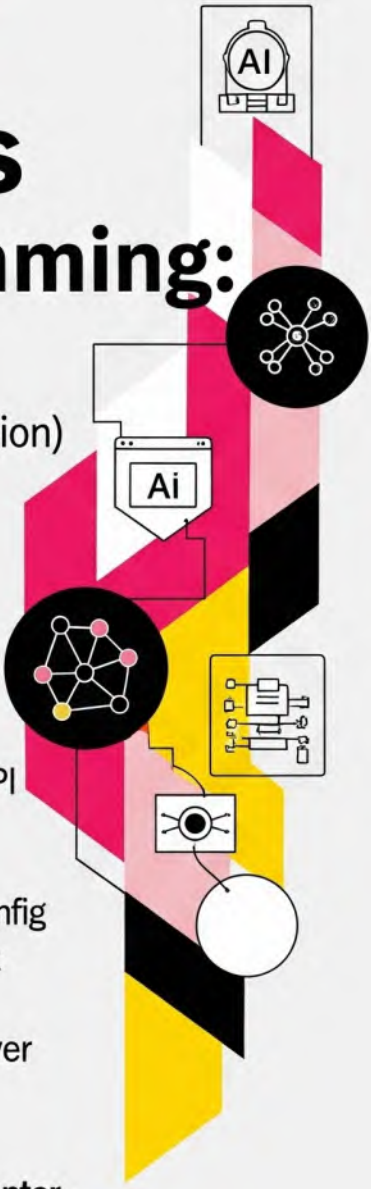
Bun.js Web Programming:

INTERMEDIATE

(Integrative-Generative AI Edition)



CONTENTS:

Student Price Book Center

คำนำ

ในโลกการพัฒนาเว็บแอปพลิเคชันยุคปัจจุบัน ความรวดเร็วในการพัฒนา ประสิทธิภาพของระบบ และความยืดหยุ่นในการปรับตัวต่อเทคโนโลยีใหม่ ๆ ถือเป็นปัจจัยสำคัญที่กำหนดความสำเร็จของนักพัฒนาและองค์กร หนึ่งในเครื่องมือที่เกิดขึ้นมาเพื่อตอบโจทย์ดังกล่าวคือ **Bun.js** — JavaScript runtime รุ่นใหม่ที่ถูกออกแบบมาเพื่อให้ทำงานได้เร็ว ใช้งานง่าย และมีฟังก์ชันครบครันภายในตัวเดียว

หนังสือ **"Bun.js Web Programming: Intermediate"** เล่มนี้ถูกเขียนขึ้นเพื่อต่อยอดความรู้จากระดับพื้นฐานไปสู่ระดับกลาง โดยเน้นการประยุกต์ใช้งานจริง ครอบคลุมหัวข้อสำคัญที่นักพัฒนาจำเป็นต้องรู้เมื่อเริ่มสร้างแอปพลิเคชันจริงด้วย Bun.js ไม่ว่าจะเป็นการสร้าง HTTP Server การจัดการไฟล์และโปรเซส การทดสอบโค้ด การจัดการค่า environment และ config การบริหาร dependencies การทำงานร่วมกับไลบรารีภายนอก ตลอดจนการใช้ Bun Dev Server เพื่อเพิ่มประสิทธิภาพการพัฒนา เนื้อหาในเล่มถูกจัดเรียงตามลำดับการเรียนรู้ที่ต่อเนื่องและสอดคล้องกับ workflow ของนักพัฒนา เริ่มต้นจาก **บทที่ 7 "การสร้าง HTTP Server ด้วย Bun"** ซึ่งจะพาผู้อ่านเรียนรู้การสร้างเว็บเซิร์ฟเวอร์ด้วย Bun APIs ทั้งในมุมมองพื้นฐานและเชิงลึก รวมถึงการใช้งาน WebSocket เพื่อรองรับการสื่อสารแบบเรียลไทม์

ต่อมาใน **บทที่ 8 "การใช้งาน File System และ Process API"** ผู้อ่านจะได้เรียนรู้การอ่านและเขียนไฟล์ด้วย Bun APIs การจัดการ directory และไฟล์แบบ asynchronous การใช้งาน process และ environment variables รวมถึงการสร้าง CLI tools ที่สามารถใช้ได้จริงในงานพัฒนา พร้อมตัวอย่างบูรณาการที่ช่วยให้เข้าใจการประยุกต์ใช้ในโปรเจกต์จริง

บทที่ 9 "การทดสอบด้วย Bun Test Runner" จะเน้นการสร้างและจัดการชุดทดสอบด้วยเครื่องมือในตัวของ Bun เช่นการเขียน unit test การตั้งค่าและรัน test suites การ mock ฟังก์ชันและโมดูล ตลอดจนการวัด coverage เพื่อประเมินคุณภาพของโค้ดอย่างมีระบบ

ใน **บทที่ 10 "การจัดการ Environment Variables และ Config"** ผู้อ่านจะได้เรียนรู้การกำหนดค่าแอปพลิเคชันผ่าน environment variables การตั้งค่า config แยกตามสภาพแวดล้อม (development, testing, production) การจัดการ secrets อย่างปลอดภัย และการ integrate กับ dotenv หรือไฟล์ config เพื่อให้การพัฒนาเป็นไปอย่างยืดหยุ่นและปลอดภัย

หัวข้อ **บทที่ 11 "การจัดการ Dependency และ Package Management"** จะครอบคลุมการติดตั้งและจัดการ dependencies ด้วย Bun การอัปเดตและควบคุมเวอร์ชัน การแก้ปัญหา conflicts รวมถึงการ publish package เพื่อเผยแพร่ผลงานสู่สาธารณะ

จากนั้น **บทที่ 12 "การทำงานร่วมกับ Third-party Libraries"** จะอธิบายวิธีการใช้ไลบรารีจาก npm ecosystem กับ Bun การแก้ปัญหา compatibility การ optimize performance และตัวอย่างการ integrate React หรือ Vue เพื่อสร้างเว็บแอปที่ทันสมัยและมีประสิทธิภาพสูง

ปิดท้ายด้วย **บทที่ 13 "การตั้งค่าและใช้งาน Bun Dev Server"** ที่เน้นการตั้งค่า hot reload, live reload, การ debug แอปพลิเคชัน และการเชื่อมต่อกับ editor/debugger เพื่อเพิ่มความสะดวกและความเร็วในการพัฒนา พร้อมตัวอย่างการใช้งานจริงในสถานการณ์หลากหลาย

หนังสือเล่มนี้ถูกออกแบบให้ผู้อ่านสามารถเรียนรู้ได้ทั้งจากเนื้อหาเชิงทฤษฎีและการปฏิบัติจริง โดยทุกบทจะมีตัวอย่างโค้ดและแนวทางประยุกต์ใช้ที่สามารถนำไปปรับใช้ได้ทันที เป้าหมายคือให้ผู้อ่านสามารถใช้ Bun.js ในการพัฒนาเว็บแอปพลิเคชันระดับ production ได้อย่างมั่นใจ มีโครงสร้างการทำงานที่มีประสิทธิภาพ ปลอดภัย และพร้อมต่อการขยายในอนาคต

หวังว่าหนังสือเล่มนี้จะเป็นคู่มือสำคัญที่ช่วยยกระดับทักษะการพัฒนาเว็บของคุณให้ก้าวหน้าไปอีกขั้น พร้อมรับมือกับความท้าทายในโลกการพัฒนาเว็บยุคใหม่ที่เปลี่ยนแปลงอย่างรวดเร็วและต่อเนื่อง.

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 7

การสร้าง HTTP Server ด้วย Bun

(HTTP Server by Bun)

เนื้อหา

- การสร้าง HTTP Server ด้วย Bun
- การสร้าง HTTP Server ด้วย Bun (เชิงลึก)
- การสร้างเว็บเซิร์ฟเวอร์ด้วย Bun APIs
- การใช้งาน WebSocket กับ Bun

บทที่ 7: การสร้าง HTTP Server ด้วย Bun เป็นบทที่มุ่งเน้นให้ผู้อ่านได้เรียนรู้การพัฒนาเว็บเซิร์ฟเวอร์ที่ทันสมัยและมีประสิทธิภาพโดยใช้ Bun ซึ่งเป็นแพลตฟอร์ม JavaScript runtime ที่รวดเร็วและทันสมัย โดยบทนี้จะเริ่มตั้งแต่พื้นฐานการสร้างเว็บเซิร์ฟเวอร์ด้วย Bun APIs ไปจนถึงการจัดการเส้นทาง (route) และการตอบสนองต่อคำร้องขอ (request/response) อย่างมีประสิทธิภาพ

เนื้อหาหลักของบทนี้จะเริ่มจากการแนะนำวิธีการสร้าง HTTP Server ด้วยฟังก์ชันและเครื่องมือที่ Bun มีให้ เพื่อให้ผู้อ่านเข้าใจโครงสร้างและวิธีการทำงานของเซิร์ฟเวอร์อย่างละเอียด ซึ่งช่วยให้สามารถปรับแต่งและพัฒนาระบบตามความต้องการได้อย่างอิสระ

นอกจากการตั้งค่าเซิร์ฟเวอร์พื้นฐาน บทนี้ยังเน้นการจัดการเส้นทางหรือ routing ซึ่งเป็นหัวใจสำคัญของเว็บเซิร์ฟเวอร์สมัยใหม่ การกำหนดและจัดการ route อย่างถูกต้องจะช่วยให้เซิร์ฟเวอร์ตอบสนองต่อคำร้องขอที่หลากหลายได้อย่างรวดเร็วและแม่นยำ ทั้งนี้จะมีตัวอย่างการใช้งานจริงเพื่อให้ผู้อ่านเห็นภาพรวมและวิธีการนำไปใช้ในโปรเจกต์จริง

ในส่วนของการจัดการ request และ response จะเจาะลึกถึงการประมวลผลข้อมูลที่ได้รับจากผู้ใช้ รวมถึงการตอบกลับข้อมูลอย่างเหมาะสม โดยจะอธิบายเทคนิคและฟังก์ชันที่ช่วยให้งานนี้เป็นไปอย่างราบรื่นและมีประสิทธิภาพ ซึ่งเป็นพื้นฐานที่สำคัญสำหรับการพัฒนาเว็บแอปพลิเคชันทุกประเภท

บทนี้ยังนำเสนอการใช้งาน WebSocket ร่วมกับ Bun ซึ่งเป็นเทคโนโลยีสำคัญสำหรับการสื่อสารแบบเรียลไทม์ในแอปพลิเคชันยุคใหม่ การเข้าใจและสามารถสร้างระบบที่รองรับการสื่อสารแบบสองทางได้อย่างมีประสิทธิภาพ จะช่วยเปิดโอกาสให้ระบบมีความตอบสนองที่รวดเร็วและประสบการณ์ผู้ใช้ที่ดีขึ้น

เพื่อให้ผู้อ่านได้ลงมือปฏิบัติจริง บทนี้ได้จัดเตรียมตัวอย่าง REST API เบื้องต้นที่ใช้ Bun เป็นตัวขับเคลื่อน ซึ่งจะช่วยให้เข้าใจการออกแบบและพัฒนา API ที่ง่ายต่อการบำรุงรักษาและขยายในอนาคต นอกจากนี้ยังแสดงถึงการเชื่อมต่อระหว่าง client และ server อย่างชัดเจน

บทเรียนในบทนี้จะช่วยให้ผู้อ่านมีความรู้และทักษะในการสร้าง HTTP Server ที่มีประสิทธิภาพ รองรับการทำงานในโลกจริงได้ ไม่ว่าจะเป็นการจัดการเส้นทางที่ซับซ้อน หรือการสร้างระบบที่ต้องการความสามารถพิเศษอย่าง WebSocket ด้วยความรวดเร็วและประสิทธิภาพของ Bun การพัฒนาเว็บเซิร์ฟเวอร์จึงเป็นไปอย่างง่ายดายและยืดหยุ่น บทนี้จึงเป็นกุญแจสำคัญที่ช่วยเปิดประตูให้กับนักพัฒนาที่ต้องการสร้างระบบที่มีความทันสมัยและพร้อมใช้งานในยุคดิจิทัล สุดท้ายนี้ บทที่ 7 จะช่วยส่งเสริมให้ผู้อ่านได้เห็นภาพรวมของการทำงานของ HTTP Server ในบริบทของ Bun และเป็นแนวทางที่ดีในการต่อยอดความรู้เพื่อสร้างแอปพลิเคชันที่ตอบโจทย์การใช้งานจริงในอนาคตอย่างมั่นใจและมีประสิทธิภาพสูงสุด

การสร้าง HTTP Server ด้วย Bun

- การสร้างเว็บเซิร์ฟเวอร์ด้วย Bun APIs
- การจัดการ route และ request/response
- การใช้งาน WebSocket กับ Bun
- ตัวอย่าง REST API เบื้องต้น

---

1. การสร้างเว็บเซิร์ฟเวอร์ด้วย Bun APIs

Bun มาพร้อม API ในตัวที่รองรับการสร้าง HTTP Server แบบง่ายและรวดเร็ว โดยใช้ฟังก์ชัน serve() จากโมดูล bun ซึ่งคล้ายกับฟังก์ชันใน Node.js เช่น http.createServer แต่ Bun ออกแบบมาให้เร็วและใช้งานง่าย

```
import { serve } from "bun";

const server = serve({
  port: 3000,
  fetch(request) {
    return new Response("Hello from Bun HTTP Server!", {
      status: 200,
      headers: { "Content-Type": "text/plain" }
    });
  }
});

console.log("Server running on http://localhost:3000");
```

- ฟังก์ชัน serve() รับอ็อบเจกต์ที่กำหนดพอร์ตและ callback fetch() เพื่อจัดการ HTTP request
- ใน fetch(request) สามารถอ่านข้อมูลใน request และส่ง Response กลับ

2. การจัดการ route และ request/response

เพื่อสร้างเว็บเซิร์ฟเวอร์ที่ซับซ้อนขึ้น เราสามารถจัดการ routing ได้ง่าย ๆ โดยเช็ค path หรือ method ของ request

```
import { serve } from "bun";

const server = serve({
  port: 3000,
  fetch(request) {
    const url = new URL(request.url);

    if (url.pathname === "/") {
      return new Response("Welcome to Home!", { status: 200 });
    } else if (url.pathname === "/about") {
      return new Response("About us page", { status: 200 });
    } else if (url.pathname === "/api/data" && request.method === "GET") {
      const data = { message: "Hello API" };
      return new Response(JSON.stringify(data), {
        status: 200,
        headers: { "Content-Type": "application/json" }
      });
    } else {
      return new Response("Not Found", { status: 404 });
    }
  }
});

console.log("Server running on http://localhost:3000");
```

- เราใช้ new URL(request.url) เพื่อแยก path, query, และอื่น ๆ
- ส่ง Response โดยสามารถกำหนด headers และ status code ได้ตาม HTTP standard

3. การใช้งาน WebSocket กับ Bun

Bun รองรับ WebSocket API ในตัว โดยเราสามารถสร้าง WebSocket server ใน HTTP server เดียวกันได้ เช่น

```
import { serve } from "bun";

const server = serve({
  port: 4000,
  websocket: {
    open(ws) {
      console.log("WebSocket connection opened");
      ws.send("Welcome to WebSocket server!");
    },
    message(ws, message) {
      console.log("Received:", message);
      ws.send(`Echo: ${message}`);
    },
    close(ws, code, reason) {
      console.log(`WebSocket closed: ${code} - ${reason}`);
    },
  },
  fetch(request) {
    return new Response("Hello from HTTP server with WebSocket!");
  }
});

console.log("Server and WebSocket running on http://localhost:4000");
```

- ตัวเลือก websocket ภายใน serve() กำหนด callbacks สำหรับการเปิด connection, รับข้อความ, และปิด connection
- ใน callback message เราสามารถตอบกลับข้อความ client ได้

---

4. ตัวอย่าง REST API เบื้องต้น

เราสามารถสร้าง REST API ง่าย ๆ ด้วย Bun โดยรับและส่งข้อมูล JSON ผ่าน HTTP methods ต่าง ๆ

```
import { serve } from "bun";

interface User {
  id: number;
  name: string;
```

```
}

const users: User[] = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];

const server = serve({
  port: 5000,
  async fetch(request) {
    const url = new URL(request.url);
    const method = request.method;

    if (url.pathname === "/users" && method === "GET") {
      return new Response(JSON.stringify(users), {
        status: 200,
        headers: { "Content-Type": "application/json" }
      });
    }

    if (url.pathname === "/users" && method === "POST") {
      try {
        const body = await request.json();
        if (!body.name) {
          return new Response("Missing 'name' field", { status: 400 });
        }
        const newUser: User = {
          id: users.length + 1,
          name: body.name
        };
        users.push(newUser);
        return new Response(JSON.stringify(newUser), {
          status: 201,
          headers: { "Content-Type": "application/json" }
```

```
        });
      } catch {
        return new Response("Invalid JSON", { status: 400 });
      }
    }

    return new Response("Not Found", { status: 404 });
  }
});

console.log("REST API server running on http://localhost:5000");
```

- ตัวอย่างนี้มี array users เป็นแหล่งข้อมูลจำลอง
- รองรับ GET /users เพื่อดึงข้อมูล users ทั้งหมด
- รองรับ POST /users เพื่อเพิ่ม user ใหม่ โดยต้องส่ง JSON body { "name": "..." }
- ใช้ request.json() อ่าน JSON body จาก request

สรุป

หัวข้อ	รายละเอียด
สร้าง HTTP Server ด้วย Bun	ใช้ serve() กำหนดพอร์ตและ callback fetch จัดการ request/response
การจัดการ route	ใช้ URL แยก path และ method เพื่อกำหนด logic route ต่าง ๆ
ใช้งาน WebSocket	กำหนด websocket callbacks สำหรับ open, message, close เพื่อสร้าง WebSocket server
ตัวอย่าง REST API เบื้องต้น	สร้าง endpoint GET, POST จัดการ JSON body และ response ด้วย Bun API

การสร้าง HTTP Server ด้วย Bun (เชิงลึก)

---

1. การสร้างเว็บเซิร์ฟเวอร์ด้วย Bun APIs

พื้นฐานของ Bun HTTP Server

Bun ถูกออกแบบมาเพื่อเป็น JavaScript runtime ที่เร็วและครบเครื่อง ตั้งแต่ runtime engine, bundler, package manager และ HTTP server ในตัว ทำให้ไม่ต้องติดตั้งแยก

- serve() คือฟังก์ชันหลักสำหรับสร้าง HTTP Server

- เป็น API ที่อิงมาตรฐาน Fetch API ของเว็บ (Web Fetch API) ทำให้โค้ดอ่านง่ายและมีความสอดคล้องกับการพัฒนาเว็บฝั่ง client

โครงสร้างพื้นฐานของ serve():

```
serve({
  port: number,              // พอร์ตที่เซิร์ฟเวอร์จะฟังคำขอ
  fetch(request) => Response,  // ฟังก์ชันจัดการ HTTP request
  websocket?: { ... },       // ตัวเลือกสำหรับ WebSocket (ถ้ามี)
  error?(error) => Response // ตัวเลือกจัดการ error ที่เกิดขึ้นใน fetch
});
```

การทำงาน

- Bun เปิดพอร์ตตามที่กำหนด และรอฟัง request
- ทุกครั้งที่มี request เข้ามา Bun จะเรียก callback fetch(request)
- ตัว callback รับ request object (แบบ Fetch API) และต้อง return response object กลับไป
- การใช้ Fetch API ทำให้เขียน server ที่ใช้โมเดล async/await ได้อย่างดี

---

2. การจัดการ route และ request/response

การ routing แบบ manual

Bun ไม่ได้มี router ในตัวเหมือน Express.js เราต้องจัดการ routing เองผ่านการตรวจสอบ URL และ HTTP method

- ใช้ new URL(request.url) เพื่อแยก path, query parameters
- ตรวจสอบ request.method เพื่อตัดสินใจว่าจะตอบสนองแบบไหน เช่น GET, POST, PUT, DELETE
- ส่ง response ด้วย new Response() ที่สามารถกำหนด status code, headers, และ body ได้ตามต้องการ

การจัดการ request body

- Bun รองรับการอ่าน body ได้หลายรูปแบบ เช่น request.text(), request.json(), request.formData()
- ต้อง await เพราะเป็น Promise

ตัวอย่างการจัดการ route (เชิงลึก)

```
async function fetchHandler(request: Request): Promise<Response> {
  const url = new URL(request.url);
  const path = url.pathname;
  const method = request.method;
```

```
  if (path === "/api/users" && method === "GET") {
    // ดึงข้อมูล users
    return new Response(JSON.stringify(users), {
      status: 200,
      headers: { "Content-Type": "application/json" }
    });
  } else if (path === "/api/users" && method === "POST") {
    try {
      const data = await request.json();  // อ่าน JSON body
      if (!data.name) {
        return new Response(JSON.stringify({ error: "Name is required" }), { status: 400 });
      }
      // เพิ่ม user ใหม่
      const newUser = { id: users.length + 1, name: data.name };
      users.push(newUser);
      return new Response(JSON.stringify(newUser), { status: 201, headers: { "Content-Type":
"application/json" } });
    } catch (e) {
      return new Response(JSON.stringify({ error: "Invalid JSON" }), { status: 400 });
    }
  }

  return new Response("Not Found", { status: 404 });
}
```

---

3. การใช้งาน WebSocket กับ Bun

ความเข้าใจเบื้องต้นเกี่ยวกับ WebSocket

- WebSocket คือ protocol สำหรับเปิดการเชื่อมต่อแบบ full-duplex ระหว่าง client และ server
- เหมาะกับแอปพลิเคชัน real-time เช่น chat, เกม, dashboard ที่ต้องอัพเดตข้อมูลสด ๆ

Bun และ WebSocket

- Bun รองรับ WebSocket API ในตัวผ่าน option websocket ใน serve()
- ตัว websocket เป็น object ที่ประกอบด้วย callback หลักๆ ดังนี้:
    - open(ws): เมื่อ client เปิด connection
    - message(ws, message): เมื่อ server รับข้อความจาก client

- close(ws, code, reason): เมื่อ connection ถูกปิด

ตัวอย่างเชิงลึก WebSocket

```
const server = serve({
  port: 8080,
  websocket: {
    open(ws) {
      console.log("Client connected");
      ws.send("Welcome!");
    },
    message(ws, message) {
      console.log("Received from client:", message);
      ws.send(`Echo: ${message}`);  // ส่งกลับข้อความเดิม
    },
    close(ws, code, reason) {
      console.log(`Connection closed: ${code} - ${reason}`);
    }
  },
  fetch(request) {
    return new Response("HTTP Response alongside WebSocket");
  }
});
```

ข้อดี

- การผสาน HTTP และ WebSocket ใน server เดียวกัน ทำให้ง่ายต่อการพัฒนา
- Bun จัดการ WebSocket ในระดับ low-level ทำให้มีความเร็วสูง

---

4. ตัวอย่าง REST API เบื้องต้น (เชิงลึก)

แนวทางการออกแบบ API

- ใช้ HTTP methods ให้เหมาะสม: GET (อ่าน), POST (สร้าง), PUT/PATCH (แก้ไข), DELETE (ลบ)
- ใช้ URL ที่ชัดเจนและสื่อความหมาย เช่น /users, /products/{id}
- ตอบกลับ JSON ที่เป็นมาตรฐาน พร้อม status code ที่เหมาะสม เช่น 200, 201, 400, 404, 500

ตัวอย่าง REST API ที่ครบถ้วน

```
interface User {
```

```
  id: number;
  name: string;
}

const users: User[] = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];

async function apiHandler(request: Request): Promise<Response> {
  const url = new URL(request.url);
  const method = request.method;

  if (url.pathname === "/users") {
    if (method === "GET") {
      return new Response(JSON.stringify(users), {
        status: 200,
        headers: { "Content-Type": "application/json" }
      });
    }
    if (method === "POST") {
      try {
        const body = await request.json();
        if (!body.name) {
          return new Response(JSON.stringify({ error: "Missing name" }), { status: 400, headers: {
"Content-Type": "application/json" } });
        }
        const newUser = { id: users.length + 1, name: body.name };
        users.push(newUser);
        return new Response(JSON.stringify(newUser), { status: 201, headers: { "Content-Type":
"application/json" } });
      } catch (error) {
        return new Response(JSON.stringify({ error: "Invalid JSON" }), { status: 400, headers: {
"Content-Type": "application/json" } });
```

```
    }
  }
}

  return new Response(JSON.stringify({ error: "Not Found" }), { status: 404, headers: { "Content-Type": "application/json" } });
}

serve({
  port: 5000,
  fetch: apiHandler
});
```

ข้อดีของการใช้ Bun สำหรับ REST API

- รันได้เร็วและกินทรัพยากรน้อย
- ติดตั้งและรันง่าย ไม่ต้องพึ่งพาไลบรารีเยอะ
- ใช้มาตรฐาน Fetch API ทำให้โค้ดสอดคล้องกับเว็บเบราว์เซอร์และง่ายต่อการเรียนรู้

สรุปเชิงลึก

หัวข้อ	เชิงลึก
Bun HTTP Server	ใช้ serve() แบบ async พร้อม Fetch API, ง่ายแต่ทรงพลัง, เหมาะกับงานที่ต้องการความเร็วสูงและดีไซน์แบบ modern
Routing	ต้องเขียนเองแต่ยืดหยุ่นมาก, รองรับการตรวจสอบ path, method, query และ body แบบ async ได้
WebSocket	รองรับ protocol WebSocket native ในตัว, ทำให้สร้าง real-time app ได้สะดวก, มี callback lifecycle ครบถ้วน
REST API Design	เน้น JSON communication, status code ชัดเจน, มี error handling และ validation ง่าย
ประสิทธิภาพและการใช้งาน	Bun เหมาะสำหรับ backend ที่ต้องการประสิทธิภาพสูง, เรียนรู้เร็ว, ลด dependency, และพร้อมใช้กับ TS/JS ทั้ง frontend/backend

---

การสร้างเว็บเซิร์ฟเวอร์ด้วย Bun APIs

ภาพรวม

Bun.js มี API สำหรับสร้าง HTTP Server ที่เน้นความเร็วและเรียบง่าย โดยฟังก์ชันหลักคือ serve() ซึ่งรับอ็อบเจกต์ config ที่กำหนดพอร์ตและฟังก์ชัน callback เพื่อจัดการ HTTP requests แบบ asynchronous ผ่าน Fetch API ที่เป็นมาตรฐานของเว็บ

ฟังก์ชัน serve() ทำงานอย่างไร

- เปิดพอร์ตที่กำหนด เช่น 3000
- รอรับ HTTP requests
- สำหรับแต่ละ request จะเรียกฟังก์ชัน callback ที่ชื่อ fetch
- callback นี้รับ Request object (ตามมาตรฐาน Fetch API)
- เราต้อง return Response object ที่ Bun จะส่งกลับ client

รูปแบบการใช้งานพื้นฐาน

```
import { serve } from "bun";

const server = serve({
  port: 3000,  // กำหนดพอร์ตที่ต้องการให้เซิร์ฟเวอร์ฟัง
  fetch(request) {
    // ตัวอย่างส่งข้อความตอบกลับง่าย ๆ
    return new Response("Hello Bun HTTP Server!", {
      status: 200,
      headers: { "Content-Type": "text/plain" }
    });
  }
});

console.log("Server is running on http://localhost:3000");
```

อธิบายโค้ดทีละบรรทัด

- import { serve } from "bun";
  นำเข้า serve ฟังก์ชันจาก Bun runtime เพื่อสร้าง HTTP Server
- serve({ ... })
  เรียกใช้ serve() เพื่อสร้าง server พร้อมตั้งค่าต่าง ๆ
- port: 3000
  กำหนดหมายเลขพอร์ตให้ server ฟังคำขอ

- fetch(request) { ... }
  กำหนด callback สำหรับจัดการ HTTP request ทุกครั้งที่เข้ามา
  - request เป็นอ็อบเจกต์แบบ Fetch API ที่มีข้อมูล request ทั้งหมด เช่น URL, headers, method, body
- return new Response("Hello Bun HTTP Server!", {...})
  สร้าง response กลับไปยัง client
  - ข้อความใน body เป็น "Hello Bun HTTP Server!"
  - สถานะ HTTP เป็น 200 (OK)
  - กำหนด header Content-Type ว่าเป็น plain text
- console.log(...)
  แสดงข้อความใน console ว่า server พร้อมทำงาน

จุดเด่นของ Bun HTTP Server

- API ใช้มาตรฐาน Fetch API ที่ developer ฝั่งเว็บคุ้นเคย ทำให้เรียนรู้ได้เร็ว
- รองรับ async/await ง่ายต่อการเขียนโค้ดแบบ asynchronous
- น้ำหนักเบา ไม่ต้องพึ่งพาไลบรารีเพิ่มเติม เช่น Express
- ประสิทธิภาพสูง เพราะ Bun เขียนด้วย Rust และใช้ JavaScriptCore engine ที่เร็วกว่า V8 ในบางงาน

ตัวอย่างรันและทดสอบ

1. สร้างไฟล์ server.js ใส่โค้ดด้านบน
2. เปิด terminal แล้วรันคำสั่ง
3. bun run server.js
4. เปิดเว็บเบราว์เซอร์เข้า URL: http://localhost:3000
   จะเห็นข้อความ **Hello Bun HTTP Server!**

ตัวอย่างโปรแกรม 3 ตัวอย่างพื้นฐานและ 3 ตัวอย่างแนวประยุกต์ที่ใช้ **การสร้างเว็บเซิร์ฟเวอร์ด้วย Bun APIs** พร้อมโครงสร้างไฟล์, คำอธิบายโค้ด และผลการรัน

ตัวอย่างโปรแกรมพื้นฐาน 3 ตัว

ตัวอย่างที่ 1: เว็บเซิร์ฟเวอร์ตอบข้อความง่าย ๆ

โครงสร้างโปรเจกต์

basic-server-1/

└── server.js

server.js

```
import { serve } from "bun";

serve({
  port: 3000,
  fetch(request) {
    return new Response("Hello from Bun basic server!", {
      status: 200,
      headers: { "Content-Type": "text/plain" }
    });
  }
});

console.log("Server running on http://localhost:3000");
```

คำอธิบายโค้ด

- สร้าง HTTP Server ฟังพอร์ต 3000
- ตอบกลับข้อความง่าย ๆ ด้วย plain text

ผลการรัน

รัน bun run server.js แล้วเข้า http://localhost:3000 จะเห็นข้อความ:

Hello from Bun basic server!

ตัวอย่างที่ 2: เว็บเซิร์ฟเวอร์มี routing แบบง่าย

โครงสร้างโปรเจกต์

basic-server-2/

└── server.js

server.js

```
import { serve } from "bun";

serve({
  port: 3001,
  fetch(request) {
    const url = new URL(request.url);
```

```
    if (url.pathname === "/") {
      return new Response("Home Page", { status: 200 });
    } else if (url.pathname === "/about") {
      return new Response("About Page", { status: 200 });
    } else {
      return new Response("Page Not Found", { status: 404 });
    }
  }
});

console.log("Server running on http://localhost:3001");
```

คำอธิบายโค้ด

- ตรวจสอบ path ของ request
- ตอบข้อความแตกต่างกันตาม route /, /about, และอื่น ๆ เป็น 404

ผลการรัน

รัน bun run server.js เข้า

- http://localhost:3001/ แสดง "Home Page"
- http://localhost:3001/about แสดง "About Page"
- URL อื่น ๆ แสดง "Page Not Found" พร้อมสถานะ 404

ตัวอย่างที่ 3: เว็บเซิร์ฟเวอร์ส่ง JSON Response

โครงสร้างโปรเจกต์

```
basic-server-3/
└── server.js
```

server.js

```
import { serve } from "bun";

serve({
  port: 3002,
  fetch(request) {
    const data = {
      message: "Hello JSON",
      time: new Date().toISOString()
    };
```

```
    return new Response(JSON.stringify(data), {
      status: 200,
      headers: { "Content-Type": "application/json" }
    });
  }
});

console.log("Server running on http://localhost:3002");
```

คำอธิบายโค้ด

- สร้างอ็อบเจกต์ JSON และส่งกลับเป็น response
- ตั้ง header Content-Type เป็น application/json

ผลการรัน

รัน bun run server.js เข้า http://localhost:3002 จะได้ JSON เช่น

```
{
  "message": "Hello JSON",
  "time": "2025-08-09T07:50:00.000Z"
}
```

ตัวอย่างโปรแกรมแนวประยุกต์ 3 ตัว

ตัวอย่างที่ 4: REST API เบื้องต้น – จัดการ Users

โครงสร้างโปรเจกต์

```
app-basic-api/
└── server.js
```

server.js

```
import { serve } from "bun";

let users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];

async function handler(request) {
  const url = new URL(request.url);
```

```
  const method = request.method;

  if (url.pathname === "/users") {
    if (method === "GET") {
      return new Response(JSON.stringify(users), {
        status: 200,
        headers: { "Content-Type": "application/json" }
      });
    } else if (method === "POST") {
      try {
        const body = await request.json();
        if (!body.name) {
          return new Response(JSON.stringify({ error: "Name required" }), { status: 400 });
        }
        const newUser = { id: users.length + 1, name: body.name };
        users.push(newUser);
        return new Response(JSON.stringify(newUser), {
          status: 201,
          headers: { "Content-Type": "application/json" }
        });
      } catch {
        return new Response(JSON.stringify({ error: "Invalid JSON" }), { status: 400 });
      }
    }
  }

  return new Response("Not Found", { status: 404 });
}

serve({
  port: 4000,
  fetch: handler
});
```

```
console.log("REST API server running on http://localhost:4000");
```

คำอธิบายโค้ด

- สร้าง API /users
- GET คืน list ของ users
- POST เพิ่ม user ใหม่ โดยต้องส่ง { "name": "..." } ใน JSON body
- ตรวจสอบ error อย่างง่าย เช่น ชื่อไม่ส่งมา, JSON ไม่ถูกต้อง
- ตอบ HTTP status 200, 201, 400, 404 ตามเหมาะสม

ผลการรัน

- GET http://localhost:4000/users จะได้

```
[
  { "id": 1, "name": "Alice" },
  { "id": 2, "name": "Bob" }
]
```

- POST ด้วย JSON { "name": "Charlie" } เพิ่ม user ใหม่ และได้ response กลับเป็น user ใหม่

ตัวอย่างที่ 5: เว็บเซิร์ฟเวอร์พร้อม WebSocket Echo Server

โครงสร้างโปรเจกต์

```
app-websocket/
└── server.js
```

server.js

```
import { serve } from "bun";

const server = serve({
  port: 5000,
  websocket: {
    open(ws) {
      console.log("WebSocket connection opened");
      ws.send("Welcome to Bun WebSocket server!");
    },
    message(ws, message) {
      console.log("Received from client:", message);
      ws.send(`Echo: ${message}`);
    },
```

```
    close(ws, code, reason) {
      console.log(`WebSocket closed: ${code} - ${reason}`);
    }
  },
  fetch(request) {
    return new Response("Hello from HTTP server with WebSocket!");
  }
});

console.log("Server and WebSocket running on http://localhost:5000");
```

คำอธิบายโค้ด

- สร้าง HTTP server ที่มี WebSocket server รวมในตัว
- เมื่อ client เชื่อมต่อจะส่งข้อความต้อนรับ
- รับข้อความจาก client แล้วส่งกลับแบบ echo
- แสดงสถานะเมื่อ WebSocket ปิด

ผลการรัน

- เข้า http://localhost:5000 จะเห็นข้อความ HTTP
- ใช้ WebSocket client (เช่นใน browser devtools หรือ app อย่าง wscat) เชื่อมต่อและส่งข้อความ จะได้รับข้อความตอบกลับ

ตัวอย่างที่ 6: เว็บเซิร์ฟเวอร์พร้อม static file serving

โครงสร้างโปรเจกต์

```
app-static-server/
├── server.js
└── public/
    └── index.html
```

public/index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Bun Static Server</title>
</head>
<body>
  <h1>Welcome to Bun Static File Server!</h1>
```

```
</body>
</html>
```

server.js

```
import { serve } from "bun";
import fs from "fs";
import path from "path";

const publicDir = path.resolve("./public");

async function handler(request) {
  const url = new URL(request.url);
  let filePath = path.join(publicDir, url.pathname === "/" ? "/index.html" : url.pathname);

  try {
    const file = await fs.promises.readFile(filePath);
    const ext = path.extname(filePath).toLowerCase();
    let contentType = "text/plain";

    if (ext === ".html") contentType = "text/html";
    else if (ext === ".js") contentType = "application/javascript";
    else if (ext === ".css") contentType = "text/css";

    return new Response(file, {
      status: 200,
      headers: { "Content-Type": contentType }
    });
  } catch {
    return new Response("File Not Found", { status: 404 });
  }
}

serve({
  port: 6000,
  fetch: handler
```

```
});

console.log("Static file server running on http://localhost:6000");
```

คำอธิบายโค้ด

- สร้าง HTTP Server ที่ตอบไฟล์ static จากโฟลเดอร์ public
- อ่านไฟล์แบบ async ด้วย fs.promises.readFile
- กำหนด Content-Type ตามนามสกุลไฟล์
- ถ้าไม่เจอไฟล์ ส่ง 404

ผลการรัน

- เปิด http://localhost:6000 จะเห็นหน้า HTML ที่กำหนดใน index.html

---

ถ้าต้องการ ผมช่วยรวมไฟล์ทั้งหมดใน ZIP พร้อมคำสั่งรันและตั้งค่า VS Code ได้ครับ!
ได้ครับ! นี่คือคำอธิบายเชิงลึกและตัวอย่างโค้ดเรื่อง

การจัดการ Route และ Request/Response ใน Bun HTTP Server

แนวคิดพื้นฐาน

Bun HTTP Server ไม่มี built-in router แบบ Express หรือ Fastify เราจึงต้องจัดการ routing เอง ด้วยการตรวจสอบ

- URL path (request.url)
- HTTP method (request.method)
- ข้อมูลใน request body (ถ้ามี)

โดยใช้มาตรฐาน Fetch API ทั้งหมด ทำให้เราสามารถเขียนโค้ดแบบ asynchronous ได้ง่ายและยืดหยุ่น

การแยก path และ method

เราจะใช้

```
const url = new URL(request.url);
const path = url.pathname;
const method = request.method;
```

เพื่อดึง path และ method มาเช็คเงื่อนไข routing

ตัวอย่างการจัดการ route แบบง่าย

```
import { serve } from "bun";
```

```
serve({
  port: 3000,
  async fetch(request) {
    const url = new URL(request.url);
    const path = url.pathname;
    const method = request.method;

    if (path === "/") {
      return new Response("Welcome to Home Page", { status: 200 });
    } else if (path === "/about") {
      return new Response("About Page", { status: 200 });
    } else if (path === "/data" && method === "POST") {
      const body = await request.text();
      return new Response(`You posted: ${body}`, { status: 200 });
    } else {
      return new Response("Not Found", { status: 404 });
    }
  }
});

console.log("Server running on http://localhost:3000");
```

การจัดการ Request Body

- อ่านข้อมูล text: await request.text()
- อ่าน JSON: await request.json()
- อ่าน form data: await request.formData()

ซึ่งเป็น async function เพราะข้อมูลอาจจะส่งมาช้า

การตอบ Response แบบละเอียด

เราสามารถสร้าง Response ด้วย constructor

```
new Response(body, {
  status: 200,
  headers: {
    "Content-Type": "application/json"
```

```
  }
});
```

ตัวอย่างการตอบ JSON:

```
return new Response(JSON.stringify({ message: "Hello" }), {
  status: 200,
  headers: { "Content-Type": "application/json" }
});
```

ตัวอย่างโค้ดจัดการ Route พร้อม Request/Response (เชิงลึก)

```
import { serve } from "bun";

const users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];

async function handler(request) {
  const url = new URL(request.url);
  const path = url.pathname;
  const method = request.method;

  if (path === "/users") {
    if (method === "GET") {
      return new Response(JSON.stringify(users), {
        status: 200,
        headers: { "Content-Type": "application/json" }
      });
    } else if (method === "POST") {
      try {
        const data = await request.json();
        if (!data.name) {
          return new Response(JSON.stringify({ error: "Name required" }), { status: 400 });
        }
        const newUser = { id: users.length + 1, name: data.name };
```

```
        users.push(newUser);
        return new Response(JSON.stringify(newUser), {
          status: 201,
          headers: { "Content-Type": "application/json" }
        });
      } catch {
        return new Response(JSON.stringify({ error: "Invalid JSON" }), { status: 400 });
      }
    } else {
      return new Response("Method Not Allowed", { status: 405 });
    }
  }

  return new Response("Not Found", { status: 404 });
}

serve({
  port: 4000,
  fetch: handler
});

console.log("Server running on http://localhost:4000");
```

สรุป

- Bun ใช้ Fetch API ทั้ง request และ response
- Routing ต้องเขียนเช็ค URL และ method เอง
- รองรับการอ่าน request body แบบ async
- สร้าง response พร้อม status และ header ได้ละเอียด

ตัวอย่างโปรแกรม 3 ตัวอย่างพื้นฐาน และ 3 ตัวอย่างแนวประยุกต์ที่เน้น **การจัดการ route และ request/response ด้วย Bun HTTP Server** พร้อมโครงสร้างไฟล์, คำอธิบายโค้ด และผลการรัน

ตัวอย่างโปรแกรมพื้นฐาน 3 ตัว

ตัวอย่างที่ 1: Routing แบบง่าย ตอบข้อความตาม path

โครงสร้างโปรเจกต์

```
route-basic-1/
└── server.js
```

server.js

```
import { serve } from "bun";

serve({
  port: 3000,
  fetch(request) {
    const url = new URL(request.url);
    const path = url.pathname;

    if (path === "/") {
      return new Response("Welcome to Home Page", { status: 200 });
    } else if (path === "/about") {
      return new Response("About Page", { status: 200 });
    } else {
      return new Response("Page Not Found", { status: 404 });
    }
  }
});

console.log("Server running on http://localhost:3000");
```

คำอธิบายโค้ด

- เช็ค URL path และตอบข้อความตาม route ที่กำหนด
- ไม่สนใจ method และ request body ในตัวอย่างนี้
- ถ้า path ไม่ตรงกับที่กำหนด ส่ง 404

ผลการรัน

- รัน bun run server.js
- เข้า http://localhost:3000/ ได้ข้อความ "Welcome to Home Page"
- เข้า http://localhost:3000/about ได้ข้อความ "About Page"
- เข้า path อื่นใด ได้ข้อความ "Page Not Found"

ตัวอย่างที่ 2: Routing พร้อมตรวจสอบ HTTP Method

โครงสร้างโปรเจกต์

```
route-basic-2/
└── server.js
```

server.js

```
import { serve } from "bun";

serve({
  port: 3001,
  async fetch(request) {
    const url = new URL(request.url);
    const path = url.pathname;
    const method = request.method;

    if (path === "/data") {
      if (method === "GET") {
        return new Response("This is GET data", { status: 200 });
      } else if (method === "POST") {
        const body = await request.text();
        return new Response(`Received POST data: ${body}`, { status: 200 });
      } else {
        return new Response("Method Not Allowed", { status: 405 });
      }
    }

    return new Response("Not Found", { status: 404 });
  }
});

console.log("Server running on http://localhost:3001");
```

คำอธิบายโค้ด

- ตรวจสอบทั้ง path และ HTTP method
- รองรับ GET และ POST สำหรับ /data
- อ่าน body แบบ async ในกรณี POST

- ตอบ 405 ถ้า method ไม่รองรับ

ผลการรัน

- GET http://localhost:3001/data ตอบ "This is GET data"
- POST ไปที่เดียวกันส่งข้อความ จะตอบ "Received POST data: ..."
- method อื่น ๆ ได้ 405

ตัวอย่างที่ 3: REST API เบื้องต้น จัดการข้อมูลใน memory

โครงสร้างโปรเจกต์

```
route-basic-3/
└── server.js
```

server.js

```
import { serve } from "bun";

let todos = [
  { id: 1, task: "Learn Bun", done: false },
  { id: 2, task: "Build HTTP Server", done: true }
];

async function handler(request) {
  const url = new URL(request.url);
  const path = url.pathname;
  const method = request.method;

  if (path === "/todos") {
    if (method === "GET") {
      return new Response(JSON.stringify(todos), {
        status: 200,
        headers: { "Content-Type": "application/json" }
      });
    } else if (method === "POST") {
      try {
        const body = await request.json();
        if (!body.task) {
          return new Response(JSON.stringify({ error: "Task is required" }), { status: 400 });
```

```
        }
        const newTodo = { id: todos.length + 1, task: body.task, done: false };
        todos.push(newTodo);
        return new Response(JSON.stringify(newTodo), {
          status: 201,
          headers: { "Content-Type": "application/json" }
        });
      } catch {
        return new Response(JSON.stringify({ error: "Invalid JSON" }), { status: 400 });
      }
    } else {
      return new Response("Method Not Allowed", { status: 405 });
    }
  }

  return new Response("Not Found", { status: 404 });
}

serve({
  port: 3002,
  fetch: handler
});

console.log("Server running on http://localhost:3002");
```

คำอธิบายโค้ด

- สร้าง REST API สำหรับ /todos
- GET คืนรายการ todos เป็น JSON
- POST เพิ่ม todo ใหม่จาก JSON body
- ตรวจสอบ error เบื้องต้น เช่น ชุดข้อมูลไม่ครบถ้วน, JSON ไม่ถูกต้อง

ผลการรัน

- GET http://localhost:3002/todos ได้ todos JSON
- POST ส่ง JSON เช่น { "task": "New task" } เพิ่มรายการใหม่

ตัวอย่างโปรแกรมแนวประยุกต์ 3 ตัว

ตัวอย่างที่ 4: REST API จัดการ Users พร้อมการจัดการ route หลายแบบ

โครงสร้างโปรเจกต์

```
app-route-users/
└── server.js
```

server.js

```
import { serve } from "bun";

let users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];

async function handler(request) {
  const url = new URL(request.url);
  const path = url.pathname;
  const method = request.method;

  // GET /users - list all users
  if (path === "/users" && method === "GET") {
    return new Response(JSON.stringify(users), {
      status: 200,
      headers: { "Content-Type": "application/json" }
    });
  }

  // POST /users - add new user
  if (path === "/users" && method === "POST") {
    try {
      const data = await request.json();
      if (!data.name) {
        return new Response(JSON.stringify({ error: "Name is required" }), { status: 400 });
      }
      const newUser = { id: users.length + 1, name: data.name };
```

```
      users.push(newUser);
      return new Response(JSON.stringify(newUser), {
        status: 201,
        headers: { "Content-Type": "application/json" }
      });
    } catch {
      return new Response(JSON.stringify({ error: "Invalid JSON" }), { status: 400 });
    }
  }

  // GET /users/:id - get user by id
  if (path.startsWith("/users/") && method === "GET") {
    const id = parseInt(path.split("/")[2]);
    const user = users.find(u => u.id === id);
    if (user) {
      return new Response(JSON.stringify(user), {
        status: 200,
        headers: { "Content-Type": "application/json" }
      });
    }
    return new Response(JSON.stringify({ error: "User not found" }), { status: 404 });
  }

  return new Response("Not Found", { status: 404 });
}

serve({
  port: 4000,
  fetch: handler
});

console.log("User API server running on http://localhost:4000");
```

คำอธิบายโค้ด

- จัดการ route หลายแบบ

- GET /users คืน list users
- POST /users เพิ่ม user
- GET /users/:id คืน user ตาม id
- ใช้การตรวจสอบ path แบบ startsWith และแยก path เป็นส่วน ๆ

ผลการรัน

- GET http://localhost:4000/users คืน user list
- POST /users เพิ่ม user ใหม่
- GET /users/1 คืน user id=1 หรือ 404 ถ้าไม่มี

ตัวอย่างที่ 5: REST API พร้อมจัดการ HTTP Headers และ Status Codes

โครงสร้างโปรเจกต์

```
app-route-headers/
└── server.js
```

server.js

```
import { serve } from "bun";

const data = { message: "Hello Bun" };

serve({
  port: 5000,
  fetch(request) {
    const url = new URL(request.url);
    const path = url.pathname;

    if (path === "/") {
      return new Response("Welcome to Bun server", { status: 200 });
    }

    if (path === "/json") {
      return new Response(JSON.stringify(data), {
        status: 200,
        headers: {
          "Content-Type": "application/json",
          "X-Custom-Header": "BunServer"
```

```
        }
      });
    }

    if (path === "/redirect") {
      return new Response(null, {
        status: 302,
        headers: {
          Location: "https://bun.sh"
        }
      });
    }

    return new Response("Not Found", { status: 404 });
  }
});

console.log("Server running on http://localhost:5000");
```

คำอธิบายโค้ด

- ตอบข้อความธรรมดา, JSON พร้อม custom header
- ทำ redirect HTTP 302 ไปเว็บ Bun
- ใช้ HTTP status codes และ headers แบบต่าง ๆ

ผลการรัน

- เข้า / แสดงข้อความธรรมดา
- เข้า /json ได้ JSON พร้อม header X-Custom-Header
- เข้า /redirect redirect ไป bun.sh

ตัวอย่างที่ 6: API รองรับ Query Parameters และ POST Form Data

โครงสร้างโปรเจกต์

```
app-route-query-form/
└── server.js
```

server.js

```
import { serve } from "bun";
```

```
serve({
  port: 6000,
  async fetch(request) {
    const url = new URL(request.url);
    const path = url.pathname;
    const method = request.method;

    if (path === "/search" && method === "GET") {
      // ดึง query param ?q=
      const query = url.searchParams.get("q") || "";
      return new Response(`You searched for: ${query}`, { status: 200 });
    }

    if (path === "/submit" && method === "POST") {
      // อ่าน form data
      const formData = await request.formData();
      const name = formData.get("name") || "Unknown";
      return new Response(`Hello, ${name}`, { status: 200 });
    }

    return new Response("Not Found", { status: 404 });
  }
});

console.log("Server running on http://localhost:6000");
```

คำอธิบายโค้ด

- GET /search?q=term ดึง query string
- POST /submit อ่านข้อมูล form data จาก request
- ตอบข้อความตามข้อมูลที่รับมา

ผลการรัน

- GET http://localhost:6000/search?q=bun ตอบ "You searched for: bun"
- POST /submit ส่ง form data เช่น name=John ตอบ "Hello, John"

การใช้งาน WebSocket กับ Bun

1. Bun รองรับ WebSocket โดยตรง

Bun มี API สำหรับจัดการ WebSocket ทั้งฝั่ง server และ client ในตัว ทำให้เราสามารถสร้าง real-time communication ได้อย่างง่ายดายและรวดเร็ว โดยใช้มาตรฐาน WebSocket API ที่คล้ายกับ Browser

2. วิธีการสร้าง WebSocket Server ด้วย Bun

เราจะใช้ serve() จาก Bun เพื่อรัน HTTP Server และตรวจสอบ request ว่าเป็นการ upgrade protocol เพื่อสร้าง WebSocket connection หรือไม่

หลัก ๆ คือ

- เช็ค request.headers.get("upgrade") === "websocket"
- สร้าง WebSocket ด้วย new WebSocket(request) (ใน Bun)
- จัดการ event เช่น onmessage, onclose บน WebSocket

3. ตัวอย่างโค้ด WebSocket Server พื้นฐาน

```
import { serve } from "bun";

serve({
  port: 8080,
  async fetch(request) {
    // เช็คว่าเป็น WebSocket upgrade หรือไม่
    if (request.headers.get("upgrade") !== "websocket") {
      return new Response("Expected websocket", { status: 400 });
    }

    const ws = new WebSocket(request);

    ws.addEventListener("open", () => {
      console.log("WebSocket connection opened");
      ws.send("Welcome to Bun WebSocket Server!");
    });

    ws.addEventListener("message", (event) => {
      console.log("Received message:", event.data);
```

```
      // ตอบกลับข้อความเดิม
      ws.send(`Echo: ${event.data}`);
    });

    ws.addEventListener("close", () => {
      console.log("WebSocket connection closed");
    });

    return ws;
  }
});

console.log("WebSocket server running at ws://localhost:8080");
```

4. การทดสอบ WebSocket Server

เราสามารถทดสอบด้วย

- โปรแกรม client ด้วย JavaScript ในเบราว์เซอร์ (Console)
- โปรแกรมเช่น websocat หรือ Postman ที่รองรับ WebSocket

ตัวอย่าง client เบื้องต้นในเบราว์เซอร์ Console:

```
const ws = new WebSocket("ws://localhost:8080");

ws.onopen = () => {
  console.log("Connected to server");
  ws.send("Hello Bun!");
};

ws.onmessage = (event) => {
  console.log("Message from server:", event.data);
};

ws.onclose = () => {
  console.log("Connection closed");
};
```

5. สรุป

- Bun รองรับ WebSocket server ในตัว
- ใช้ new WebSocket(request) เพื่ออัพเกรด connection
- ติดตั้ง event listeners เพื่อรับส่งข้อมูลกับ client
- เหมาะกับแอป real-time เช่น chat, game, dashboard

นี่คือตัวอย่างโปรแกรม **WebSocket กับ Bun** จำนวน 3 ตัวอย่างพื้นฐาน และ 3 ตัวอย่างแนวประยุกต์ พร้อมโครงสร้างไฟล์, คำอธิบายโค้ด, และผลการรัน

ตัวอย่างโปรแกรมพื้นฐาน 3 ตัวอย่าง: การใช้งาน WebSocket กับ Bun

---

ตัวอย่างที่ 1: WebSocket Echo Server

โครงสร้างโปรเจกต์

```
websocket-basic-1/
└── server.js
```

server.js

```
import { serve } from "bun";

serve({
  port: 8080,
  async fetch(request) {
    if (request.headers.get("upgrade") !== "websocket") {
      return new Response("Expected websocket", { status: 400 });
    }

    const ws = new WebSocket(request);

    ws.addEventListener("open", () => {
      console.log("WebSocket connection opened");
      ws.send("Welcome to Bun WebSocket Echo Server!");
    });

    ws.addEventListener("message", (event) => {
      console.log("Received message:", event.data);
```

```
      ws.send(`Echo: ${event.data}`);
    });

    ws.addEventListener("close", () => {
      console.log("WebSocket connection closed");
    });

    return ws;
  }
});

console.log("WebSocket Echo Server running at ws://localhost:8080");
```

คำอธิบายโค้ด

- ตรวจสอบ header ว่าเป็น WebSocket upgrade request หรือไม่
- สร้าง WebSocket connection จาก request
- ฟัง event open เพื่อส่งข้อความต้อนรับ
- ฟัง event message เพื่อรับข้อความและตอบกลับ echo
- ฟัง event close เพื่อ log ข้อความเมื่อ connection ปิด

ผลการรัน

- รัน bun run server.js
- client เชื่อมต่อ WebSocket ที่ ws://localhost:8080
- รับข้อความต้อนรับ และข้อความที่ส่งกลับทุกข้อความที่ client ส่งไป

ตัวอย่างที่ 2: WebSocket Broadcast Server

โครงสร้างโปรเจกต์

```
websocket-basic-2/
└── server.js
```

server.js

```
import { serve } from "bun";

const clients = new Set();

serve({
  port: 8081,
```