svelte

SVELTE WEB PROGRAMMING:
INTERMEDIATE
(Integrative-Generative AI Edition)

Student Price Book Center

คำนำ

ปัจจุบันการพัฒนาเว็บแอปพลิเคชันได้ก้าวไปไกลกว่าการจัดวาง HTML และเขียน JavaScript เพื่อโต้ตอบกับผู้ใช้เท่านั้น เทคโนโลยีเฟรมเวิร์กสมัยใหม่ได้เข้ามามีบทบาทสำคัญในการเพิ่มประสิทธิภาพของแอปพลิเคชันและยกระดับประสบการณ์ของผู้ใช้งาน หนึ่งในเฟรมเวิร์กที่ได้รับความนิยมและได้รับการจับตามองอย่างต่อเนื่องในช่วงไม่กี่ปีที่ผ่านมาคือ **Svelte** — เฟรมเวิร์กที่เน้นความเรียบง่าย ประสิทธิภาพ และแนวทางที่แตกต่างจากเฟรมเวิร์กแบบเดิม

จากความสำเร็จของหนังสือ *Svelte Web Programming: Beginner* ซึ่งได้วางรากฐานให้กับผู้อ่านในด้านแนวคิดพื้นฐานของ Svelte และการสร้างเว็บแอปพลิเคชันอย่างเป็นระบบ หนังสือ *Svelte Web Programming: Intermediate* เล่มนี้จึงได้ถูกเขียนขึ้นเพื่อสานต่อความรู้ไปอีกขั้น โดยมุ่งเน้นในระดับ “กลางถึงขั้นสูง” ที่จำเป็นสำหรับการพัฒนาเว็บแอปพลิเคชันที่มีความซับซ้อนมากขึ้น พร้อมรองรับการใช้งานในโลกจริงอย่างมีประสิทธิภาพ

ภายในหนังสือเล่มนี้ ผู้อ่านจะได้พบกับหัวข้อที่ลึกขึ้นเกี่ยวกับ **กลไก Reactivity** ซึ่งเป็นหัวใจสำคัญของ Svelte ไม่ว่าจะเป็นการใช้ $: เพื่อสร้าง reactive statements, การทำ derived reactivity, และการใช้ reactive blocks เพื่อสร้างตรรกะที่ปรับตัวตามข้อมูลอย่างชาญฉลาด นอกจากนี้ยังครอบคลุมการจัดการ **Event และ Lifecycle** ของ component ในหลากหลายบริบท เช่น การใช้งาน event handler, การเชื่อมโยงกับ API ผ่าน lifecycle functions รวมถึงการสร้าง custom event ที่ใช้สื่อสารระหว่าง components

อีกหนึ่งจุดเด่นของหนังสือคือการนำเสนอเรื่อง **Component ที่ซ้อนกัน**, การใช้ **Slot**, การส่ง **Props**, และเทคนิคการอ้างอิง DOM ผ่าน bind:this อย่างถูกวิธี ซึ่งเป็นประเด็นที่มักสร้างความสับสนให้กับผู้เริ่มต้นใช้งาน นอกจากนี้ยังอธิบายเรื่อง **Store และ Global State** ซึ่งมีความสำคัญต่อการพัฒนาแอปพลิเคชันหลายหน้า โดยจะกล่าวถึงการใช้ writable, readable, derived store ตลอดจนการสร้าง custom store เพื่อจัดการ state ร่วมในลักษณะต่าง ๆ

ท้ายที่สุด หนังสือยังเสนอวิธีการ **เชื่อมต่อกับ External APIs** ซึ่งถือเป็นองค์ประกอบสำคัญของแอปพลิเคชันสมัยใหม่ ผู้อ่านจะได้เรียนรู้การใช้ fetch(), การโหลดข้อมูลผ่าน onMount(), การจัดการสถานะ loading/error/empty และตัวอย่างแนวประยุกต์ที่สามารถนำไปปรับใช้กับระบบจริงได้อย่างมั่นใจ

หนังสือเล่มนี้จึงเหมาะสำหรับผู้ที่มีพื้นฐานการใช้งาน Svelte อยู่แล้ว และต้องการขยับขยายทักษะไปสู่การพัฒนาแอปพลิเคชันที่มีความยืดหยุ่นและสามารถดูแลรักษาในระยะยาว ไม่ว่าจะเป็นนักพัฒนารายบุคคล ทีมพัฒนาในองค์กร หรือผู้สอนที่ต้องการทรัพยากรการเรียนรู้ที่ครบถ้วน

ผู้เขียนขอขอบคุณผู้อ่านทุกท่านที่ร่วมเดินทางในการเรียนรู้ Svelte อย่างจริงจัง และหวังเป็นอย่างยิ่งว่า *Svelte Web Programming: Intermediate* เล่มนี้ จะเป็นคู่มือสำคัญที่ช่วยต่อยอดศักยภาพของท่านในการสร้างสรรค์เว็บแอปพลิเคชันที่ทันสมัย มีประสิทธิภาพ และรองรับการพัฒนาในระดับ production ได้อย่างเต็มที่

ขอให้ทุกท่านสนุกกับการเรียนรู้ และพัฒนาโปรเจกต์ที่ยอดเยี่ยมด้วย Svelte ต่อไป

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 6

Reactivity และ $: statement

(Reactivity and $: statement)

เนื้อหา

บทที่ 6: Reactivity และ $: statement

Svelte เป็นเฟรมเวิร์กที่มีจุดเด่นในเรื่องของ **Reactivity** หรือการตอบสนองต่อการเปลี่ยนแปลงของข้อมูลแบบอัตโนมัติ ซึ่งช่วยให้การพัฒนาเว็บแอปพลิเคชันมีความกระชับและเป็นธรรมชาติมากขึ้นกว่าหลายเฟรมเวิร์กที่ต้องอาศัยแนวคิด Virtual DOM การจัดการ Reactivity ใน Svelte มีความแตกต่างอย่างสิ้นเชิง เพราะถูกออกแบบให้คอมไพล์ในขั้นตอน build time ส่งผลให้การอัปเดต UI มีประสิทธิภาพสูงและมีโครงสร้างโค้ดที่เข้าใจง่ายกว่า ในบทนี้ เราจะสำรวจแนวคิดและการใช้งาน **Reactivity** โดยเฉพาะการใช้ **$: statement** และเทคนิคที่เกี่ยวข้อง เช่น **reactive block** และ **derived values** เพื่อสร้างการตอบสนองที่เป็นธรรมชาติภายในคอมโพเนนต์

หัวใจของ Reactivity ใน Svelte เริ่มต้นจากการใช้ตัวแปร **let** ซึ่งมีบทบาทสำคัญในการกำหนดค่าที่สามารถเปลี่ยนแปลงได้ภายในคอมโพเนนต์ เมื่อค่าของตัวแปรที่ประกาศด้วย let ถูกเปลี่ยนแปลง Svelte จะตรวจจับและอัปเดต UI ที่เกี่ยวข้องโดยอัตโนมัติ ลักษณะการทำงานนี้ทำให้ไม่จำเป็นต้องใช้ฟังก์ชัน setState หรือการผูก event แบบซับซ้อน ตัวอย่างเช่น การเปลี่ยนค่าของตัวแปร count ที่ประกาศด้วย let จะทำให้ UI อัปเดตโดยทันที ทำให้การโต้ตอบของผู้ใช้กับคอมโพเนนต์เป็นไปอย่างราบรื่น

อย่างไรก็ตาม การเปลี่ยนแปลงตัวแปรบางครั้งต้องมีการคำนวณค่าที่ตามมาใหม่ การใช้ **$: statement** จึงเป็นอีกฟีเจอร์สำคัญของ Svelte ซึ่งทำหน้าที่เป็น **reactive declaration** เมื่อใดที่ตัวแปรที่อยู่ใน $: statement เปลี่ยนค่า โค้ดที่อยู่ใน $: ... จะถูกประมวลผลซ้ำโดยอัตโนมัติ นี่คือวิธีการที่สะดวกในการจัดการค่าที่ขึ้นกับค่าของตัวแปรอื่นโดยไม่ต้องเขียนฟังก์ชันหรือ watcher เพิ่มเติม

นอกจาก $: statement แล้ว Svelte ยังมีโครงสร้างที่เรียกว่า **reactive block** ซึ่งใช้รูปแบบ {$: { ... }} เพื่อตอบสนองต่อการเปลี่ยนแปลงหลายตัวแปรภายในบล็อกเดียวกัน Reactive block ทำให้สามารถรวมตรรกะที่ซับซ้อนในการคำนวณหรือดำเนินการหลายขั้นตอนเมื่อมีการเปลี่ยนแปลงค่าของตัวแปรที่เกี่ยวข้อง ตัวอย่างเช่น การอัปเดตข้อมูลจาก API หรือการคำนวณค่าที่ซับซ้อนหลายขั้นตอนภายใน reactive block จะช่วยให้โค้ดอ่านง่ายและดูเป็นลำดับขั้นตอนตามธรรมชาติ

อีกแนวคิดหนึ่งที่สำคัญคือ **derived values** ซึ่งหมายถึงค่าที่คำนวณขึ้นมาจากตัวแปรอื่น ๆ แบบอัตโนมัติ โดยไม่จำเป็นต้องกำหนดค่าใหม่ด้วยตนเอง Derived values ช่วยลดความซ้ำซ้อนของโค้ดและทำให้แน่ใจว่าค่าที่ได้จะถูกอัปเดตตามการเปลี่ยนแปลงของตัวแปรต้นทางเสมอ ตัวอย่างเช่น ถ้าเรามีตัวแปร firstName และ lastName เราสามารถสร้าง derived value fullName เพื่อรวมชื่อเต็มได้แบบอัตโนมัติเมื่อมีการเปลี่ยนชื่อใดชื่อหนึ่ง

เมื่อรวมแนวคิดทั้งหมดนี้เข้าด้วยกัน เราจะเห็นได้ว่า Reactivity ใน Svelte มีความยืดหยุ่นและทรงพลังอย่างมาก การใช้ **let**, **$: statement**, **reactive block**, และ **derived values** ทำให้การสร้างคอมโพเนนต์แบบ interactive เป็นเรื่องง่ายและชัดเจน ทั้งยังช่วยลดภาระในการจัดการสถานะและลดบั๊กที่อาจเกิดจากการอัปเดต UI ด้วยตนเอง ซึ่งเป็นข้อได้เปรียบที่สำคัญเมื่อเทียบกับเฟรมเวิร์กอื่น ๆ

ในบทถัดไป เราจะลงลึกถึงการประยุกต์ใช้ Reactivity และ **$: statement** ผ่านตัวอย่างโค้ดจริง เพื่อให้เห็นภาพการทำงานของมันอย่างเป็นรูปธรรม คุณจะได้เรียนรู้วิธีสร้างคอมโพเนนต์ที่ตอบสนองอย่างมีประสิทธิภาพ การสร้างตรรกะที่ซับซ้อนใน reactive block และการออกแบบ derived values ให้มีความกระชับและอ่านง่าย ทั้งหมดนี้จะช่วยเพิ่มศักยภาพในการพัฒนาเว็บแอปพลิเคชันด้วย Svelte ให้มีคุณภาพสูงและใช้งานได้อย่างราบรื่นที่สุด

Reactivity และ $: statement

---

1. Reactivity ของตัวแปร (let)

- Svelte ใช้ตัวแปรแบบ let เพื่อเก็บสถานะ (state) ภายใน component
- เมื่อเปลี่ยนค่า let ตัวแปร Svelte จะ **ติดตาม (track)** และอัปเดต DOM หรือค่าที่เกี่ยวข้องให้อัตโนมัติ
- ไม่จำเป็นต้องใช้ setState หรือฟังก์ชันพิเศษเหมือน React
- ตัวแปรที่ประกาศด้วย let จะทำงานแบบ reactive โดยตรง เมื่อค่ามันเปลี่ยน ตัว component จะ re-render เฉพาะส่วนที่ใช้ค่าตัวแปรนั้น

---

2. การใช้ $: เพื่อ re-compute

- $: เป็น syntax พิเศษใน Svelte สำหรับ **reactive statement**
- ใช้เขียนบรรทัดคำสั่งที่ต้องการให้ re-run ทุกครั้งที่ค่าที่เกี่ยวข้องเปลี่ยน
- $: ทำหน้าที่เหมือน “watcher” หรือ “computed property” ใน Vue

- Syntax:

$: derivedValue = someExpression;

- ทุกครั้งที่ตัวแปรใน someExpression เปลี่ยน ค่า derivedValue จะถูกคำนวณใหม่อัตโนมัติ

---

3. การใช้ reactive block

- สามารถเขียน block ของคำสั่งที่ต้องการให้ re-run ได้ โดยใช้ $: ตามด้วยบล็อกคำสั่ง

```
$: {
  // คำสั่งที่ต้องการให้ re-run เมื่อค่าที่เกี่ยวข้องเปลี่ยน
  console.log(`Count is ${count}`);
}
```

- เหมาะสำหรับ side effect หรือ logging ที่ต้องทำซ้ำเมื่อค่าเปลี่ยน

---

4. การทำ derived values

- Derived values คือค่าที่คำนวณจากตัวแปรอื่น ๆ แบบ reactive
- ใช้ $: เพื่อประกาศค่าใหม่ที่อัปเดตอัตโนมัติเมื่อค่าต้นทางเปลี่ยน

ตัวอย่าง:

```
<script>
  let price = 100;
  let quantity = 2;

  // derived value: total price คำนวณจาก price และ quantity
  $: total = price * quantity;
</script>

<p>Price: {price}</p>
<p>Quantity: {quantity}</p>
<p>Total: {total}</p>
```

เมื่อเปลี่ยน price หรือ quantity ค่าของ total จะอัปเดตอัตโนมัติ

Reactivity และ $: statement — รายละเอียดเชิงลึก

---

1. Reactivity ของตัวแปร (let)

- Svelte ทำงานโดยการ **คอมไพล์โค้ดล่วงหน้า (compile-time)** เพื่อสร้างโค้ดที่คอยติดตามและอัปเดต DOM เมื่อข้อมูลเปลี่ยน

- ตัวแปรที่ประกาศด้วย let ภายใน <script> ของ Svelte component จะเป็นตัวแปร reactive โดยอัตโนมัติ
- เมื่อเราสั่งเปลี่ยนค่า let เช่น count = 1 Svelte จะตรวจจับและรีเฟรช (re-render) เฉพาะส่วนของ DOM ที่ใช้ตัวแปรนั้น
- ต่างจาก React ที่ต้องเรียกใช้ setState() เพื่อบอกให้ re-render
- ใน Svelte เพียงแค่อัปเดตค่า let ก็เพียงพอ

ตัวอย่าง:

```
<script>
  let count = 0;

  function increment() {
    count += 1; // แค่เปลี่ยนค่า Svelte จะ update UI อัตโนมัติ
  }
</script>

<button on:click={increment}>
  Clicked {count} {count === 1 ? 'time' : 'times'}
</button>
```

- เมื่อคลิกปุ่ม count เปลี่ยน ค่าใน DOM ก็เปลี่ยนตามทันที

---

2. การใช้ $: เพื่อ re-compute

- $: เป็นเครื่องหมายพิเศษที่ใช้ประกาศ **reactive declaration หรือ reactive statement**
- คำสั่งนี้จะถูกรันใหม่ทุกครั้งที่ค่าที่เกี่ยวข้องเปลี่ยน
- มักใช้สำหรับ **สร้างค่าใหม่ (derived value)** หรือ **เรียกฟังก์ชันที่ต้องทำงานใหม่ตามตัวแปร**
- ช่วยให้หลีกเลี่ยงการเขียนโค้ดซ้ำ ๆ ในฟังก์ชันและช่วยเพิ่มประสิทธิภาพ

ตัวอย่าง:

```
<script>
  let price = 20;
  let quantity = 3;

  // total จะคำนวณใหม่อัตโนมัติเมื่อ price หรือ quantity เปลี่ยน
  $: total = price * quantity;
</script>
```

```
<p>Price: {price}</p>
<p>Quantity: {quantity}</p>
<p>Total: {total}</p>
```

- เมื่อแก้ price หรือ quantity ตัวแปร total จะถูกคำนวณใหม่ทันที

---

3. การใช้ reactive block

- นอกจาก reactive declaration ที่เป็นการกำหนดค่าแล้ว เราสามารถใช้ **reactive block** เพื่อรันชุดคำสั่งหลายบรรทัดได้
- ประโยชน์คือสามารถทำ side effect เช่น log, call API, หรืออัปเดตค่าตัวแปรอื่น ๆ
- reactive block จะรันเมื่อใดก็ตามที่ค่าที่ใช้ในบล็อกนั้นเปลี่ยน

ตัวอย่าง:

```
<script>
  let count = 0;

  $: {
    console.log(`Count updated to ${count}`);
  }
</script>

<button on:click={() => count++}>
  Clicked {count} times
</button>
```

- ทุกครั้งที่ count เปลี่ยน จะมีการพิมพ์ log ลง console อัตโนมัติ

---

4. การทำ derived values

- Derived values คือค่าที่ **คำนวณได้จากตัวแปรอื่นโดยอัตโนมัติ**
- โดยใช้ $: ร่วมกับการคำนวณภายใน reactive declaration
- ทำให้ไม่ต้องคอยเขียนโค้ดซ้ำหรือสร้าง state เพิ่มเติม

ตัวอย่างการใช้งานจริง:

```
<script>
  let firstName = "John";
  let lastName = "Doe";
```

```
  // ชื่อเต็ม derived จาก firstName และ lastName
  $: fullName = `${firstName} ${lastName}`;
</script>

<input bind:value={firstName} placeholder="First Name" />
<input bind:value={lastName} placeholder="Last Name" />

<p>Your full name is: {fullName}</p>
```

- เมื่อเปลี่ยนชื่อหรือนามสกุล ข้อความชื่อเต็มจะอัปเดตทันที

สรุปจุดสำคัญ

หัวข้อ	รายละเอียดสำคัญ
Reactive let	เปลี่ยนค่า let ภายใน component จะ trigger update UI โดยอัตโนมัติ
$: statement	ใช้เขียนการคำนวณหรือฟังก์ชันที่ต้องรันซ้ำเมื่อค่าที่เกี่ยวข้องเปลี่ยน
Reactive block	บล็อกคำสั่งหลายบรรทัดที่รันใหม่เมื่อค่าในบล็อกเปลี่ยน เหมาะกับ side effects
Derived values	คำนวณค่าที่ขึ้นกับตัวแปรอื่น ๆ โดยไม่ต้องเขียนโค้ดเพิ่มหรือเก็บ state ซ้ำ

Reactivity ของตัวแปร (let) ใน Svelte — เชิงลึก

---

1. ตัวแปร let คือ State ของ Component

- ใน Svelte ตัวแปรที่ประกาศด้วย let ภายใน <script> ของ component คือ **state** หรือข้อมูลที่ component ใช้เก็บค่าและสถานะ
- สามารถเปลี่ยนค่าได้ตลอดเวลา โดยใช้ assignment ปกติ เช่น count = 1;

---

2. การเปลี่ยนค่า let ตัวแปรจะทำให้ Component อัปเดต UI อัตโนมัติ

- เมื่อเราเขียนโค้ดเปลี่ยนค่า let เช่น count = count + 1
- Svelte จะ **ตรวจจับการเปลี่ยนแปลงนี้และรีเฟรช (re-render) เฉพาะส่วนของ DOM ที่ใช้ตัวแปรนี้** ให้เองโดยอัตโนมัติ
- ไม่ต้องเรียกฟังก์ชันพิเศษ หรือ setState เหมือน React
- ทำให้โค้ดสั้นและเขียนง่าย

---

3. ตัวอย่างง่าย ๆ

```
<script>
  let count = 0;

  function increment() {
    count = count + 1;
  }
</script>

<button on:click={increment}>
  You clicked {count} {count === 1 ? 'time' : 'times'}
</button>
```

- เมื่อคลิกปุ่ม ตัวแปร count ถูกเพิ่มขึ้นทีละ 1
- ข้อความในปุ่มจะอัปเดตตามค่า count แบบเรียลไทม์

---

4. วิธีที่ Svelte ทำงานกับ let

- เมื่อเราเขียน count = count + 1 หรือกำหนดค่าใหม่ให้ตัวแปร let
- Svelte คอมไพล์โค้ดให้มีการแจ้งเตือน (notify) เพื่อรีเฟรช UI เฉพาะส่วนที่เกี่ยวข้อง
- ช่วยเพิ่มประสิทธิภาพ ลดการ re-render ที่ไม่จำเป็น

---

5. ข้อควรระวัง

- ต้องเปลี่ยนค่าโดยตรง เช่น count = 10 หรือ count++ (ต้องเขียนแบบ count = count + 1 หรือ count += 1)
- ถ้าเปลี่ยนค่าโดยแก้ไขตัวแปรที่เป็นอ็อบเจ็กต์หรืออาร์เรย์ เช่น obj.prop = 1 แบบนี้ Svelte **ไม่รู้ว่าค่าเปลี่ยน** เพราะไม่ได้เปลี่ยน reference ของ obj
- ในกรณีนั้น ต้องเขียนแบบสร้างอ็อบเจ็กต์ใหม่ เช่น

```
obj = {...obj, prop: 1};
```

เพื่อให้ Svelte รู้ว่ามีการเปลี่ยนแปลงและจะอัปเดต UI

---

6. สรุป

ข้อดีของ Reactive let ใน Svelte

เปลี่ยนแปลงค่าปกติด้วย = แล้ว UI อัปเดตทันที

ไม่ต้องเรียก setState หรือฟังก์ชันพิเศษ

ข้อดีของ Reactive let ใน Svelte

มีประสิทธิภาพเพราะ re-render เฉพาะส่วนที่ใช้ตัวแปร

ง่ายต่อการเขียนและอ่านโค้ด

นี่คือตัวอย่างโปรแกรม Svelte แบบเต็มไฟล์ รวมทั้งโครงสร้างโปรเจกต์ คำอธิบายโค้ด และผลการรัน จำนวน 3 โปรแกรมพื้นฐาน และ 3 โปรแกรมแนวประยุกต์ ที่เน้นหัวข้อ **Reactivity ของตัวแปร (let)**

โครงสร้างโปรเจกต์ (เหมือนกันทุกตัวอย่าง)

```
my-svelte-app/
└── src/
    ├── BasicCounter.svelte
    ├── ToggleSwitch.svelte
    ├── TextInput.svelte
    ├── ShoppingCart.svelte
    ├── DarkModeToggle.svelte
    └── Timer.svelte
```

โปรแกรมพื้นฐาน 3 ตัวอย่าง

---

1. BasicCounter.svelte

```
<script>
  let count = 0;

  function increment() {
    count += 1;
  }
</script>

<button on:click={increment}>
  Clicked {count} {count === 1 ? 'time' : 'times'}
</button>
```

คำอธิบายโค้ด:

- ประกาศตัวแปร count ด้วย let

- ฟังก์ชัน increment เพิ่มค่า count ทีละ 1
- ปุ่มแสดงจำนวนครั้งที่คลิกแบบเรียลไทม์ด้วย reactive state

ผลการรัน:

- กดปุ่มครั้งแรก จะแสดง "Clicked 1 time"
- กดเพิ่มขึ้น จะแสดง "Clicked 2 times", "Clicked 3 times" เป็นต้น

---

2. ToggleSwitch.svelte

```
<script>
  let isOn = false;

  function toggle() {
    isOn = !isOn;
  }
</script>

<button on:click={toggle}>
  {isOn ? "ON" : "OFF"}
</button>
```

คำอธิบายโค้ด:

- ตัวแปร Boolean isOn เก็บสถานะเปิด/ปิด
- ฟังก์ชัน toggle สลับสถานะ
- ปุ่มแสดงข้อความ ON หรือ OFF ตามสถานะแบบ reactive

ผลการรัน:

- ปุ่มแสดง "OFF" เริ่มต้น
- คลิกปุ่มเปลี่ยนเป็น "ON" และสลับกลับได้

---

3. TextInput.svelte

```
<script>
  let name = "";
</script>

<input placeholder="Type your name" bind:value={name} />
<p>Hello, {name || "stranger"}!</p>
```

คำอธิบายโค้ด:

- ตัวแปร name เก็บข้อความจาก input
- ใช้ two-way binding กับ <input>
- ข้อความแสดงตามค่าตัวแปรแบบ reactive

ผลการรัน:

- เมื่อพิมพ์ชื่อ จะเห็นข้อความ "Hello, [name]!" อัปเดตทันที
- ถ้าไม่พิมพ์แสดง "Hello, stranger!"

โปรแกรมแนวประยุกต์ 3 ตัวอย่าง

---

4. ShoppingCart.svelte

```
<script>
  let items = [
    { id: 1, name: "Apple", price: 50, quantity: 1 },
    { id: 2, name: "Banana", price: 20, quantity: 2 },
  ];

  // ฟังก์ชันเพิ่มจำนวนสินค้า
  function incrementQuantity(id) {
    const item = items.find(i => i.id === id);
    item.quantity += 1;
    items = [...items]; // รีแอสซายน์เพื่อแจ้ง Svelte ว่าข้อมูลเปลี่ยน
  }

  // ฟังก์ชันลดจำนวนสินค้า
  function decrementQuantity(id) {
    const item = items.find(i => i.id === id);
    if (item.quantity > 0) {
      item.quantity -= 1;
      items = [...items];
    }
  }

  // คำนวณราคารวม (derived)
  $: total = items.reduce((sum, item) => sum + item.price * item.quantity, 0);
```

```
</script>

<h2>Shopping Cart</h2>

<ul>
  {#each items as item (item.id)}
    <li>
      {item.name} - ฿{item.price} x {item.quantity}
      <button on:click={() => decrementQuantity(item.id)}>-</button>
      <button on:click={() => incrementQuantity(item.id)}>+</button>
    </li>
  {/each}
</ul>

<p><strong>Total: ฿{total}</strong></p>
```

คำอธิบายโค้ด:

- ใช้ let items เก็บรายการสินค้า
- ฟังก์ชันเพิ่ม/ลดจำนวนสินค้าปรับค่าภายในอาร์เรย์แล้วรีแอสซายน์ใหม่ เพื่อให้ Svelte รู้ว่าข้อมูลเปลี่ยน
- ใช้ reactive declaration $: คำนวณ total อัตโนมัติ

ผลการรัน:

- กดปุ่ม + หรือ - เพื่อปรับจำนวนสินค้า
- ราคารวมอัปเดตตามจำนวนสินค้าแบบเรียลไทม์

---

5. DarkModeToggle.svelte

```
<script>
  let darkMode = false;

  function toggleDarkMode() {
    darkMode = !darkMode;
    if (darkMode) {
      document.body.style.backgroundColor = "#222";
      document.body.style.color = "#eee";
    } else {
```

```
      document.body.style.backgroundColor = "";
      document.body.style.color = "";
    }
  }
</script>

<button on:click={toggleDarkMode}>
  {darkMode ? "Switch to Light Mode" : "Switch to Dark Mode"}
</button>
```

คำอธิบายโค้ด:

- ใช้ตัวแปร Boolean darkMode เก็บสถานะธีม
- เมื่อกดปุ่ม เปลี่ยนค่า darkMode และเปลี่ยนสีพื้นหลังและข้อความของ <body> โดยตรง
- ข้อความปุ่มอัปเดตตามสถานะ

ผลการรัน:

- เริ่มต้นเป็น Light Mode
- กดปุ่มเปลี่ยนเป็น Dark Mode และกลับมาได้

---

6. Timer.svelte

```
<script>
  let time = 0;
  let interval;

  function startTimer() {
    clearInterval(interval);
    interval = setInterval(() => {
      time += 1; // เพิ่มเวลา ทุก 1 วินาที
    }, 1000);
  }

  function stopTimer() {
    clearInterval(interval);
  }

  function resetTimer() {
```

```
    time = 0;
    clearInterval(interval);
  }
</script>

<p>Time elapsed: {time} seconds</p>
<button on:click={startTimer}>Start</button>
<button on:click={stopTimer}>Stop</button>
<button on:click={resetTimer}>Reset</button>
```

คำอธิบายโค้ด:

- ตัวแปร time เก็บเวลาที่ผ่านไป
- ใช้ setInterval เพื่อเพิ่มค่า time ทุกวินาที
- ปุ่ม Start, Stop, Reset ควบคุม timer
- การเปลี่ยนค่า time จะอัปเดต UI อัตโนมัติ

ผลการรัน:

- กด Start แล้วเวลานับเพิ่มทีละวินาที
- กด Stop หยุดเวลา
- กด Reset ตั้งค่าเวลาเป็น 0

การใช้ $: เพื่อ re-compute ใน Svelte — เชิงลึก

---

1. $: คืออะไร?

- $: (dollar-colon) เป็น **reactive statement** หรือ **reactive declaration** ใน Svelte
- ใช้สำหรับประกาศว่าบรรทัดหรือบล็อกคำสั่งนี้ควรถูก **รันใหม่ (re-run)** ทุกครั้งที่ค่าที่เกี่ยวข้องถูกเปลี่ยนแปลง
- คล้ายกับ **computed properties** ใน Vue หรือ **useMemo** ใน React แต่เรียบง่ายและตรงไปตรงมา

---

2. รูปแบบการใช้งาน $:

แบบประกาศค่าใหม่ (Reactive declaration)

```
$: total = price * quantity;
```

- ทุกครั้งที่ price หรือ quantity เปลี่ยน ค่า total จะถูกคำนวณใหม่อัตโนมัติ

แบบ reactive block (หลายบรรทัด)

```
$: {
  console.log(`Count is now ${count}`);
  doSomething(count);
}
```

- บล็อกนี้จะถูกรันใหม่ทุกครั้งที่ค่าที่ใช้ภายในบล็อก เช่น count เปลี่ยน

---

3. การใช้งานหลัก

- **คำนวณค่าใหม่แบบอัตโนมัติ (Derived Values)**
- **ทำงาน side effect เมื่อข้อมูลเปลี่ยน (เช่น log, เรียก API)**
- **หลีกเลี่ยงการเขียนโค้ดซ้ำในฟังก์ชัน**

---

4. ข้อควรระวัง

- อย่าเขียน reactive statement ที่ทำให้เกิด **loop ไม่สิ้นสุด** เช่น

```
let count = 0;
$: count = count + 1; // ห้ามทำแบบนี้เพราะจะวนลูปไม่หยุด
```

- ควรตรวจสอบ logic ให้ reactive statement ไม่ทำให้ค่าเปลี่ยนตัวเองวนลูป

---

5. ตัวอย่างโปรแกรมง่าย ๆ

```
<script>
  let price = 10;
  let quantity = 2;

  $: total = price * quantity;
</script>

<p>Price: {price}</p>
<p>Quantity: {quantity}</p>
<p>Total: {total}</p>

<button on:click={() => price += 5}>Increase Price</button>
<button on:click={() => quantity += 1}>Increase Quantity</button>
```

- เมื่อกดปุ่มเพิ่มราคา หรือเพิ่มจำนวน total จะคำนวณและแสดงใหม่ทันที

นี่คือตัวอย่างโปรแกรม Svelte แบบเต็มไฟล์ พร้อมโครงสร้างโปรเจกต์ คำอธิบายโค้ด และผลการรัน จำนวน 3 โปรแกรมพื้นฐาน และ 3 โปรแกรมแนวประยุกต์ ที่เน้นการใช้ **$: เพื่อ re-compute** โดยเฉพาะ

โครงสร้างโปรเจกต์ (เหมือนกันทุกตัวอย่าง)

```
my-svelte-app/
└── src/
    ├── BasicReactive.svelte
    ├── TemperatureConverter.svelte
    ├── ShoppingCalculator.svelte
    ├── BudgetTracker.svelte
    ├── CryptoPriceTracker.svelte
    └── FormValidation.svelte
```

โปรแกรมพื้นฐาน 3 ตัวอย่าง

---

1. BasicReactive.svelte

```
<script>
  let a = 5;
  let b = 10;

  // reactive declaration คำนวณ sum ใหม่เมื่อ a หรือ b เปลี่ยน
  $: sum = a + b;
</script>

<p>a = {a}</p>
<p>b = {b}</p>
<p>Sum = {sum}</p>

<button on:click={() => a += 1}>Increase a</button>
<button on:click={() => b += 1}>Increase b</button>
```

คำอธิบายโค้ด:

- ตัวแปร a และ b เป็น state
- $: sum = a + b คือ reactive declaration คำนวณผลรวมใหม่ทุกครั้งที่ a หรือ b เปลี่ยน

- ปุ่มเพิ่มค่า a หรือ b จะทำให้ sum คำนวณและแสดงผลใหม่ทันที

ผลการรัน:

- เริ่มต้นแสดง a=5, b=10, sum=15
- กดปุ่มเพิ่มค่า จะเห็น sum อัปเดตตามค่าที่เปลี่ยน

---

2. TemperatureConverter.svelte

```
<script>
  let celsius = 0;

  // แปลงเป็นฟาเรนไฮต์โดย reactive declaration
  $: fahrenheit = (celsius * 9/5) + 32;
</script>

<label>
  Celsius:
  <input type="number" bind:value={celsius} />
</label>

<p>Fahrenheit: {fahrenheit}</p>
```

คำอธิบายโค้ด:

- รับค่าอุณหภูมิในองศาเซลเซียสจาก input
- $: แปลงเป็นฟาเรนไฮต์และแสดงผลทันทีเมื่อเซลเซียสเปลี่ยน

ผลการรัน:

- เมื่อพิมพ์ค่าเซลเซียส ผลลัพธ์ฟาเรนไฮต์อัปเดตตามแบบเรียลไทม์

---

3. ShoppingCalculator.svelte

```
<script>
  let price = 100;
  let quantity = 1;

  // คำนวณราคาทั้งหมด reactive
  $: total = price * quantity;

  // คำนวณภาษี 7%
```

```
  $: tax = total * 0.07;

  // ราคาสุทธิรวมภาษี
  $: finalPrice = total + tax;
</script>

<label>Price: <input type="number" bind:value={price} /></label>
<label>Quantity: <input type="number" bind:value={quantity} /></label>

<p>Total: ฿{total}</p>
<p>Tax (7%): ฿{tax.toFixed(2)}</p>
<p><strong>Final Price: ฿{finalPrice.toFixed(2)}</strong></p>
```

คำอธิบายโค้ด:

- Reactive declaration หลายตัวคำนวณราคา, ภาษี และรวมราคาสุทธิ
- ค่าอัปเดตทุกครั้งที่ price หรือ quantity เปลี่ยน

ผลการรัน:

- ป้อนราคาหรือจำนวนจะคำนวณราคาและภาษีอัตโนมัติและแสดงผลแบบเรียลไทม์

โปรแกรมแนวประยุกต์ 3 ตัวอย่าง

---

4. BudgetTracker.svelte

```
<script>
  let incomes = [1000, 500, 200];
  let expenses = [400, 150];

  $: totalIncome = incomes.reduce((a, b) => a + b, 0);
  $: totalExpenses = expenses.reduce((a, b) => a + b, 0);
  $: balance = totalIncome - totalExpenses;
</script>

<h2>Budget Tracker</h2>
<p>Total Income: ฿{totalIncome}</p>
<p>Total Expenses: ฿{totalExpenses}</p>
<p>Balance: ฿{balance}</p>
```

คำอธิบายโค้ด:

- คำนวณรายได้รวม รายจ่ายรวม และยอดคงเหลือแบบ reactive
- หากเพิ่ม/ลบหรือแก้ไขข้อมูลใน array ต้องรีแอสซายน์ใหม่เพื่อให้ reactive ทำงาน

ผลการรัน:

- แสดงยอดรวมและยอดคงเหลือตามข้อมูลที่มี

---

5. CryptoPriceTracker.svelte

```
<script>
  import { onMount } from 'svelte';

  let btcPrice = 0;
  let ethPrice = 0;
  let totalValue = 0;

  onMount(async () => {
    const res = await
fetch('https://api.coingecko.com/api/v3/simple/price?ids=bitcoin,ethereum&vs_currencies=usd');
    const data = await res.json();
    btcPrice = data.bitcoin.usd;
    ethPrice = data.ethereum.usd;
  });

  let btcAmount = 1;
  let ethAmount = 2;

  // คำนวณมูลค่ารวมแบบ reactive
  $: totalValue = (btcPrice * btcAmount) + (ethPrice * ethAmount);
</script>

<h2>Crypto Portfolio</h2>
<p>BTC Price: ${btcPrice}</p>
<p>ETH Price: ${ethPrice}</p>

<label>BTC Amount: <input type="number" bind:value={btcAmount} /></label>
```

```
<label>ETH Amount: <input type="number" bind:value={ethAmount} /></label>

<p>Total Portfolio Value: ${totalValue.toFixed(2)}</p>
```

คำอธิบายโค้ด:

- โหลดราคาคริปโตจาก API เมื่อ component mount
- คำนวณมูลค่ารวม portfolio โดยใช้ reactive declaration $:
- ค่า totalValue อัปเดตทันทีเมื่อราคา หรือจำนวนเหรียญเปลี่ยน

ผลการรัน:

- แสดงราคาคริปโตล่าสุด
- แก้จำนวนเหรียญจะคำนวณมูลค่ารวมใหม่ทันที

---

6. FormValidation.svelte

```
<script>
  let email = "";
  let password = "";

  // ตรวจสอบ email ว่าถูกต้องหรือไม่
  $: emailValid = /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);

  // ตรวจสอบ password อย่างง่ายว่ามากกว่า 6 ตัวอักษรหรือไม่
  $: passwordValid = password.length > 6;

  // ปุ่ม submit ใช้งานได้เมื่อทั้งสองถูกต้อง
  $: formValid = emailValid && passwordValid;
</script>

<form on:submit|preventDefault={() => alert("Form submitted!")}>
  <input type="email" bind:value={email} placeholder="Email" />
  {#if email && !emailValid}
    <p style="color:red;">Invalid email format</p>
  {/if}

  <input type="password" bind:value={password} placeholder="Password" />
  {#if password && !passwordValid}
```

```
    <p style="color:red;">Password must be more than 6 characters</p>
  {/if}

  <button type="submit" disabled={!formValid}>Submit</button>
</form>
```

คำอธิบายโค้ด:

- ใช้ reactive declarations ตรวจสอบความถูกต้องของ email และ password
- กำหนดสถานะปุ่ม submit ตามผลการตรวจสอบ
- แสดงข้อความเตือนแบบ reactive

ผลการรัน:

- ป้อน email และ password แล้วแสดงข้อความผิดพลาดทันทีหากไม่ถูกต้อง
- ปุ่ม Submit จะใช้งานได้เมื่อฟอร์มถูกต้องเท่านั้น

การใช้ Reactive Block ใน Svelte — เชิงลึก

---

1. Reactive Block คืออะไร?

- Reactive block คือ **บล็อกคำสั่งหลายบรรทัด** ที่ใช้เครื่องหมาย $: นำหน้า ตามด้วย { ... }
- บล็อกนี้จะถูก **รันใหม่ทุกครั้งเมื่อค่าที่ใช้อยู่ในบล็อกนั้นเปลี่ยนแปลง**
- เหมาะสำหรับการทำงานที่มีหลายคำสั่ง เช่น การคำนวณหลายค่า, การเรียกฟังก์ชัน side effect (log, fetch, update ตัวแปรอื่น ฯลฯ)

---

2. รูปแบบการใช้งาน

```
$: {
  // คำสั่งหลายบรรทัด
  const fullName = firstName + " " + lastName;
  console.log(`Name updated: ${fullName}`);
  updateUI(fullName);
}
```

- บล็อกนี้จะรันใหม่เมื่อใดก็ตามที่ค่า firstName หรือ lastName เปลี่ยนแปลง
- ไม่ต้องประกาศตัวแปรส่งออก (export) เสมอไป แค่เขียนคำสั่งในบล็อกก็ได้

---

3. การใช้งาน Reactive Block กับ Side Effects

- Reactive block สามารถใช้แทนการเขียน watch หรือ useEffect ใน React
- เหมาะสำหรับการทำ side effect ที่ต้องการให้เกิดขึ้นทันทีเมื่อค่าตัวแปรเปลี่ยน

4. ข้อควรระวัง

- หลีกเลี่ยงการเขียนโค้ดที่ทำให้เกิด **ลูป reactive ไม่สิ้นสุด** เช่น reactive block เปลี่ยนตัวแปรที่ถูกบล็อกนี้สังเกตอยู่
- เขียน reactive block ให้คำนวณหรือ side effect อย่างระมัดระวัง

5. ตัวอย่างโปรแกรมง่าย ๆ

```
<script>
  let count = 0;

  // reactive block รันทุกครั้งที่ count เปลี่ยน
  $: {
    console.log(`Count has changed to ${count}`);
  }
</script>

<button on:click={() => count++}>Increase</button>
```

- ทุกครั้งที่ count เปลี่ยน จะพิมพ์ข้อความลง console

นี่คือตัวอย่างโปรแกรม Svelte แบบเต็มไฟล์ พร้อมโครงสร้าง คำอธิบายโค้ด และผลการรัน จำนวน 3 โปรแกรมพื้นฐาน และ 3 โปรแกรมแนวประยุกต์ ที่เน้นการใช้ **Reactive Block ($: { ... })** โดยเฉพาะ

โครงสร้างโปรเจกต์ (เหมือนกันทุกตัวอย่าง)

```
my-svelte-app/
└── src/
    ├── CounterLogger.svelte
    ├── UserGreeting.svelte
    ├── PriceTaxCalculator.svelte
    ├── AutoSaveForm.svelte
    ├── WindowSizeTracker.svelte
    └── ThemeSwitcher.svelte
```

โปรแกรมพื้นฐาน 3 ตัวอย่าง

1. CounterLogger.svelte

```
<script>
  let count = 0;

  // reactive block รันทุกครั้งที่ count เปลี่ยน
  $: {
    console.log(`Count changed to ${count}`);
  }
</script>

<button on:click={() => count++}>Increase</button>
<p>Count: {count}</p>
```

คำอธิบายโค้ด:

- ตัวแปร count เก็บจำนวนคลิก
- Reactive block ใช้ console.log แสดงค่าของ count ทุกครั้งที่มันเปลี่ยน
- ปุ่มเพิ่มค่า count ทีละ 1 และแสดงค่าบนหน้าจอ

ผลการรัน:

- กดปุ่มทุกครั้ง จะเห็น console log แสดงค่าที่เปลี่ยน
- บนหน้าเว็บจะแสดงจำนวนคลิกล่าสุด

---

2. UserGreeting.svelte

```
<script>
  let firstName = "";
  let lastName = "";
  let fullName = "";

  // reactive block คำนวณ fullName และ log เมื่อชื่อเปลี่ยน
  $: {
    fullName = firstName + " " + lastName;
    console.log(`Full name updated: ${fullName}`);
  }
</script>

<label>First Name: <input bind:value={firstName} /></label>
```