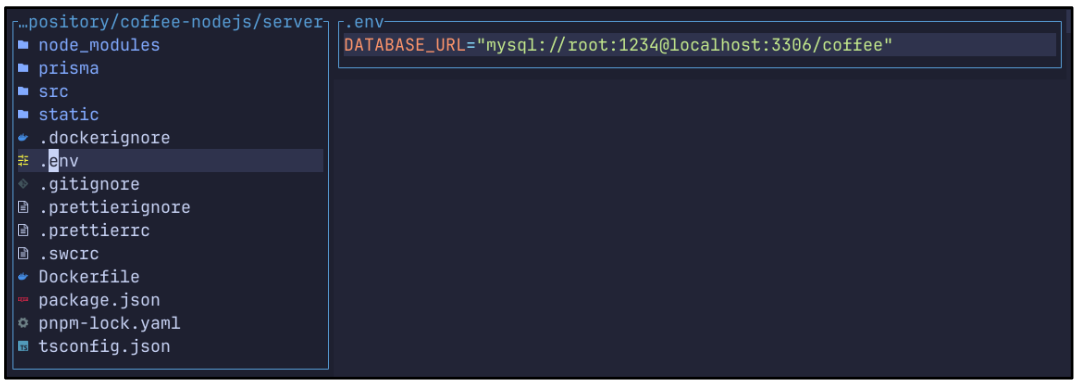

*รูปที่ 10 สร้างฐานข้อมูล*




*รูปที่ 11 โครงสร้างฐานข้อมูล (Database Design)*

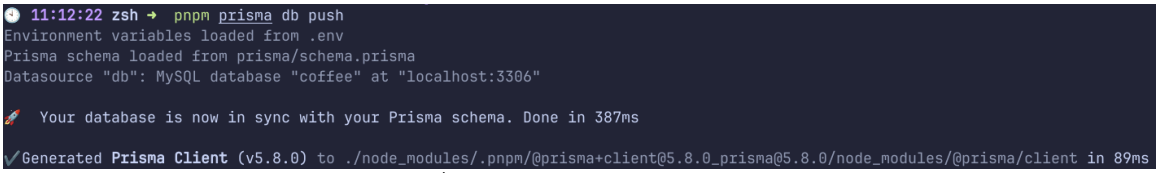
5. เตรียมโครงสร้างให้ตรงกับ Database Design ที่ทำไว้ใน server/prisma/schema.prisma ดังรูปที่ 5 ซึ่งมีขั้นตอนดังนี้

5.1 สร้างไฟล์ server/.env โดยมีเนื้อหาดังรูปที่ 6 โดยเปลี่ยนชื่อผู้ใช้ (root) รหัสผ่าน (1234) และฐานข้อมูล (coffee) ให้ถูกต้อง



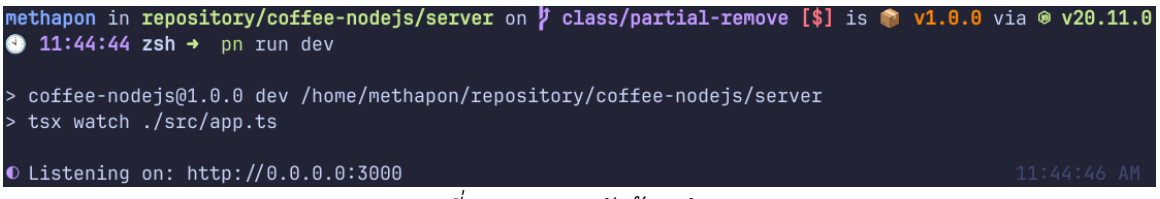

*รูปที่ 12 ค่าการเชื่อมต่อฐานข้อมูล*

5.2 ใช้คำสั่ง pnpm run db:push หรือ pnpm prisma db push หรือ npx prisma db push




*รูปที่ 13 สร้างโครงสร้างฐานข้อมูล*

6. ทดลองระบบหลังบ้าน ในการทำงานของระบบนั้นจะใช้ Package ที่ชื่อว่า tsx ในการสั่งให้ TypeScript ทำงาน แต่เมื่อต้องการนำไปใช้ใน Production ปกติจะต้องทำการ Build เป็น JavaScript ก่อน จะไม่ใช้ TypeScript แบบตรงๆ




*รูปที่ 14 ระบบหลังบ้านทำงาน*

7. การเขียนระบบ API หรือ Rest API ในระบบนี้ตัวอย่างใช้ oak จาก deno (เป็นเพียง 1 ใน Package ตัวเลือกที่สามารถนำมาใช้ได้) ทั้งนี้หากต้องการสร้างขึ้นมาเองก็สามารถเลือกได้จากตัวเลือกต่างๆ เช่น express, koa, fastify, elysia(bun), oak(deno), hono(deno) และอื่นๆ ซึ่งส่วนใหญ่การเลือกใช้งานจะขึ้นอยู่ Performance และ Plugins ที่เพิ่มความสามารถให้กับ Framework นั้นๆ

7.1 การเขียน Endpoint สำหรับดึงข้อมูลจะมีหลักๆ อยู่ 5 Method คือ GET, POST, PUT, PATCH และ DELETE โดยการใช้งานจะแตกต่างกันดังนี้

7.1.1 GET Method จะใช้สำหรับดึงข้อมูล

7.1.2 POST Method สำหรับให้เพิ่มข้อมูลใน Rest API หรืออาจใช้เป็น Action ใน API

7.1.3 PUT Method สำหรับแทนที่หรือแก้ไขข้อมูล

7.1.4 PATCH Method สำหรับแก้ไขบางส่วนของข้อมูล

7.1.5 DELETE Method สำหรับลบ

เมื่อทำการเขียน Endpoint แล้วสามารถทดสอบได้โดยการใช้ Postman หรือ HTTPie ในการเรียกใช้ API ตาม Method และเส้นทางที่ตั้งค่าไว้ เช่น ในรูปที่ 9 สามารถเข้าได้โดยใช้ Method GET ในโปรแกรม http://localhost:3000/coffee/<รหัสของข้อมูล>

สามารถดูเส้นทางทั้งหมดได้ภายใน **server/src/routes**

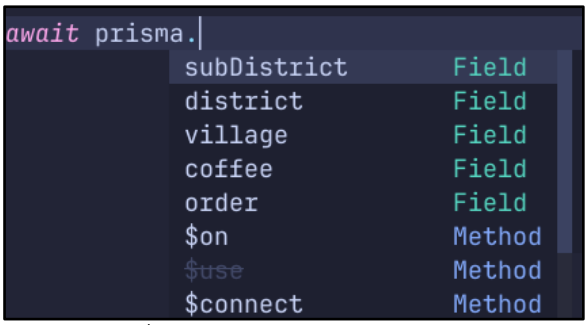
```ts
router.get("/coffee/:id", async (ctx) => {
  // do something
});
```

*รูปที่ 15 Endpoint ของระบบ*

7.2 การติดต่อกับฐานข้อมูลด้วย PrismaORM ซึ่งเมื่อเราได้ดำเนินการในขั้นตอนที่ 5 ไปแล้ว เราจะสามารถเขียนติดต่อฐานข้อมูลได้โดยใช้ Prisma Client

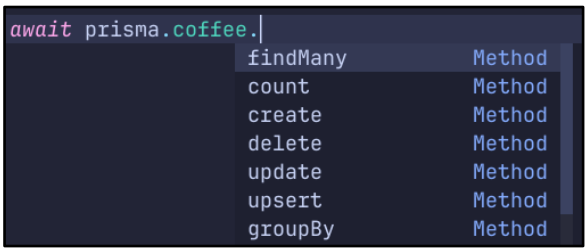
```ts
const prisma = new PrismaClient({
  errorFormat: "minimal",
});
```

*รูปที่ 16 Prisma Client สำหรับติดต่อฐานข้อมูล*

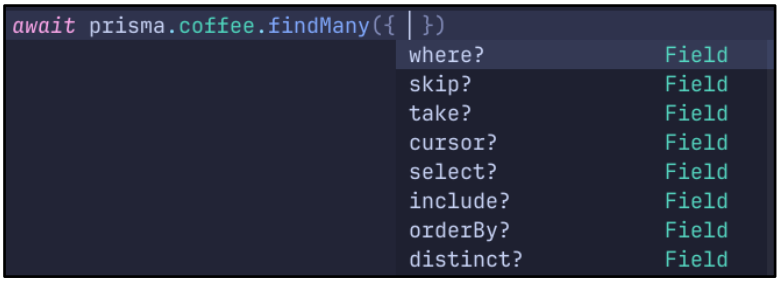



*รูปที่ 17 ตัวอย่างการเขียน Prisma*

เมื่อเขียนโค้ดใน IDE ที่มี Auto-Completion ให้จะแสดง entity ต่างๆ ที่สามารถเข้าถึงได้มาให้ ซึ่งข้อมูลสำหรับ Auto-Completion นี้ถูกดึงมาจาก Type ที่ได้ในขั้นตอนที่ 6

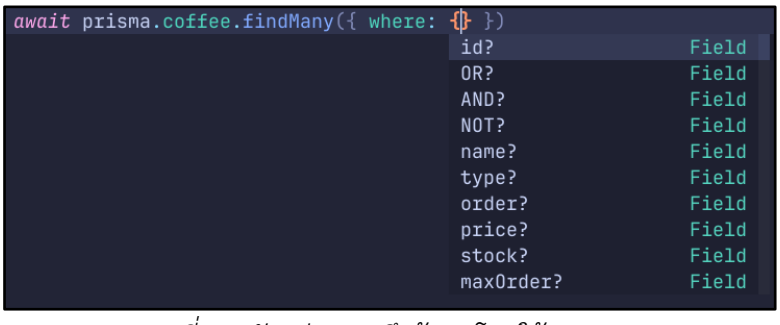



*รูปที่ 18 ตัวอย่างคำสั่งต่างๆ ที่สามารถทำได้ใน PrismaORM*

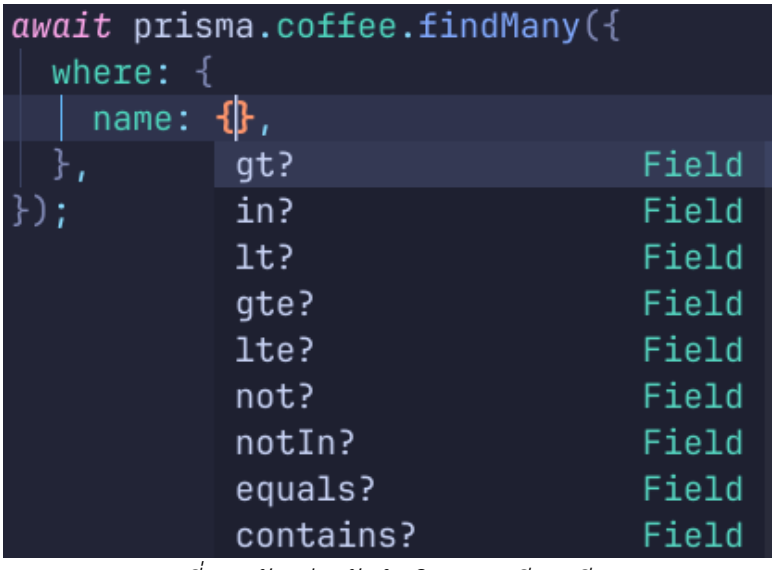



*รูปที่ 19 ตัวอย่างการดึงข้อมูล*

การดำเนินการต่างๆ สามารถทำได้โดยการส่ง Parameter สำหรับควบคุมการกระทำนั้นๆ ตามค่าดังรูปที่ 12

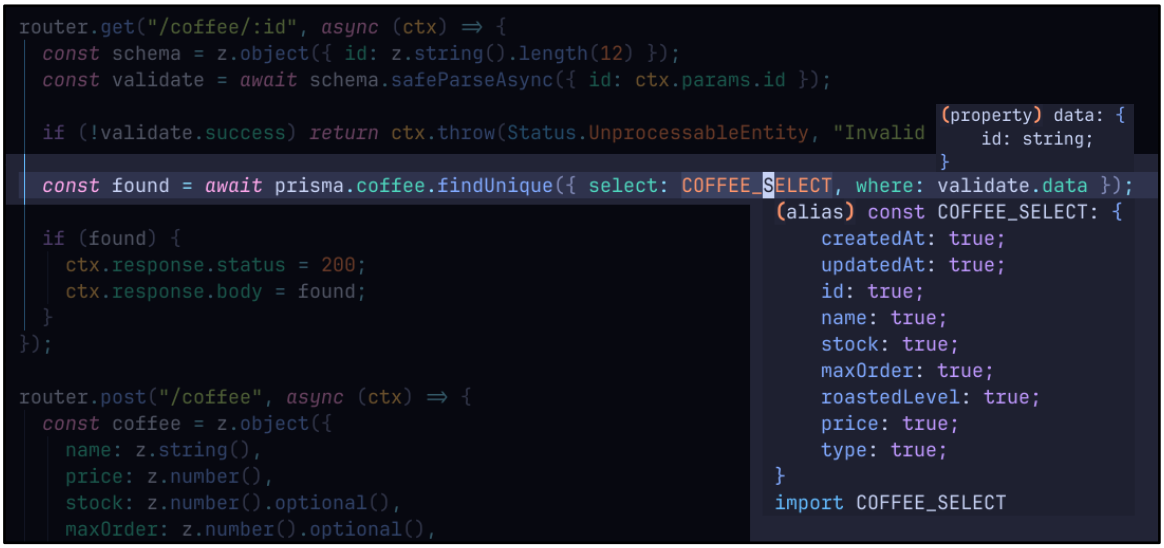



*รูปที่ 20 ตัวอย่างการดึงข้อมูลโดยใช้ Where*




*รูปที่ 21 ตัวอย่างตัวดำเนินการเปรียบเทียบ*

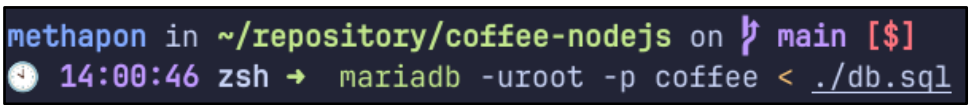
การใช้งานเพิ่มเติมสามารถอ่านเพิ่มเติมได้ที่ https://www.prisma.io/docs/orm/prisma-client/queries ทั้งนี้ การเขียนโปรแกรมเราสามารถพึ่งพาระบบ Type และ Auto-Completion ได้ซึ่งช่วยให้ไม่จำเป็นต้องค้นหาการใช้งานบนเว็บ เนื่องจากบาง Function ใน Package ต่างๆ จะมีการอธิบายไว้ว่าใช้ยังไง ทั้งนี้ ขึ้นอยู่กับ Package นั้นๆ ด้วยว่าได้อธิบายไว้หรือไม่



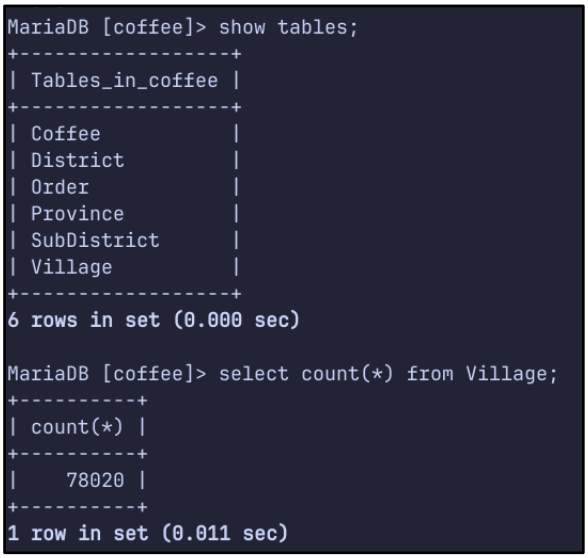

*รูปที่ 22 ตัวอย่างการดึงข้อมูลโดยเลือก Field และ เงื่อนไขที่ id ตรงกัน*


*รูปที่ 23 ตัวอย่างการเพิ่มข้อมูล*

8. เมื่อทุกอย่างพร้อมแล้วให้ทำการนำเข้าข้อมูลที่มีอยู่ก่อนแล้ว เข้าสู่ฐานข้อมูล (coffee) โดยสามารถใช้เครื่องมือที่ถนัดเช่น PhpMyAdmin ที่มากับ XAMPP, MySQL Workbench หรือ Beekeeper Studio ก็ได้ ในตัวอย่างจะใช้ CLI โดยใช้คำสั่งดังนี้




*รูปที่ 24 การนำเข้าข้อมูล*




*รูปที่ 25 ข้อมูลหลังการนำเข้า*

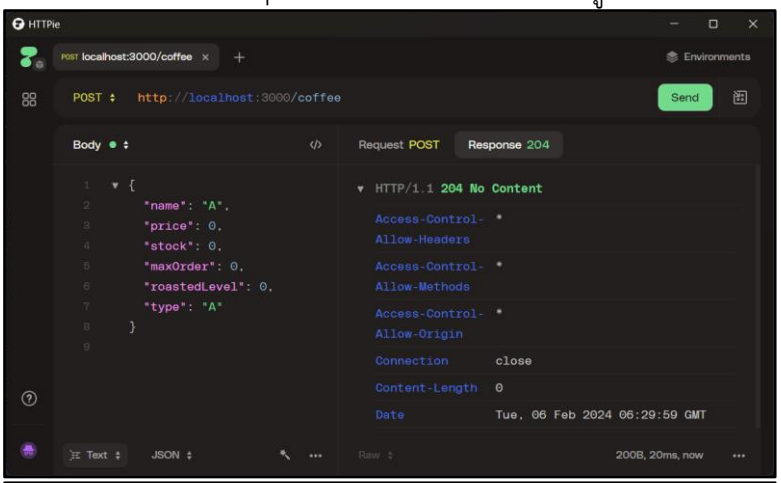
ข้อที่ 1) ให้ทำการเขียนโค้ดในการเพิ่มข้อมูลชนิดกาแฟ ใน **server/src/routes/coffee.ts** บรรทัดที่ 54 ถึง 59 ใช้ **PrismaORM** โดยใช้ **Method** ที่ชื่อว่า **create** และ **createMany**

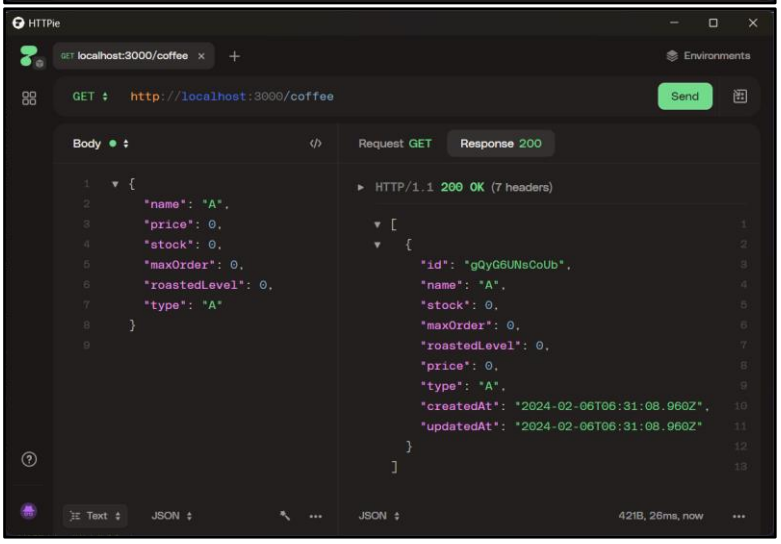
อ่านเพิ่มเติมได้ที่

- https://www.prisma.io/docs/orm/prisma-client/queries/crud#create
- https://www.prisma.io/docs/orm/prisma-client/queries/crud#create-multiple-records

```ts
router.post("/coffee", async (ctx) => {
  const coffee = z.object({
    name: z.string(),
    price: z.number(),
    stock: z.number().optional(),
    maxOrder: z.number().optional(),
    roastedLevel: z.number().optional(),
    type: z.string(),
  });
  const schema = z.union([coffee, coffee.array()]);

  const payload = await ctx.request.body().value;
  const validate = await schema.safeParseAsync(payload);

  if (!validate.success) return ctx.throw(Status.UnprocessableEntity, "Invalid Body");

  // check if input is array or not
  if (true || false) {
    // Add data to database
  } else {
    // Add multiple data to database
  }

  consola.success("Insert Success");
  ctx.response.status = 204;
});
```

การทดสอบ Endpoint สามารถทดสอบได้ตามรูปต่อไปนี้



ข้อที่ 2) ให้ทำการเขียนโค้ดในการเพิ่มข้อมูล ใน **server/src/routes/order.ts** บรรทัดที่ 52 ถึง 56 ใช้ **PrismaORM** โดยใช้ **Method** ที่ชื่อว่า **update** และ **create** ภายใน **transaction** โดยมีเงื่อนไขว่าเมื่อทำการสั่งซื้อ (order) แล้วหากจำนวน stock ไม่เพียงพอต่อจำนวนที่สั่งซื้อ คำสั่งซื้อนั้นจะถูกปฏิเสธและการดำเนินการใดๆ ที่เกิดขึ้น เช่นการ update จำนวนสินค้าในสต็อก จะต้องกลับไปเป็นเหมือนเดิมก่อนการสั่งซื้อ และแจ้ง Error ไปโดยมี Status Code เป็น 406 (NotAcceptable) พร้อมข้อความ “Not Enough Quantity”

ตัวอย่างอันกอริธึมเช่น

Stock: 5

Order: 10

1. Order < Stock
2. IF TRUE: Stock = Stock – Order
3. ELSE: Throw Error 406

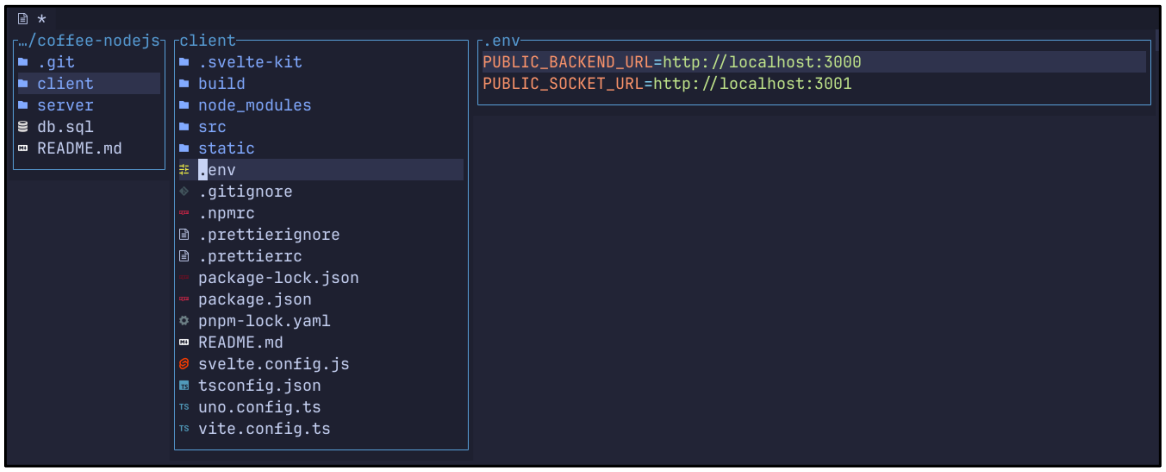
(ตัวอย่างการใช้งาน Transaction อยู่ในบรรทัดที่ 82 ถึง 104 ในไฟล์เดียวกันเมื่อยังไม่มีการแก้ไข ซึ่งในตัวอย่างดังกล่าวเป็นการตรวจสอบสถานะปัจจุบัน)

```
11 router.post("/order", async (ctx) => {
10   const schema = z.object({
9      name: z.string(),
8      qty: z.number().gt(0),
7      coffeeId: z.string().length(12),
6      villageId: z.string().length(8),
5    });
4    const validate = await schema.safeParseAsync(await ctx.request.body().value);
3
2    if (!validate.success) return ctx.throw(Status.UnprocessableEntity, "Invalid Body");
1
52   // prevent racing condition - do transaction  You, 2024-01-19 - Initial commit
1    await prisma.$transaction(async (tx) => {     ■ 'tx' is declared but its value is never read.
2      // implement transaction
3    });
4
5    ws.io.emit("OrderReceived");
6
7    consola.success("Order Success");
8    ctx.response.status = 204;
9  });
```

การใช้ Transaction จะช่วยให้ฐานข้อมูลทำตามคำสั่งย่อยๆ โดยที่เมื่อมีข้อผิดพลาดหรือเงื่อนไขบางอย่างที่ทำให้ Transaction นั้นไม่สำเร็จ จะทำให้ย้อนกลับไปเป็นเหมือนก่อนเกิด Transaction อีกทั้ง Transaction จะเป็นการกระทำคำสั่งทั้งหมด เหมือนเป็นคำสั่งเดียวและไม่ถูกแทรกแซงซึ่งเป็นการป้องกันไม่ให้กระทำกับข้อมูลดังกล่าวในเวลาเดียวกัน (Lock) ซึ่งจะป้องกัน Racing Condition ได้

การทดสอบสามารถทำได้เหมือนข้อแรก ให้ลองทำการ Order และลองใส่จำนวนเกินกว่าที่มีใน Stock และเรียกใช้ Endpoint โดยมีข้อมูลคือ name, qty, coffeeId (ใช้ตัวอย่างจากข้อ 1 ที่ทำเองและได้รับ id มา) และ villageId (ตัวอย่างเช่น 50010701)

9. ให้ทำการสร้างไฟล์ **client/.env** โดยมีเนื้อหาดังนี้




*รูปที่ 26* ไฟล์ .env ในส่วนของ client

10. โครงสร้างไฟล์ของ SvelteKit จะมีการสร้างหน้าใหม่ไว้ใน **routes/+page.svelte** เมื่อต้องการสร้างหน้าใหม่ให้สร้างแฟ้มซึ่งจะเป็น URL หรือ path ที่จะเข้าถึงจากหน้าเว็บ และ **+page.svelte** ที่จะเป็นเนื้อหาภายในหน้านั้นๆ


*รูปที่ 27 ไฟล์ .env ในส่วนของ client*


*รูปที่ 28 ตัวอย่าง State ใน Svelte 5*

จาก รูปที่ 28 เป็นตัวแปรที่เก็บ State ซึ่งเมื่อตัวแปรมีการเปลี่ยนแปลง และมีการนำตัวแปรดังกล่าวไปแสดงผลภายใน HTML จะทำให้ HTML Element ที่มีความเกี่ยวข้องกับตัวแปรดังกล่าว มีการเปลี่ยนแปลงตาม ขึ้นอยู่กับว่าใช้ทำอะไร เช่นใช้ควบคุมแสดงผลด้วย if else หรือใช้ แสดงข้อความ ฯลฯ

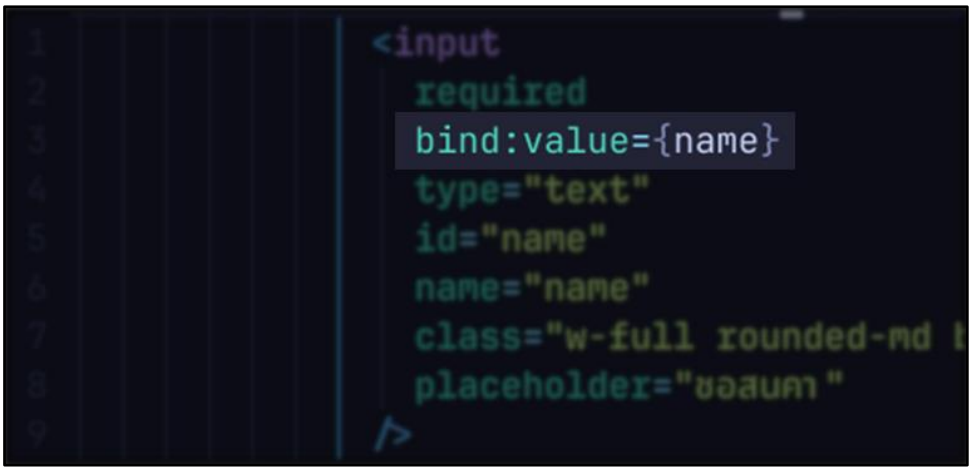
ณ เวลาที่เขียน ใช้ Svelte 5 ซึ่งเป็น Version ใหม่ทียังไม่พร้อมใช้งานใน Production มีความต่างกับ Version 4 คือต้องประการ $state ในขณะที่ Version 4 ไม่จำเป็น สามารถอ่านเพิ่มเติมได้ที่ https://svelte-5-preview.vercel.app/docs/introduction

ใน Framework อย่างเช่น React, Vue และ Svelte จะมีสิ่งที่เรียกว่า State ซึ่งเป็นเหมือนกับสถานะที่จะสามารถใช้ในการควบคุมการแสดงผลบนหน้าเว็บเมื่อมีการเปลี่ยนแปลงได้


*รูปที่ 29 ตัวอย่างการแสดงผล ตัวแปร ควบคุมด้วย if else และ loop*

จาก รูปที่ 29 จะเป็นการควบคุมการแสดงผลด้วย if else และ each กับตัวแปร coffee ซึ่งเป็น State การแสดงผลจะเปลี่ยนแปลงตามตัวแปรที่เกี่ยวข้องซึ่งก็คือ coffee

ภายใน each ที่เป็น loop จะมีการแสดงผลเปลี่ยนแปลงตามค่าที่อยู่ในตัวแปร item ที่อยู่ใน coffee อีกทีนึง




*รูปที่ 30 การนำ State ไปใช้งานกับ Input*

ในส่วนของ Input เนื่องจากการทำงานโดยใช้ JavaScript Framework ส่วนใหญ่มักจะมีการนำข้อมูลที่กรอกเก็บไว้ในตัวแปร ซึ่งหากเก็บไว้ในตัวแปรปกติ จะทำให้ ไม่ว่าพิมพ์อะไรเข้าไป ก็จะทำให้ Input กลับมาเหมือนเดิมหรือค่าที่กรอกไม่ถูกเก็บไว้ในตัวแปร เนื่องจาก ตัวแปรนั้นไม่ใช่ State ทำให้ไม่เกิด Reactive

ข้อที่ 3) ให้สร้าง **State** ใน **client/src/routes/manage/coffee-modal.svelte** บรรทัดที่ 26 ถึง 30 สำหรับเก็บข้อมูล Input สำหรับ **สร้าง** และ **แก้ไข** ข้อมูล โดยเมื่อเป็นการแก้ไขข้อมูล จะต้องมีค่าเริ่มต้นให้โดยรับค่ามาจากข้อมูลที่ต้องการแก้ไข (ข้อมูลที่ส่งเข้ามาสามารถเข้าถึงได้จากตัวแปร props (ตัวอย่าง บรรทัดที่ 25)

```
25  let name = $state((props.action === 'edit' && props.name) || '');
 1  let stock = ; // number      ■ ts: Expression expected.
 2  let maxOrder = ; // number      ■ ts: Expression expected.
 3  let roastedLevel = ; // number      ■ ts: Expression expected.
 4  let price = ; // number      ■ ts: Expression expected.
 5  let type = ; // string      ■ ts: Expression expected.
 6  let open = $state<boolean>(false);
```

เมื่อเข้าหน้า **/manage** แล้ว กดสร้างใหม่ เมื่อกรอกข้อมูลในแต่ละช่อง และดู Console ของ Web Browser เมื่อกดส่ง ข้อมูลที่ขึ้นใน Console ควรจะมีค่าเหมือนกับที่กรอกทุกประการ

ข้อที่ 4) ให้เชื่อมต่อหน้าบ้านกับหลังบ้านใน **client/src/routes/manage/coffee-modal.svelte** บรรทัดที่ 47 ถึง 62 และ **client/src/routes/manage/delete-coffee-modal.svelte** บรรทัดที่ 13 ให้เชื่อมต่อกับหลังบ้านโดยใช้ Fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) เพื่อใช้ในการจัดการข้อมูลเมล็ดกาแฟ ตัวอย่างสามารถดูได้ใน **client/src/routes/checkout/+page.svelte** บรรทัดที่ 72 และสามารถตรวจสอบเส้นทาง (Endpoint) ที่อยู่ในระบบหลังบ้านได้จาก รูปที่ 15

```
11  async function submit(e: SubmitEvent) {
10    e.preventDefault();
 9    console.log(
 8      `name: ${name}\n`,
 7      `stock: ${stock}\n`,
 6      `maxOrder: ${maxOrder}\n`,
 5      `roastedLevel: ${roastedLevel}\n`,
 4      `price: ${price}\n`,
 3      `type: ${type}\n`,
 2      `open: ${open}\n`,
 1    );
46    You, 2024-02-05 - Fix querystring on submit
 1    switch (props.action) {
 2      case 'add': {
 3        // send request to backend to add data
 4        console.log('Coffee added successfully!');
 5        props.onAction();
 6        open = false;
 7        break;
 8      }
 9      case 'edit': {
10        // send request to backend to update data
11        console.log('Coffee editted successfully!');
12        props.onAction();
13        open = false;
14        break;
15      }
16    }
17  }
```

```
11
  async function submit() {
    // send request to backend to delete data using id
    console.log('Delete Success');
    open = false;
    onDelete();
  }
</script>
```

```
async function submit(e: SubmitEvent) {
  e.preventDefault();

  const res = await fetch(`${env.PUBLIC_BACKEND_URL}/order`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      name: cart.order.name,
      qty: cart.order.qty,
      coffeeId: cart.order.coffee?.id,
      villageId: villageSelected,
    }),
  }).catch((e) => console.error('Error: ', e));

  if (!res) return;
  if (!res.ok) return console.error('Failed to submit order:', await res.text());

  console.log('Order submitted successfully!');
  await goto('/status');
}
```

(ตัวอย่าง)

**เฉลยใบงาน**

ข้อที่ 1)

```
if (!Array.isArray(validate.data)) {
  await prisma.coffee.create({
    data: validate.data,
  });
} else {
  await prisma.coffee.createMany({
    data: validate.data,
  });
}
```

ข้อที่ 2)

```
// prevent racing condition - do transaction
await prisma.$transaction(async (tx) => {
  const coffee = await tx.coffee.update({
    data: {
      stock: { decrement: validate.data.qty },
    },
    where: { id: validate.data.coffeeId },
  });

  if (coffee.stock < 0) return ctx.throw(Status.NotAcceptable, "Not Enough Quantity");

  await tx.order.create({
    data: { ...validate.data, status: OrderStatus.PENDING },
  });
});
```

ข้อที่ 3)

```
let name = $state((props.action === 'edit' && props.name) || '');
let stock = $state((props.action === 'edit' && props.stock) || 0);
let maxOrder = $state((props.action === 'edit' && props.maxOrder) || 1);
let roastedLevel = $state((props.action === 'edit' && props.roastedLevel) || 0);
let price = $state((props.action === 'edit' && props.price) || 0);
let type = $state((props.action === 'edit' && props.type) || '');
let open = $state<boolean>(false);
```

ข้อที่ 4)

```
case 'add': {
  const res = await fetch(`${env.PUBLIC_BACKEND_URL}/coffee`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      name,
      stock,
      maxOrder,
      roastedLevel,
      price,
      type,
      open,
    }),
  }).catch((e) => console.error('Error: ', e));

  if (!res) return;
  if (!res.ok) return console.error('Failed to add coffee: ', await res.text());

  console.log('Coffee added successfully!');
  props.onAction();
  open = false;
  break;
}
```

```
case 'edit': {
  const res = await fetch(`${env.PUBLIC_BACKEND_URL}/coffee/${props.id}`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      name,
      stock,
      maxOrder,
      roastedLevel,
      price,
      type,
      open,
    }),
  }).catch((e) => console.error('Error: ', e));

  if (!res) return;
  if (!res.ok) return console.error('Failed to edit coffee: ', await res.text());

  console.log('Coffee editted successfully!');
  props.onAction();
  open = false;
  break;
}
```