Bun.js

Web Programming:

Advance

(Integrative-Generative AI Edition)

Key Contents

Student Price Book Center

คำนำ

ในยุคดิจิทัลปัจจุบัน การพัฒนาเว็บแอปพลิเคชันที่รวดเร็ว มีประสิทธิภาพ และสามารถรองรับผู้ใช้จำนวนมาก เป็นสิ่งจำเป็นสำหรับนักพัฒนาและองค์กรทุกขนาด Bun.js ได้กลายเป็นหนึ่งใน runtime ที่น่าสนใจ ด้วยความเร็วในการประมวลผลสูง, การสนับสนุน JavaScript และ TypeScript, และความสามารถในการทำงานร่วมกับเครื่องมือและ frameworks ยอดนิยม ทำให้ Bun.js เป็นทางเลือกที่น่าสนใจสำหรับการสร้างแอปพลิเคชันสมัยใหม่

หนังสือเล่มนี้มุ่งเน้นให้ผู้อ่านสามารถ **ต่อยอดความรู้จากพื้นฐานไปสู่ระดับ Advance** โดยเริ่มตั้งแต่การ build และ bundle แอปพลิเคชันขนาดใหญ่ การจัดการ bundle, code splitting, tree shaking และ dead code elimination ล้วนเป็นแนวทางสำคัญในการสร้างแอปที่โหลดเร็วและมีประสิทธิภาพ บทที่ 14 จะช่วยให้ผู้อ่านเข้าใจการตั้งค่า build pipeline ด้วย Bun รวมถึงการจัดการ assets และ static files พร้อมตัวอย่างบูรณาการเพื่อให้เห็นภาพการใช้งานจริง

การ deploy แอปเป็นอีกหนึ่งหัวใจสำคัญ หนังสือเล่มนี้จะพาผู้อ่านเข้าสู่การเตรียมโปรเจกต์สำหรับ production การ deploy บน cloud platform ยอดนิยม เช่น Vercel, Netlify หรือ Bun.sh การตั้งค่า CDN และ cache control รวมถึงการจัดการ environment production บทที่ 15 เน้นทั้งแนวคิดและตัวอย่างเชิงปฏิบัติ เพื่อให้ผู้อ่านสามารถ deploy แอป Bun ได้อย่างปลอดภัย รวดเร็ว และมีเสถียรภาพ

ประสิทธิภาพของแอปพลิเคชันคือปัจจัยสำคัญของผู้ใช้และธุรกิจ บทที่ 16 จะเน้นการทำ **performance tuning และ profiling** ด้วย Bun Profiler การแก้ไขปัญหาคอขวด การ optimize memory และ CPU usage รวมถึงเทคนิคการเขียนโค้ดให้เร็วที่สุดบน Bun ช่วยให้แอปสามารถตอบสนองรวดเร็ว รองรับผู้ใช้จำนวนมาก และใช้ resource อย่างมีประสิทธิภาพสูงสุด

การทำงานร่วมกับ frameworks ยอดนิยมช่วยให้การพัฒนาแอปเต็มรูปแบบง่ายขึ้น บทที่ 17 จะสอนการ integrate Bun กับ **React, Next.js, Astro** และ backend frameworks เช่น Fastify และ Koa รวมถึงการจัดการ **SSR และ hydration** เพื่อให้หน้าเว็บ render อย่างรวดเร็วและ interactive พร้อมตัวอย่าง full-stack project ที่ช่วยให้ผู้อ่านเห็นภาพการพัฒนาในโลกจริง

การ maintain และ scale โปรเจกต์เป็นสิ่งสำคัญเมื่อทีมพัฒนาและแอปเติบโต บทที่ 18 ครอบคลุมการจัดโครงสร้างโปรเจกต์ที่ดี การจัดการ dependencies และ version control เทคนิค refactoring และ code review รวมถึงการเตรียมโปรเจกต์สำหรับทีม เพื่อให้โค้ด maintainable, readable และพร้อมรองรับการเติบโตของผู้ใช้และทีมพัฒนา

คุณภาพของซอฟต์แวร์ขึ้นอยู่กับการทดสอบและ workflow อัตโนมัติ บทที่ 19 จะเน้น **integration test และ end-to-end (E2E) test** การตั้งค่า CI/CD pipeline ด้วย GitHub Actions และ GitLab CI การทำ automation tests และ deployment รวมถึงการ monitor คุณภาพโค้ด เพื่อให้ทีมพัฒนาสามารถ release ฟีเจอร์ใหม่ได้อย่างมั่นใจและรวดเร็ว

ความปลอดภัยเป็นสิ่งที่ไม่สามารถละเลยได้ บทที่ 20 จะสอนการจัดการ **security vulnerabilities, การตั้งค่า CORS, CSP และ headers ปลอดภัย**, การป้องกัน injection, XSS และ CSRF รวมถึงแนวทางการเขียนโค้ดที่ปลอดภัยบน Bun.js การปฏิบัติตาม best practices เหล่านี้ทำให้โปรเจกต์สามารถป้องกันความเสี่ยงและรักษาความน่าเชื่อถือของผู้ใช้ได้

หนังสือเล่มนี้ไม่ได้เน้นเพียงแค่ทฤษฎี แต่ยังสอดแทรก **ตัวอย่างบูรณาการและโค้ดเชิงปฏิบัติ** เพื่อให้ผู้อ่านสามารถลงมือทำตามได้จริง และปรับใช้เทคนิคเหล่านี้กับโปรเจกต์จริงของตน การเรียนรู้ผ่านตัวอย่างจริงช่วยให้เข้าใจแนวคิดและกระบวนการต่าง ๆ ได้อย่างรวดเร็วและลึกซึ้ง

ท้ายที่สุด หนังสือ **Bun.js Web Programming: Advance** เล่มนี้ถูกออกแบบมาเพื่อเป็นคู่มือสำหรับนักพัฒนาที่ต้องการพัฒนาทักษะและสร้างแอปพลิเคชัน Bun ที่เร็ว, เสถียร, ปลอดภัย และปรับขนาดได้ ผู้อ่านจะได้รับทั้งความรู้เชิงทฤษฎี เทคนิคเชิงปฏิบัติ และแนวทาง best practices เพื่อให้สามารถสร้างแอปพลิเคชันสมัยใหม่ที่ตอบโจทย์ผู้ใช้และธุรกิจได้อย่างมั่นใจ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 14

การ Build และ Bundle แอปพลิเคชันขนาดใหญ่

(Build and Bundle)

เนื้อหา

บทนำ – การ Build และ Bundle แอปพลิเคชันขนาดใหญ่

ในโลกของการพัฒนาเว็บแอปพลิเคชันสมัยใหม่ การจัดการกับโปรเจกต์ขนาดใหญ่ถือเป็นความท้าทายสำคัญ เนื่องจากแอปพลิเคชันเหล่านี้มักประกอบด้วยโค้ดจำนวนมากและไฟล์ assets ที่หลากหลาย การ build และ bundle โค้ดอย่างมีประสิทธิภาพจึงเป็นสิ่งจำเป็นเพื่อให้การทำงานราบรื่นและการ deploy เป็นไปอย่างรวดเร็ว

หนึ่งในเทคนิคสำคัญคือ **การจัดการ bundle ขนาดใหญ่และ code splitting** ซึ่งช่วยให้โค้ดที่ผู้ใช้ต้องโหลดในแต่ละครั้งมีขนาดเล็กลง ส่งผลให้เวลาในการโหลดหน้าเว็บลดลงและประสบการณ์ผู้ใช้ดีขึ้น การแบ่ง bundle อย่างเหมาะสมยังช่วยให้นักพัฒนาสามารถอัปเดตส่วนต่าง ๆ ของแอปโดยไม่กระทบทั้งระบบ

นอกจากนี้ การ **optimize tree shaking และ dead code elimination** เป็นเทคนิคที่ช่วยลดขนาดไฟล์ bundle โดยการตัดโค้ดที่ไม่ได้ใช้ออกไป ทำให้โค้ดสุดท้ายมีประสิทธิภาพมากขึ้น และยังช่วยประหยัดแบนด์วิธและทรัพยากรของผู้ใช้ การประยุกต์ใช้เทคนิคเหล่านี้ต้องอาศัยความเข้าใจเชิงลึกเกี่ยวกับโครงสร้างโมดูลของภาษาและเครื่องมือ build

การตั้งค่า **build pipeline ด้วย Bun** เป็นอีกหัวข้อที่สำคัญ เนื่องจาก Bun นำเสนอความเร็วในการ compile และ bundle ที่เหนือกว่าเครื่องมือเดิม ๆ ด้วยการใช้ runtime แบบ high-performance ทำให้นักพัฒนาสามารถสร้าง pipeline ที่รวดเร็วและยืดหยุ่น รองรับทั้งการพัฒนาแบบ local และการ deploy ใน production environment

นอกจากโค้ดแล้ว การ **จัดการ assets และ static files** เช่น รูปภาพ ไอคอน fonts และไฟล์ CSS/JS ก็เป็นสิ่งที่ต้องคำนึงถึง การจัดระเบียบ assets อย่างมีระบบช่วยให้ build process มีประสิทธิภาพ และลดปัญหาไฟล์ขาดหายหรือโหลดช้าในระหว่างการใช้งานจริง

บทนี้ยังเน้นไปที่แนวทางการบูรณาการเทคนิคต่าง ๆ เข้าด้วยกัน เพื่อให้ได้ pipeline ที่ครบวงจร ตั้งแต่การเขียนโค้ด การจัดการ dependency การ optimize bundle จนถึงการ deploy แอปพลิเคชันขนาดใหญ่ เทคนิคเหล่านี้ไม่เพียงช่วยลดขนาดไฟล์และเพิ่มความเร็ว แต่ยังช่วยให้นักพัฒนาสามารถ maintain ระบบได้ง่ายและยั่งยืน

ท้ายที่สุด การเข้าใจแนวคิดและเครื่องมือในบทนี้เป็นพื้นฐานสำคัญสำหรับนักพัฒนาที่ต้องการสร้างแอปพลิเคชันขนาดใหญ่ที่มีประสิทธิภาพและสามารถปรับขนาดได้ บทนี้จะพาผู้อ่านสำรวจทั้งทฤษฎีและตัวอย่างเชิงปฏิบัติ เพื่อให้สามารถนำความรู้ไปปรับใช้กับโปรเจกต์จริงได้อย่างมั่นใจ

การ Build และ Bundle แอปพลิเคชันขนาดใหญ่

- การจัดการ bundle ขนาดใหญ่และ code splitting
- การ optimize tree shaking และ dead code elimination
- การตั้งค่า build pipeline ด้วย Bun
- การจัดการ assets และ static files

---

1 การจัดการ Bundle ขนาดใหญ่ และ Code Splitting

ปัญหา

- แอปพลิเคชันใหญ่ ๆ มักมีโค้ดหลายหมื่นบรรทัด
- ถ้ารวมทุกอย่างเป็นไฟล์เดียว (Single Bundle) จะโหลดช้า และใช้แรมเยอะในตอนเริ่มต้น
- จำเป็นต้อง **แยกโค้ด (Code Splitting)** เพื่อให้โหลดเฉพาะส่วนที่จำเป็น

แนวคิด Code Splitting ใน Bun

- Bun รองรับ **ES Modules** ทำให้สามารถใช้ **Dynamic Import** ได้ทันที
- Bun bundler จะสร้างไฟล์แยก (chunks) ให้โดยอัตโนมัติ
- เหมาะกับ SPA, SSR, หรือแม้แต่ CLI tools

ตัวอย่าง:

```
// main.ts
console.log("Main app loaded");

// โหลดโมดูลเฉพาะเมื่อกดปุ่ม
document.getElementById("load-report")?.addEventListener("click", async () => {
  const { generateReport } = await import("./report");
```

```
  generateReport();
});
// report.ts
export function generateReport() {
  console.log("Report module loaded!");
}
```

ผลลัพธ์เมื่อ Build:

- Bun จะสร้างไฟล์ bundle สำหรับ main.js
- และสร้าง chunk แยกของ report.js
- โหลด report.js เฉพาะตอนที่ผู้ใช้ต้องการ

ข้อดี:

- ลดขนาด initial bundle
- เร็วขึ้นในหน้าแรก (First Load Time)

---

2 การ Optimize Tree Shaking และ Dead Code Elimination

ความหมาย

- **Tree Shaking**: ลบโค้ดที่ไม่ได้ถูกใช้งานออกจาก bundle
- **Dead Code Elimination**: ลบโค้ดที่ไม่มีทางรันได้ เช่น:

```
if (false) {
  console.log("จะไม่มีวันรัน");
}
```

ใน Bun.js

- Bun ใช้ **esbuild-like optimization** ภายใน
- รองรับการลบโค้ดที่ไม่ได้ใช้ โดยอิงจาก **ESM imports**
- การเขียนโค้ดให้ **pure** และใช้ import แบบเจาะจง จะช่วย Tree Shaking ได้มากขึ้น

ตัวอย่างการเขียนที่ช่วย Tree Shaking:

```
// utils.ts
export function usedFunc() {
  console.log("This is used");
}
export function unusedFunc() {
  console.log("This is never used");
}
// app.ts
```

```
import { usedFunc } from "./utils";
usedFunc();
```

เมื่อ build:

- Bun จะลบ unusedFunc ออกไปโดยอัตโนมัติ

Tips:

- ใช้ --minify เพื่อลดโค้ดและเปิดการ optimize เพิ่มเติม
- หลีกเลี่ยงการใช้ require() แบบ dynamic ถ้าอยากให้ Tree Shaking ทำงานเต็มที่

---

3 การตั้งค่า Build Pipeline ด้วย Bun

เหตุผล

- แอปใหญ่ต้องมีขั้นตอน build ที่ชัดเจน เช่น transpile TS ⟶ bundle ⟶ minify ⟶ copy assets
- Bun รวมทุกขั้นตอนนี้ได้ในตัว ไม่ต้องติดตั้ง webpack, babel, rollup แยก

โครงสร้างโปรเจกต์ตัวอย่าง:

```
project/
  src/
    index.ts
    components/
      App.tsx
  public/
    images/
  bunfig.toml
```

bunfig.toml (ตั้งค่า Build Pipeline):

```
[build]
entrypoints = ["src/index.ts"]
outdir = "dist"
minify = true
target = "browser"
splitting = true
```

คำสั่ง build:

```
bun build
```

สิ่งที่ Bun ทำอัตโนมัติ:

1. Compile TypeScript/JSX
2. Bundle และทำ code splitting

3. Minify
4. สร้างไฟล์ใน dist/

---

4 การจัดการ Assets และ Static Files

ปัญหา

- แอปขนาดใหญ่มีไฟล์ static เยอะ เช่น รูปภาพ, CSS, ฟอนต์
- ต้องการวิธี copy หรือ serve assets อย่างรวดเร็ว

ใน Bun.js

- Bun ไม่ต้องมี server middleware แยก (เช่น Express static)
- สามารถ serve assets ตรงจาก fetch ได้
- หรือใช้ **static directory** แยก และ build script copy อัตโนมัติ

ตัวอย่าง Bun HTTP Server ที่ serve static:

```
import { readFile } from "fs/promises";
import path from "path";

const publicDir = path.join(import.meta.dir, "public");

export default {
  port: 3000,
  async fetch(req: Request) {
    const url = new URL(req.url);
    const filePath = path.join(publicDir, url.pathname);

    try {
      const file = await readFile(filePath);
      return new Response(file);
    } catch {
      return new Response("Not Found", { status: 404 });
    }
  }
};
```

ใน Build Pipeline:

- ใช้ script หรือ bun build --copy-assets เพื่อคัดลอกไฟล์จาก public/ ⟶ dist/public/
- Bun bundler จะไม่รวมไฟล์ static เข้า bundle โดยตรง ยกเว้นใช้ import.meta.url เพื่อ embed

☐ **สรุปเทคนิค Build & Bundle สำหรับโปรเจกต์ใหญ่ใน Bun.js**

1. **ใช้ Code Splitting** เพื่อลดขนาด initial load
2. **เขียนโค้ดให้ Tree Shaking ทำงานง่าย** เช่นใช้ ESM imports
3. **ตั้งค่า bunfig.toml** เพื่อกำหนด build pipeline ที่ชัดเจน
4. **จัดการ assets ให้เหมาะสม** ทั้ง serve ตรงและ copy ตอน build
5. ใช้ --minify และ --target=browser เพื่อให้โค้ดขนาดเล็กและเหมาะกับ production

---

บทที่ 14: การ Build และ Bundle แอปพลิเคชันขนาดใหญ่ด้วย Bun

บทนี้ลงลึกเรื่องสถาปัตยกรรมการบันเดิลสำหรับโปรเจ็กต์ใหญ่ ตั้งแต่การแยกโค้ด (code splitting), tree-shaking/dead-code elimination, ไปจนถึงการตั้งค่า build pipeline และการจัดการ assets/static อย่างเป็นระบบด้วย **Bun.build** และ CLI ของ bun build

---

1) จัดการ bundle ใหญ่และ Code Splitting

เมื่อไรควรแยกโค้ด

- เส้นทาง (route) ใหญ่ ๆ, หน้าฟีเจอร์ที่ใช้น้อย, editor/viewer ขนาดใหญ่, หรือ dependency กลุ่ม “rarely used”
- ช่วยลด **TTI**/First Load และเพิ่ม **cache efficiency** (ไฟล์หลักเล็กลง เปลี่ยนบ่อยน้อยลง)

วิธีทำใน Bun

- เปิด **chunk splitting** ด้วย splitting: true (API) หรือ --splitting (CLI)
- ใช้ import() แบบไดนามิกเพื่อแตก chunk ตาม boundary ที่คุณกำหนดเอง
- หลาย entrypoints ช่วยให้ได้ไฟล์ entry + shared chunks แยกกัน

ตัวอย่าง

```
// src/main.ts (entry หลัก)
document.getElementById("btn-report")!.addEventListener("click", async () => {
  const { renderReport } = await import("./features/report");
  renderReport();
});
// build.ts (สคริปต์ build แบบ programmatic)
await Bun.build({
  entrypoints: ["src/main.ts"],
  outdir: "dist",
```

```
  splitting: true,
  format: "esm",
  sourcemap: "linked",
  minify: { whitespace: true, identifiers: true, syntax: true },
});
```

splitting, minify, sourcemap, format, entrypoints, outdir เป็นออปชันใน **BuildConfig** ของ Bun.build. (Bun)

หมายเหตุ: เมื่อมีหลาย entrypoints Bun จะสร้างไฟล์ entry + shared chunk ให้อัตโนมัติ และสามารถตั้งรูปแบบชื่อไฟล์ได้ด้วย naming.entry, naming.chunk, naming.asset (เช่นใส่ content hash) เพื่อช่วย cache/CDN ที่ปลอดภัยขึ้น. (Bun)

---

2) Tree Shaking และ Dead Code Elimination (DCE)

ความต่าง (สรุปสั้น)

- **Tree shaking**: วิเคราะห์กราฟการอิมพอร์ตเพื่อตัดฟังก์ชัน/ตัวแปรที่ “ไม่ถูกใช้” ออก
- **Dead code elimination**: ลบโค้ดที่ “ไม่มีวันรัน” (เช่นสาขา if ที่คอมไพเลอร์พิสูจน์ว่า false เสมอ) หรือเรียกที่ถูก drop

สิ่งที่ Bun ทำให้

- ตัวโหลด js/ts/tsx/jsx ของ Bun ใช้ **DCE + tree shaking** เป็นค่าเริ่มต้นเวลา bundle เลย (ไม่มีต้องตั้งค่าเพิ่ม) ซึ่งช่วยให้ output สะอาดและเล็กลงตั้งแต่แรก. (Bun)

เทคนิคเสริมเพื่อให้สั่นแรงขึ้น

- ตั้ง minify (โดยเฉพาะ syntax/identifiers/whitespace) เพื่อเปิดทางให้ DCE ทำงานลึกขึ้น. (Bun)
- ใช้ annotation/เมทาดาทาที่เป็นที่รู้จักของ bundler:
  - ใส่ /* @__PURE__ */ บนฟังก์ชันที่ side-effect-free เพื่อให้ถูกตัดเวลาผลลัพธ์ไม่ถูกใช้
  - package.json ของไลบรารี: sideEffects: false (หรือระบุตารางไฟล์ที่มีผลข้างเคียง)
- ตัวเลือกขั้นสูงใน Bun:
  - emitDCEAnnotations: บังคับใส่ @PURE annotation แม้จะ minify แล้ว
  - ignoreDCEAnnotations: ข้าม annotation/sideEffects (มีไว้เป็น workaround เมื่อไลบรารีระบุผิด)
  - drop: ระบุคอลล์ที่ให้ลบทิ้ง (เช่น console.*, debug) ระหว่างบันเดิล
    ทั้งหมดนี้อยู่ใน **BuildConfig**. (Bun)

---

3) ตั้งค่า Build Pipeline ด้วย Bun

ทางเลือก 2 สาย: CLI vs API

- **CLI**: ตรงไปตรงมา เหมาะกับแอปทั่วไป/CI
- bun build src/main.ts --outdir=dist --splitting --format=esm --minify --sourcemap=linked
- **API (Bun.build)**: ยืดหยุ่นสำหรับงานซับซ้อน (หลายทาร์เก็ต, ปลั๊กอิน, เงื่อนไข)
- // build.ts
- const client = Bun.build({
- entrypoints: ["src/client/index.html"], // HTML ก็เป็น entry ได้
- outdir: "dist/client",
- minify: true,
- splitting: true,
- publicPath: "/static/",
- naming: { entry: "[name]-[hash].js", chunk: "chunks/[name]-[hash].js", asset: "assets/[name]-[hash][ext]" },
- });
-
- const server = Bun.build({
- entrypoints: ["src/server/index.ts"],
- outdir: "dist/server",
- target: "bun", // โค้ดฝั่งเซิร์ฟเวอร์รันบน Bun runtime
- packages: "external", // ไม่ bundle node_modules เพื่อลดขนาด
- format: "esm",
- sourcemap: "external",
- minify: { syntax: true, whitespace: true, identifiers: true },
- drop: ["console.debug", "console.trace"],
- });

ออปชันอย่าง packages: "external", publicPath, naming, target, sourcemap, drop มีใน **BuildConfig** ของ Bun. (Bun)

Watch/Dev & แยกขั้นตอนที่เหมาะสม

- ใช้ bun build --watch ระหว่างพัฒนา/ทำไลบรารี (เร็วมาก) และเมื่อพร้อมค่อย build โปรดักชัน. (Bun)
- ใช้ tsc --noEmit แยกขั้นตอน type-check (Bun ไม่ทำ typecheck ระหว่าง build) แล้วให้ Bun.build ทำหน้าที่ทรานส์ไพล์/บันเดิล. (Bun)
- **Plugins** ตอนนี้:

- ในโหมด production build ผ่าน CLI ยัง *จำกัด* — ปลั๊กอินเต็ม ๆ ใช้ผ่าน API (Bun.build) หรือ dev server ผ่าน bunfig.toml. (Bun)

Target/Format/Bytecode (งานฝั่งเซิร์ฟเวอร์)

- target: "bun" + format: "cjs" เปิดทางให้ bytecode: true เพื่อเร่ง cold-start (แลกกับไฟล์ใหญ่ขึ้นเล็กน้อย) — ใช้กับงาน serverless/Bun-hosted ได้ผลดีมาก. (Bun)

---

4) การจัดการ Assets และ Static Files

HTML/CSS/Assets แบบ one-stop

- ให้ **HTML** เป็น entry แล้ว bun build จะ:
  - bundle & hash <script src>, <link rel="stylesheet">
  - คัดลอก/แฮชรูปภาพ วิดีโอ ฟอนต์ ฯลฯ และเขียนพาธใหม่ให้อัตโนมัติ
- bun build ./index.html --outdir=dist --minify
  สิ่งนี้ทำให้สร้าง static site/landing page ได้ *zero-config* และยังรองรับหลาย HTML entry/glob ได้ด้วย. (Bun)
- **CSS**:
  - Bun มี CSS bundler ในตัว (minify, CSS Modules, transpile ฟีเจอร์สมัยใหม่, vendor prefixing) และรวม CSS จาก @import/import ใน JS เป็นไฟล์ต่อ entry ได้อัตโนมัติ. (Bun)

File & HTML Loader (สำคัญกับ CDN/caching)

- **file loader**: เวลา import logo from "./logo.svg" ตอน build Bun จะคัดลอกไฟล์ไปที่ outdir และคืนค่าเป็นพาธของไฟล์นั้น (เปลี่ยนเป็น absolute/relative ตาม publicPath). ชื่อไฟล์กำหนดด้วย naming.asset. (Bun)
- **publicPath**: ตั้ง prefix เส้นทางไฟล์ให้ออกมาชี้ไป CDN เช่น /static/ หรือ https://cdn.example.com/ ได้ทันที (ใช้ได้กับ asset ทั้งหมด) — ทำงานร่วมกับ naming.asset ที่มี content hash เพื่อ cache-busting. (Bun)
- **HTML loader**: สแกน <img>, <link>, <script> ฯลฯ แล้ว bundle/copy/hash และ rewrite พาธให้; ใน full-stack build (--target=bun) การ import HTML จะให้ **manifest object** เพื่อให้ Bun.serve เสิร์ฟไฟล์ที่บันเดิลไว้ล่วงหน้าได้มีประสิทธิภาพ. (Bun)

เสริม: เสิร์ฟ static ในโปรดักชันฝั่ง Bun

```
// ตัวอย่างเสิร์ฟไฟล์ที่ build แล้ว
const server = Bun.serve({
  port: 3000,
  fetch(req) {
    const url = new URL(req.url);
```

```
    if (url.pathname.startsWith("/static/")) {
      return new Response(Bun.file(`dist${url.pathname}`));
    }
    return new Response("OK");
  },
});
```

ใน full-stack build ที่ import HTML เป็น entry, Bun จะให้ข้อมูลแม็พไฟล์เพื่อเสิร์ฟ asset ที่บันเดิลไว้ได้สะดวกยิ่งขึ้น. (Bun)

---

5) รูปแบบการตั้งชื่อไฟล์ (naming) และ Source Map

- ปรับแต่ง naming ได้ละเอียด:
- naming: {
- entry: "[name]-[hash].js",
- chunk: "chunks/[name]-[hash].js",
- asset: "assets/[name]-[hash][ext]"
- },
- publicPath: "/static/",
- sourcemap: "linked" // หรือ "external" สำหรับ CI/prod

  ช่วยให้:
  - **Long-term caching** ด้วย content hash
  - จัดโครง dist/ ให้เป็นหมวดหมู่ (chunks/assets) ชัดเจน
  - เลือกชนิด source map ให้เหมาะกับ production/debug

    (ตัวเลือก sourcemap: "none" | "linked" | "external" | "inline"). (Bun)

---

6) ข้อควรระวัง/แนวปฏิบัติสำหรับโปรเจ็กต์ใหญ่

โค้ด & Dependency

- หลีกเลี่ยง “barrel file” ที่ export ทุกอย่างแบบรวมไฟล์ เพราะอาจลดประสิทธิภาพ tree-shaking
- ชอบ import { fn } from "lib/subpath" มากกว่า import * as lib from "lib" ในกรณีไลบรารีที่ side-effects ไม่ชัด
- ตั้ง sideEffects ให้ถูกในไลบรารีของคุณเอง
- แยก boundary ด้วย import() ตามจุดที่ผู้ใช้กด/เส้นทาง/โหมดแอดมิน

บิลด์

- สำหรับ **server build**: ใช้ packages: "external" ลดขนาดบันเดิล (โหลดจาก node_modules ตอนรันแทน) เว้นแต่คุณต้องการ single-file artifact จริง ๆ. (Bun)
- ใช้ drop: ["console.*"] เพื่อลบคอลล์ที่ไม่จำเป็นออกจากโปรดักชัน. (Bun)
- ใช้ bytecode: true (พร้อมเงื่อนไข target: "bun" + format: "cjs") สำหรับงาน server ที่ต้องการ cold-start ที่ไวมาก. (Bun)

Assets/CDN

- ตั้ง publicPath ให้ตรงกับเส้นทาง CDN จริง และตรวจสอบว่า reverse proxy ส่ง Cache-Control ที่เหมาะสม
- ใช้ naming.asset + content hash เสมอ เพื่อ cache-busting

DX/CI

- แยก step: tsc --noEmit (typecheck) + bun build (bundle)
- เปิด --watch ใน local dev เท่านั้น; ใน CI ทำโปรดักชัน build ครั้งเดียว
- ตรวจจับ “ข้าม DCE” ด้วยการดูขนาด bundle เปรียบเทียบก่อน/หลังเปิด minify และระบุ drop

---

7) สรุปภาพรวม Workflow แนะนำ

1. **พัฒนา**
   - bun index.html หรือ dev server/MPA; import CSS/asset ตามปกติ
   - ใช้ import() วาง boundary การโหลดแบบ lazy สำหรับฟีเจอร์ใหญ่
   - bun build --watch เมื่อต้องทดสอบ bundle ใน local. (Bun)
2. **โปรดักชัน (Client)**
   - bun build ./index.html --outdir=dist --minify --splitting --sourcemap=linked
   - ตั้ง naming.* + publicPath ให้เหมาะกับ CDN. (Bun)
3. **โปรดักชัน (Server)**
   - Bun.build({ target: "bun", packages: "external", drop: ["console.*"], sourcemap: "external" })
   - พิจารณา bytecode: true ถ้าต้องการ cold-start ไว. (Bun)

การจัดการ bundle ขนาดใหญ่และ code splitting

---

1 ปัญหาของ Bundle ขนาดใหญ่

ในโปรเจกต์ใหญ่ เช่น SPA, dashboard, หรือระบบที่รวมฟีเจอร์เยอะ

- ถ้าเราบันเดิลโค้ดทั้งหมดเป็นไฟล์เดียว (single bundle)
  - **Initial Load Time** (เวลาโหลดครั้งแรก) จะนาน

- **TTI** (Time to Interactive) ช้า
- ไฟล์ใหญ่ทำให้ **cache invalidation** บ่อย (เปลี่ยนโค้ดนิดเดียวต้องโหลดไฟล์ใหม่ทั้งก้อน)

- สิ่งนี้กระทบ **UX** โดยตรง เพราะผู้ใช้ต้องรอนาน

 วิธีแก้คือ **Code Splitting** — การแยกโค้ดออกเป็น chunk ย่อย ๆ และโหลดเมื่อจำเป็น

---

2 Code Splitting ใน Bun.js

Bun รองรับการทำ code splitting ได้ 2 แบบหลัก

2.1 Automatic Splitting ด้วย splitting: true

ถ้าเปิด splitting ตอน build:

bun build src/main.ts --outdir dist --splitting --format esm

Bun จะ:

- วิเคราะห์ dependency graph
- แยกไฟล์ library/common ออกเป็น chunk กลาง
- ทำให้ entry หลัก (เช่น main.ts) มีขนาดเล็กลง
- ไฟล์ chunk จะถูกโหลดอัตโนมัติเมื่อมีการอ้างถึง

2.2 Manual Splitting ด้วย Dynamic Import

ใช้ ES dynamic import (import()) เพื่อบอก Bun ว่าให้โหลดโค้ดส่วนนี้แยกไฟล์

```
// src/main.ts
document.getElementById("btn-report")?.addEventListener("click", async () => {
  const { renderReport } = await import("./features/report");
  renderReport();
});
// src/features/report.ts
export function renderReport() {
  console.log("Report loaded!");
}
```

เมื่อ build ด้วย --splitting:

- main.ts จะเป็นไฟล์หลัก
- report.ts จะถูกบันเดิลเป็น chunk แยก (report-XXXX.js)
- โหลดเฉพาะตอนกดปุ่ม

---

3 แนวทางการจัดการ Bundle ขนาดใหญ่

1. **แยกตาม Route/Page**

   ถ้าใช้ SPA หรือ Framework เช่น React Router:

2. const Page = lazy(() => import('./pages/Page'));

   แต่ละหน้าจะกลายเป็น chunk แยก

3. **แยกตาม Feature Module**

   แยก logic หรือ component กลุ่มใหญ่ เช่น analytics, admin, report

4. **แยก Third-party Libraries**

   บาง library ใหญ่ เช่น Chart.js, Three.js ควรโหลดแยก:

5. const { Chart } = await import('chart.js');

6. **ใช้ Multiple Entry Points**

   ตั้งหลาย entry ให้ Bun:

7. bun build src/admin.ts src/client.ts --outdir dist --splitting

---

4 เทคนิคเพิ่มประสิทธิภาพ

- เปิด minify เพื่อลดขนาดไฟล์:
- bun build src/main.ts --splitting --minify --outdir dist
- ใช้ naming.chunk และ naming.asset ใส่ content hash เพื่อ cache busting:
- naming: { chunk: "chunks/[name]-[hash].js" }
- วาง chunk ไว้ใน /static/ แล้วตั้ง publicPath ให้โหลดผ่าน CDN

---

5 สรุป Workflow แนะนำ

1. วางโครงสร้างโปรเจกต์ให้รองรับการ split (feature/module separation)
2. ใช้ import() สำหรับโหลดโค้ดแบบ lazy
3. เปิด --splitting และ --minify ตอน build
4. จัดการ output naming/hash เพื่อให้ cache-friendly
5. ใช้ CDN เสิร์ฟ chunk และ asset

---

6 โปรแกรม (พื้นฐาน 3 + แนวประยุกต์ 3) สำหรับหัวข้อ **"การจัดการ bundle ขนาดใหญ่และ code splitting"** ใน **Bun.js** โดยจะมีทั้ง

- **โครงสร้างโปรเจกต์**
- **โค้ดเต็มไฟล์**
- **คำอธิบายโค้ดละเอียด**
- **ผลการรัน** (mock-up ตามลักษณะจริง)

Part 1 — โปรแกรมพื้นฐาน (3 ตัวอย่าง)

โปรแกรมพื้นฐานที่ 1 — Dynamic Import สำหรับลดขนาด Bundle

โครงสร้าง

```
bun-code-splitting-basic1/
├── src/
│   ├── main.ts
│   └── math.ts
└── bunfig.toml
```

ไฟล์: src/math.ts

```
export function heavyCalculation(a: number, b: number): number {
  console.log("□ heavyCalculation module loaded!");
  return a ** b;
}
```

ไฟล์: src/main.ts

```
console.log("□ App Started");


document.getElementById("calcBtn")?.addEventListener("click", async () => {
  const { heavyCalculation } = await import("./math.js");
  const result = heavyCalculation(2, 10);
  console.log("Result:", result);
});
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
splitting = true
minify = true
```

คำอธิบาย

- ใช้ import() แบบ Dynamic เพื่อโหลดโมดูลเฉพาะเมื่อกดปุ่ม ⟶ ลดขนาด bundle เริ่มต้น
- Bun จะสร้างไฟล์แยก เช่น main.js และ math-xxxxx.js
- ตัวนี้เหมาะสำหรับฟีเจอร์ที่ไม่ต้องโหลดทันที

ผลการรัน (จำลอง)

□ App Started

(เมื่อกดปุ่ม)

 heavyCalculation module loaded!

Result: 1024

โปรแกรมพื้นฐานที่ 2 — แบ่ง Chunk ตามฟีเจอร์

โครงสร้าง

```
bun-code-splitting-basic2/
├── src/
│   ├── main.ts
│   ├── chart.ts
│   └── report.ts
└── bunfig.toml
```

ไฟล์: src/chart.ts

```
export function renderChart() {
  console.log(" Chart module loaded!");
}
```

ไฟล์: src/report.ts

```
export function generateReport() {
  console.log(" Report module loaded!");
}
```

ไฟล์: src/main.ts

```
console.log(" App Started");

document.getElementById("chartBtn")?.addEventListener("click", async () => {
  const { renderChart } = await import("./chart.js");
  renderChart();
});

document.getElementById("reportBtn")?.addEventListener("click", async () => {
  const { generateReport } = await import("./report.js");
  generateReport();
});
```

ไฟล์: bunfig.toml

```
[build]
```

```
entrypoints = ["src/main.ts"]
outdir = "dist"
splitting = true
```

คำอธิบาย

- แบ่ง chunk เป็น chart และ report
- โหลดตามการกระทำของผู้ใช้ ⟶ ลด initial bundle size
- เหมาะสำหรับระบบที่มีหลายฟีเจอร์แต่ผู้ใช้ใช้แค่บางส่วน

ผลการรัน

 App Started

(เมื่อกด Chart)

 Chart module loaded!

(เมื่อกด Report)

 Report module loaded!

โปรแกรมพื้นฐานที่ 3 — Code Splitting + Lazy Rendering

โครงสร้าง

```
bun-code-splitting-basic3/
├── src/
│   ├── main.ts
│   └── gallery.ts
└── bunfig.toml
```

ไฟล์: src/gallery.ts

```
export function loadGallery() {
  console.log(" Gallery module loaded!");
}
```

ไฟล์: src/main.ts

```
console.log(" App Started");

let galleryLoaded = false;

document.getElementById("galleryBtn")?.addEventListener("click", async () => {
  if (!galleryLoaded) {
    const { loadGallery } = await import("./gallery.js");
    loadGallery();
```

```
    galleryLoaded = true;
  }
});
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
splitting = true
```

คำอธิบาย

- โหลดโมดูล gallery เฉพาะครั้งแรกที่เรียก
- ป้องกันการโหลดซ้ำเพื่อลดการใช้หน่วยความจำ
- เหมาะกับ UI ที่โหลดข้อมูลครั้งเดียว

ผลการรัน

□ App Started
(กดครั้งแรก)
□ Gallery module loaded!
(กดซ้ำ)
(no extra load)

Part 2 — โปรแกรมแนวประยุกต์ (3 ตัวอย่าง)

แนวประยุกต์ที่ 1 — โหลดโมดูล AI เฉพาะตอนใช้งาน

โครงสร้าง

```
bun-code-splitting-adv1/
├── src/
│   ├── main.ts
│   └── ai.ts
└── bunfig.toml
```

ไฟล์: src/ai.ts

```
export async function runAI(input: string) {
  console.log("□ AI module loaded!");
  return `AI processed: ${input}`;
}
```

ไฟล์: src/main.ts

```
console.log("□ Dashboard Started");
```

```
document.getElementById("aiBtn")?.addEventListener("click", async () => {
  const { runAI } = await import("./ai.js");
  const result = await runAI("Hello World");
  console.log(result);
});
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
splitting = true
```

ผลการรัน

□ Dashboard Started

(เมื่อกด AI)

□ AI module loaded!

AI processed: Hello World

แนวประยุกต์ที่ 2 — โหลด Visualization Library เฉพาะตอนเปิดกราฟ

โครงสร้าง

```
bun-code-splitting-adv2/
├── src/
│   ├── main.ts
│   └── d3.ts
└── bunfig.toml
```

ไฟล์: src/d3.ts

```
import * as d3 from "d3";

export function drawGraph() {
  console.log("□ D3 Graph module loaded!");
}
```

ไฟล์: src/main.ts

```
console.log("□ Analytics App");

document.getElementById("graphBtn")?.addEventListener("click", async () => {
```

```
  const { drawGraph } = await import("./d3.js");
  drawGraph();
});
```

ผลการรัน

□ Analytics App

(เมื่อกด Graph)

□ D3 Graph module loaded!

แนวประยุกต์ที่ 3 — Progressive Loading ของหน้า eCommerce

โครงสร้าง

```
bun-code-splitting-adv3/
├── src/
│   ├── main.ts
│   ├── cart.ts
│   └── payment.ts
└── bunfig.toml
```

ไฟล์: src/cart.ts

```
export function loadCart() {
  console.log("□ Cart module loaded!");
}
```

ไฟล์: src/payment.ts

```
export function processPayment() {
  console.log("□ Payment module loaded!");
}
```

ไฟล์: src/main.ts

```
console.log("□ E-Commerce App Started");

document.getElementById("cartBtn")?.addEventListener("click", async () => {
  const { loadCart } = await import("./cart.js");
  loadCart();
});

document.getElementById("payBtn")?.addEventListener("click", async () => {
  const { processPayment } = await import("./payment.js");
```

```
  processPayment();
});
```

ผลการรัน

□ E-Commerce App Started

(กด Cart)

□ Cart module loaded!

(กด Payment)

□ Payment module loaded!

การ optimize tree shaking และ dead code elimination

1□ ความหมายและเป้าหมาย

Tree Shaking

- คือการ **ตัดโค้ดที่ไม่ได้ถูกใช้งานออก** จาก bundle
- อิงกับ **ESM (ES Modules)** เพราะ Bun สามารถวิเคราะห์ import/export graph ได้ชัดเจน
- ช่วยลดขนาด bundle และโหลดเร็วขึ้น

Dead Code Elimination (DCE)

- คือการ **ลบโค้ดที่ไม่สามารถรันได้จริง**
- เช่น เงื่อนไข if(false){ ... } หรือ function ที่ไม่มีใครเรียก
- เพิ่มประสิทธิภาพทั้ง size และ runtime

เป้าหมายรวม

- ลดขนาดไฟล์ bundle
- ลด initial load time
- ป้องกัน code ที่ไม่จำเป็นถูกโหลดเข้าหน้าเว็บ

2□ หลักการทำงานของ Bun

1. Bun ใช้ **ESM dependency graph** ในการวิเคราะห์
2. ระบุว่าฟังก์ชัน/ตัวแปร/คลาสไหนถูกใช้งาน
3. ลบโค้ดส่วนที่ไม่มีการใช้งานออก
4. Minify ร่วมกับ DCE ⟶ ทำให้ bundle เล็กและเร็วขึ้น

ข้อสังเกต

- Bun tree shaking **ใช้ได้ดีที่สุดกับ ESM**
- CommonJS (require) จะทำ tree shaking ยากกว่า

3 เทคนิคการ Optimize Tree Shaking

3.1 ใช้ ESM Import แบบเจาะจง

```
// ❌ ไม่ดีสำหรับ tree shaking
import * as utils from "./utils";

// ✅ ดีสำหรับ tree shaking
import { usedFunc } from "./utils";
usedFunc();
```

3.2 ตั้งค่า sideEffects: false ใน package.json

```
{
  "name": "my-lib",
  "sideEffects": false
}
```

- บอก bundler ว่า **ไฟล์ที่ import ไม่มีผลข้างเคียง** ⟶ Bun สามารถลบโค้ดที่ไม่ได้ใช้ได้

3.3 ใช้ Annotation /* @__PURE__ */ สำหรับ function ที่ไม่มี side effect

```
/* @__PURE__ */
function helper() {
  return 123;
}
```

- ช่วยให้ minifier + DCE ตัด function ที่ไม่ได้ถูกเรียก

3.4 ลบ console/debug ที่ไม่จำเป็น

```
// bun build --drop ["console.*"]
```

- Bun จะลบ console.log, console.debug อัตโนมัติใน bundle production

3.5 แยก module ขนาดใหญ่เป็น dynamic import

- Module ที่โหลด lazy จะ **ไม่ถูกนับใน initial tree** ⟶ ลด initial bundle

4 เทคนิค Dead Code Elimination

4.1 เงื่อนไขคงที่ (constant condition)

```
if (false) {
  console.log("This will be removed");
}
```

- Bun จะลบโค้ดสาขานี้ออกอัตโนมัติ

4.2 Environment Variables

- ใช้ Bun define process.env.NODE_ENV

```
if (process.env.NODE_ENV === "development") {
  console.log("Dev mode");
}
```

- process.env.NODE_ENV = "production" ⟶ Bun จะลบโค้ด branch ที่ไม่ใช้

4.3 Remove Unused Exports

```
export function used() {}
export function unused() {}
```

- ถ้า unused() ไม่มีใครเรียก ⟶ Bun tree shaking จะลบออก

---

5 ตั้งค่า Bun Build สำหรับ Optimization

ตัวอย่าง bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
minify = true
splitting = true
target = "browser"
sourcemap = "linked"
drop = ["console.*"]          # ลบ console/debug
```

คำอธิบาย

- minify = true ⟶ ลดขนาด code + identifiers
- splitting = true ⟶ lazy load + code splitting
- drop = ["console.*"] ⟶ ลบ debug logs
- target = "browser" ⟶ optimize สำหรับ browser
- sourcemap = "linked" ⟶ debug ได้แต่ไม่เสีย performance

---

6 ตัวอย่างโปรแกรมแบบเต็มไฟล์ (Tree Shaking + DCE)

โครงสร้างโปรเจกต์

```
bun-tree-dce/
├── src/
│   ├── main.ts
```

```
│   └── utils.ts
└── bunfig.toml
```

ไฟล์: src/utils.ts

```
export function usedFunc() {
  console.log(" usedFunc called");
}

export function unusedFunc() {
  console.log(" unusedFunc called");
}

/* @__PURE__ */
export function pureHelper() {
  return 123;
}
```

ไฟล์: src/main.ts

```
import { usedFunc } from "./utils.js";

console.log(" App Started");
usedFunc();

// Dead code
if (false) {
  console.log("This will never run");
}
```

ผลการรันหลัง build

 App Started

 usedFunc called

- Bun จะ **ลบ unusedFunc และ branch if(false)`** ออก
- ขนาด bundle ลดลงอย่างมาก

ตัวอย่าง 3 โปรแกรมพื้นฐาน + 3 โปรแกรมแนวประยุกต์ สำหรับ **Tree Shaking + Dead Code Elimination** พร้อม:

- โครงสร้างโปรเจกต์

- โค้ดเต็มไฟล์
- bunfig.toml
- ผลการรัน

6 โปรแกรม สำหรับหัวข้อ **“การ optimize Tree Shaking และ Dead Code Elimination (DCE)”** ใน **Bun.js** โดยแบ่งเป็น

- **3 โปรแกรมพื้นฐาน**
- **3 โปรแกรมแนวประยุกต์**

และแต่ละโปรแกรมจะมี

- **โครงสร้างโปรเจกต์**
- **ไฟล์โค้ดเต็ม**
- **คำอธิบายโค้ดเชิงลึก**
- **ผลการรันจำลอง**

Part 1 — โปรแกรมพื้นฐาน (3 ตัวอย่าง)

พื้นฐาน 1 — ตัด unused function อัตโนมัติ

โครงสร้าง

```
bun-tree-basic1/
├── src/
│   ├── main.ts
│   └── utils.ts
└── bunfig.toml
```

ไฟล์: src/utils.ts

```
export function usedFunc() {
  console.log("✅ usedFunc executed");
}

export function unusedFunc() {
  console.log("❌ unusedFunc executed");
}

/* @__PURE__ */
```

```
export function pureHelper() {
  return 42;
}
```

ไฟล์: src/main.ts

```
import { usedFunc } from "./utils.js";

console.log("□ App Started");
usedFunc();

// Dead code
if (false) {
  console.log("This will never run");
}
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
minify = true
splitting = true
drop = ["console.*"]
```

คำอธิบาย

- unusedFunc ถูกตัดออกโดย tree shaking
- pureHelper ถูกลบเนื่องจากไม่ได้เรียกและมี annotation @__PURE__
- if(false) ถูกลบด้วย DCE
- drop: ["console.*"] จะลบ debug log ที่ไม่จำเป็น

ผลการรัน

□ App Started

□ usedFunc executed

พื้นฐาน 2 — ตัดเงื่อนไขไม่จำเป็น (Dead Code)

โครงสร้าง

```
bun-tree-basic2/
├── src/
│   ├── main.ts
```

```
│   └── config.ts
└── bunfig.toml
```

ไฟล์: src/config.ts

```
export const isProd = true;
export const isDev = false;
```

ไฟล์: src/main.ts

```
import { isProd, isDev } from "./config.js";

if (isProd) {
  console.log("□ Production Mode");
}

if (isDev) {
  console.log("□ Development Mode");
}
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
minify = true
drop = ["console.*"]
```

คำอธิบาย

- Bun วิเคราะห์ constant condition (isDev = false) ⟶ ตัด branch if(isDev)
- เพิ่มประสิทธิภาพ load และลด bundle size
- เงื่อนไขที่เป็นจริง (isProd = true) จะถูกเก็บไว้

ผลการรัน

□ Production Mode

พื้นฐาน 3 — ลบโมดูลที่ไม่ได้ใช้งาน

โครงสร้าง

```
bun-tree-basic3/
├── src/
│   ├── main.ts
│   └── helpers.ts
```

└── bunfig.toml

ไฟล์: src/helpers.ts

```
export function logHello() {
  console.log("Hello");
}

export function logBye() {
  console.log("Bye");
}
```

ไฟล์: src/main.ts

```
import { logHello } from "./helpers.js";

logHello();
```

คำอธิบาย

- logBye ไม่ถูกเรียก ⟶ Bun tree shaking ตัดออก
- ลดขนาด bundle โดยไม่กระทบฟังก์ชันที่ใช้จริง

ผลการรัน

```
Hello
```

Part 2 — โปรแกรมแนวประยุกต์ (3 ตัวอย่าง)

แนวประยุกต์ 1 — Lazy Loading + Tree Shaking

โครงสร้าง

```
bun-tree-adv1/
├── src/
│   ├── main.ts
│   └── analytics.ts
└── bunfig.toml
```

ไฟล์: src/analytics.ts

```
export function trackPageView() {
  console.log("□ Pageview tracked");
}

export function trackEvent() {
```

```
  console.log("□ Event tracked");
}
```

ไฟล์: src/main.ts

```
console.log("□ App Started");

document.getElementById("trackBtn")?.addEventListener("click", async () => {
  const { trackPageView } = await import("./analytics.js");
  trackPageView();
});
```

คำอธิบาย

- trackEvent ถูก tree shaking ตัดออกเพราะไม่ได้เรียก
- trackPageView โหลดเฉพาะตอน click ⟶ ลด initial bundle
- เหมาะสำหรับระบบ Analytics/Monitoring

ผลการรัน

□ App Started
(เมื่อกด Track)
□ Pageview tracked

แนวประยุกต์ 2 — Module ขนาดใหญ่ + Dead Code Elimination

โครงสร้าง

```
bun-tree-adv2/
├── src/
│   ├── main.ts
│   └── payment.ts
└── bunfig.toml
```

ไฟล์: src/payment.ts

```
export function payCreditCard() {
  console.log("□ Credit Card Payment");
}

export function payPaypal() {
  console.log("□ PayPal Payment");
}
```

```
if (false) {
  console.log("This is dead code");
}
```

ไฟล์: src/main.ts

```
import { payCreditCard } from "./payment.js";

console.log("□ Checkout Started");
payCreditCard();
```

คำอธิบาย

- payPaypal และ if(false) ถูก DCE และ tree shaking ตัดออก
- ขนาด bundle เล็กลง ⟶ faster load

ผลการรัน

□ Checkout Started

□ Credit Card Payment

แนวประยุกต์ 3 — Feature Modules + Tree Shaking

โครงสร้าง

```
bun-tree-adv3/
├── src/
│   ├── main.ts
│   ├── featureA.ts
│   └── featureB.ts
└── bunfig.toml
```

ไฟล์: src/featureA.ts

```
export function featureA() {
  console.log("⚡ Feature A executed");
}

export function unusedFeatureA() {
  console.log("□ Unused Feature A");
}
```

ไฟล์: src/featureB.ts

```
export function featureB() {
```

```
  console.log("□ Feature B executed");
}
```

ไฟล์: src/main.ts

```
console.log("□ App Started");

document.getElementById("btnA")?.addEventListener("click", async () => {
  const { featureA } = await import("./featureA.js");
  featureA();
});
```

คำอธิบาย

- featureB และ unusedFeatureA ถูก tree shaking ตัดออก
- Module featureA โหลดเฉพาะเมื่อผู้ใช้ click ⟶ ลด initial bundle size
- เหมาะสำหรับแอปที่มีหลาย feature แต่ผู้ใช้ใช้น้อยครั้ง

ผลการรัน

□ App Started

(เมื่อกด Feature A)

⚡ Feature A executed

การตั้งค่า Build Pipeline ด้วย Bun

1□ ความหมายของ Build Pipeline

Build Pipeline คือกระบวนการอัตโนมัติสำหรับเตรียมโค้ดให้พร้อมสำหรับ deployment หรือ production โดยรวมขั้นตอน เช่น

1. **Transpile/Compile** – แปลง TypeScript ⟶ JavaScript หรือ JSX ⟶ JS
2. **Code Splitting / Tree Shaking / Minify** – ลดขนาด bundle
3. **Asset Handling** – จัดการ static files, images, CSS
4. **Environment Variables** – ตั้งค่า production/dev variables
5. **Output / Deployment** – สร้าง folder พร้อม deploy

เป้าหมาย

- ทำให้โค้ดพร้อมใช้งานเร็วและมีขนาดเล็ก
- ลด human error
- ทำงานซ้ำได้อัตโนมัติ

---

2 Bun.js กับ Build Pipeline

Bun เป็น **all-in-one runtime + bundler + task runner** ⟶ สามารถสร้าง build pipeline แบบง่าย แต่มีประสิทธิภาพสูง

ความสามารถหลัก

- bun build ⟶ bundle + tree shaking + minify
- bun run ⟶ run script
- bun watch ⟶ watch file changes + rebuild
- การจัดการ assets (copy static files)
- การกำหนด multiple entry points
- Environment variables ผ่าน process.env

---

3 ขั้นตอนสร้าง Build Pipeline ด้วย Bun

3.1 สร้างไฟล์ bunfig.toml

ตัวอย่าง bunfig.toml สำหรับโปรเจกต์ใหญ่

```
[build]
entrypoints = ["src/main.ts", "src/admin.ts"]
outdir = "dist"
splitting = true
minify = true
target = "browser"
sourcemap = "linked"
drop = ["console.*"]

[assets]
include = ["public/**/*"]
outdir = "dist/public"

[env]
NODE_ENV = "production"
API_URL = "https://api.example.com"
```

คำอธิบาย

- entrypoints ⟶ จุดเริ่มต้นของ build หลายหน้า / หลาย feature
- splitting ⟶ เปิด code splitting

- minify ⟶ ลดขนาด JS
- sourcemap ⟶ สำหรับ debugging
- drop ⟶ ลบ console/debug
- [assets] ⟶ copy static files เช่น images, CSS
- [env] ⟶ environment variables สำหรับ production

3.2 การจัดการ Assets

สมมติโปรเจกต์มี folder public/

```
project/
├── src/
│   └── main.ts
├── public/
│   ├── style.css
│   └── logo.png
└── bunfig.toml
```

- Bun จะ copy public/* ไปที่ dist/public/ อัตโนมัติ
- สามารถ import CSS/Images แบบ relative path ได้ เช่น

```
import logoUrl from "../public/logo.png";
```

3.3 การ Build และ Watch

คำสั่ง build

```
bun build
```

- Bun อ่าน bunfig.toml ⟶ ทำ bundle, code splitting, minify, tree shaking, copy assets

คำสั่ง watch

```
bun watch
```

- จะ rebuild อัตโนมัติเมื่อไฟล์ใน src/ เปลี่ยน ⟶ ดีสำหรับ dev

---

3.4 การกำหนด Environment Variables

ในโค้ดสามารถใช้ process.env ได้ทันที:

```
console.log("Environment:", process.env.NODE_ENV);
console.log("API URL:", process.env.API_URL);
```

- Bun จะ replace ค่าใน production build ⟶ dead code branch สามารถถูกตัดออก

3.5 Workflow ตัวอย่าง Build Pipeline

1. พัฒนาใน src/
2. เขียน static assets ใน public/
3. ตั้งค่า bunfig.toml (entrypoints, splitting, minify, assets, env)
4. รัน bun build ⟶ สร้าง bundle + copy assets ⟶ ไปที่ dist/
5. รัน bun watch ระหว่าง dev ⟶ rebuild อัตโนมัติ
6. Deploy folder dist/ ขึ้น server หรือ CDN

---

4 ตัวอย่างโปรเจกต์ Build Pipeline

โครงสร้างโปรเจกต์

```
bun-build-pipeline/
├── src/
│   ├── main.ts
│   └── utils.ts
├── public/
│   ├── style.css
│   └── logo.png
└── bunfig.toml
```

ไฟล์: src/utils.ts

```
export function greet(name: string) {
  console.log(`Hello, ${name}!`);
}

export function unusedFunc() {
  console.log("This function will be removed by tree shaking");
}
```

ไฟล์: src/main.ts

```
import { greet } from "./utils.js";
import logo from "../public/logo.png";

console.log(" App Started");
greet("Bun.js User");
console.log("Logo URL:", logo);
```

```
if (process.env.NODE_ENV === "development") {
  console.log("Dev mode enabled");
}
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
splitting = true
minify = true
sourcemap = "linked"
drop = ["console.*"]
target = "browser"

[assets]
include = ["public/**/*"]
outdir = "dist/public"

[env]
NODE_ENV = "production"
API_URL = "https://api.example.com"
```

ผลการรัน build

```
 App Started
Hello, Bun.js User
Logo URL: /public/logo.png
```

- unusedFunc ถูก tree shaking ตัดออก
- console.log ที่อยู่ใน branch development ถูก drop
- Assets ถูก copy ไป dist/public/

---

6 โปรแกรม สำหรับหัวข้อ **“การตั้งค่า Build Pipeline ด้วย Bun”** โดยแบ่งเป็น

- **3 โปรแกรมพื้นฐาน**
- **3 โปรแกรมแนวประยุกต์**

แต่ละโปรแกรมจะมี

- **โครงสร้างโปรเจกต์**

- ไฟล์โค้ดเต็ม
- คำอธิบายโค้ดเชิงลึก
- ผลการรันจำลอง

Part 1 — โปรแกรมพื้นฐาน (3 ตัวอย่าง)

พื้นฐาน 1 — Build Pipeline พื้นฐานกับ Bun

โครงสร้าง

```
bun-build-basic1/
├── src/
│   └── main.ts
└── bunfig.toml
```

ไฟล์: src/main.ts

```
console.log("□ Hello Bun Build Pipeline");

// ตัวอย่าง function สำหรับ tree shaking
export function unusedFunc() {
  console.log("□ This will be removed");
}

export function usedFunc() {
  console.log("□ This is used");
}

usedFunc();
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
splitting = true
minify = true
target = "browser"
drop = ["console.*"]
sourcemap = "linked"
```

คำอธิบาย

- entrypoints ⟶ จุดเริ่ม build
- splitting ⟶ เปิด code splitting
- minify ⟶ ลดขนาด JS
- drop ⟶ ลบ console/debug
- Tree shaking จะตัด unusedFunc ออก

ผลการรัน

□ Hello Bun Build Pipeline

□ This is used

พื้นฐาน 2 — Build Pipeline + Asset Copy

โครงสร้าง

```
bun-build-basic2/
├── src/
│   └── main.ts
├── public/
│   └── logo.png
└── bunfig.toml
```

ไฟล์: src/main.ts

```
import logo from "../public/logo.png";

console.log("□ App Started");
console.log("Logo URL:", logo);
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
splitting = true
minify = true

[assets]
include = ["public/**/*"]
outdir = "dist/public"
```

คำอธิบาย

- Bun จะ **copy assets** จาก public/ ⟶ dist/public/
- Tree shaking + minify ยังทำงานตามปกติ

ผลการรัน

□ App Started

Logo URL: /public/logo.png

พื้นฐาน 3 — Build Pipeline + Environment Variables

โครงสร้าง

bun-build-basic3/

```
├── src/
│   └── main.ts
└── bunfig.toml
```

ไฟล์: src/main.ts

```
console.log("□ App Started");
console.log("Environment:", process.env.NODE_ENV);
console.log("API URL:", process.env.API_URL);
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts"]
outdir = "dist"
minify = true

[env]
NODE_ENV = "production"
API_URL = "https://api.example.com"
```

คำอธิบาย

- Bun จะ replace environment variables ใน build
- Dead code branch สามารถถูกตัดออก

ผลการรัน

□ App Started

Environment: production

API URL: https://api.example.com

Part 2 — โปรแกรมแนวประยุกต์ (3 ตัวอย่าง)

แนวประยุกต์ 1 — Multi-entry Build + Code Splitting

โครงสร้าง

```
bun-build-adv1/
├── src/
│   ├── main.ts
│   └── admin.ts
└── bunfig.toml
```

ไฟล์: src/main.ts

```
console.log("□ Client App Started");

export function clientFeature() {
  console.log("□ Client feature loaded");
}
clientFeature();
```

ไฟล์: src/admin.ts

```
console.log("□ Admin App Started");

export function adminFeature() {
  console.log("□ Admin feature loaded");
}
adminFeature();
```

ไฟล์: bunfig.toml

```
[build]
entrypoints = ["src/main.ts", "src/admin.ts"]
outdir = "dist"
splitting = true
minify = true
target = "browser"
```

คำอธิบาย

- มีหลาย entrypoints ⟶ Bun สร้างหลาย bundle
- ลดขนาดแต่ละ bundle ⟶ โหลดเร็วขึ้น
- Code splitting ทำงานตาม entrypoint