Full Stack System Development (Draft-01)

การพัฒนาระบบสารสนเทศแบบครบองค์ประกอบ

Anirach Mingkhwan
INE-FITM-KMUTNB

November 24, 2025

คำนำ

หนังสือ เล่ม นี้ มี จุด มุ่ง หมาย เพื่อ เป็น แนวทาง ใน การ พัฒนา ระบบ สารสนเทศ แบบ ครบ องค์ ประกอบ (Full Stack System Development) โดยเน้นการผสมผสานทักษะและเครื่องมือที่จำเป็นสำหรับการพัฒนาแอปพลิเคชันที่สามารถทำงานได้ทั้งฝั่งผู้ใช้ (Front-End) และฝั่งเซิร์ฟเวอร์ (Back-End) หนังสือเล่มนี้ครอบคลุมหัวข้อที่สำคัญในการพัฒนาเว็บแอปพลิเคชัน รวมถึงการใช้ JavaScript, Node.js, Express, React, และฐานข้อมูล SQL และ NoSQL นอกจากนี้ยังได้อธิบายแนวคิดและเทคนิคการเขียนโปรแกรมเชิงวัตถุ การจัดการ API การใช้ Docker เพื่อการคอนเทนเนอร์แอปพลิเคชัน และการปรับใช้แอปพลิเคชันในคลาวด์

การพัฒนาระบบสารสนเทศแบบครบองค์ประกอบเป็นหนึ่งในทักษะที่ได้รับความนิยมมากในยุคปัจจุบัน เนื่องจากความสามารถในการพัฒนาแอปพลิเคชันที่ตอบสนองความต้องการของผู้ใช้ได้อย่างครบถ้วน และสามารถปรับปรุงและขยายระบบได้อย่างง่ายดาย หนังสือเล่มนี้ถูกออกแบบมาให้เหมาะสำหรับทั้งนักพัฒนาใหม่ที่ต้องการเริ่มต้นในสายงานนี้ และนักพัฒนาที่มีประสบการณ์แล้วที่ต้องการเพิ่มพูนทักษะใหม่ๆ

หวังว่าหนังสือเล่มนี้จะเป็นประโยชน์และช่วยให้ผู้อ่านสามารถพัฒนาแอปพลิเคชันที่มีคุณภาพสูง และสามารถก้าวหน้าในสายงานการพัฒนาระบบสารสนเทศแบบครบองค์ประกอบได้อย่างมั่นคง

ขอขอบคุณผู้อ่านทุกท่านที่ให้ความสนใจและหวังว่าหนังสือเล่มนี้จะช่วยเสริมสร้างความรู้และทักษะให้กับทุกท่านในการเดินทางสู่การเป็นนักพัฒนาที่มีความสามารถในระดับสูง

ขอให้ทุกท่านโชคดีในการพัฒนาระบบสารสนเทศแบบครบองค์ประกอบ

Anirach Mingkhwan
INE-FITM-KMUTNB
August 18, 2024

กิตติกรรมประกาศ

ข้าพเจ้าขอแสดงความขอบคุณอย่างสูงต่อทุกท่านที่มีส่วนช่วยให้หนังสือเล่มนี้เสร็จสมบูรณ์ ข้าพเจ้าขอขอบคุณเพื่อนร่วมงานที่ให้การสนับสนุนและคำแนะนำที่มีคุณค่า ซึ่งทำให้การเขียนหนังสือเล่มนี้เป็นไปได้

ขอขอบคุณครอบครัวและเพื่อนๆ ที่คอยให้กำลังใจและสนับสนุนในทุกช่วงเวลา ข้าพเจ้าขอขอบคุณเป็นพิเศษต่อผู้ที่มีส่วนช่วยตรวจทานต้นฉบับและให้คำแนะนำในการปรับปรุงเนื้อหาให้ดียิ่งขึ้น การสนับสนุนและความเข้าใจของท่านเป็นส่วนสำคัญที่ทำให้ข้าพเจ้าสามารถทำงานนี้ให้สำเร็จได้

สุดท้ายนี้ ข้าพเจ้าขอขอบคุณผู้อ่านทุกท่านที่ให้ความสนใจในหนังสือเล่มนี้ และหวังว่าหนังสือเล่มนี้จะเป็นประโยชน์แก่ท่านในการพัฒนาทักษะและความรู้ในสายงานการพัฒนาระบบสารสนเทศแบบครบองค์ประกอบ

Anirach Mingkhwan
INE-FITM-KMUTNB
August 18, 2024

สารบัญ

บทที่ 1

แนะนำภาษา JavaScript

1.1 ภาพรวมของ JavaScript: ประวัติและวิวัฒนาการของ JavaScript

JavaScript เป็นภาษาการเขียนโปรแกรมที่ถูกออกแบบมาให้สามารถทำงานในเบราว์เซอร์ ทำให้เว็บไซต์มีการโต้ตอบและเปลี่ยนแปลงตามการกระทำของผู้ใช้ได้ นอกจากนี้ยังเป็นส่วนสำคัญของเทคโนโลยีหลักของเว็บ รวมถึง HTML และ CSS ซึ่งเป็นโครงสร้างพื้นฐานที่จำเป็นสำหรับการพัฒนาเว็บสมัยใหม่

JavaScript เริ่มต้นการพัฒนาจากแนวคิดที่ต้องการสร้างภาษาโปรแกรมที่สามารถทำงานได้บนเบราว์เซอร์ เพื่อให้ผู้ใช้งานเว็บไซต์สามารถโต้ตอบกับหน้าเว็บได้โดยตรง แทนที่จะเป็นการแสดงผลแบบสถิติเท่านั้น ภาษา JavaScript ถูกพัฒนาและออกแบบให้สามารถเข้าใจได้ง่าย ทำให้เป็นที่นิยมในหมู่นักพัฒนาเว็บทั่วโลก การทำงานที่ง่ายและยืดหยุ่นของ JavaScript ช่วยให้นักพัฒนาสามารถสร้างประสบการณ์การใช้งานที่ดีขึ้นให้กับผู้ใช้ โดยเฉพาะการทำให้เว็บไซต์มีความอินเทอร์แอคทีฟมากขึ้น เช่น การตรวจสอบข้อมูลที่ป้อนในฟอร์มโดยไม่ต้องรีเฟรชหน้าเว็บ

1.1.1 การเกิดของ JavaScript

JavaScript เริ่มแรกถูกเรียกว่า **Mocha** ระหว่างการพัฒนา ต่อมาเปลี่ยนชื่อเป็น **LiveScript** และสุดท้ายเปลี่ยนชื่อเป็น **JavaScript**และสุดท้ายเปลี่ยนชื่อเป็น JavaScript เพื่อเป็นกลยุทธ์ทางการตลาดที่ใช้ความนิยมของ Java ซึ่งเป็นภาษาการเขียนโปรแกรมอีกภาษา แม้ว่าจะมีชื่อคล้ายกัน แต่ JavaScript และ Java เป็นภาษาที่แตกต่างกันทั้งในแง่ของการใช้งานและโครงสร้าง การเปลี่ยนชื่อทำให้ JavaScript กลายเป็นที่รู้จักและได้รับความสนใจอย่างรวดเร็ว

การเปลี่ยนชื่อเป็น JavaScript เพื่อให้ดูมีความสัมพันธ์กับ Java ที่กำลังได้รับความนิยมในขณะนั้น แม้ในความเป็นจริงสองภาษานี้จะมีโครงสร้างและการใช้งานที่แตกต่างกันมาก การตั้งชื่อเพื่อสร้างการยอมรับนี้เป็นส่วนหนึ่งของแผนการตลาดที่ทำให้ JavaScript ได้รับความสนใจจากนักพัฒนาอย่างมาก นอกจากนี้การที่ JavaScript เป็นภาษาที่ง่ายต่อการเรียนรู้ยังช่วยให้ผู้เริ่มต้นสามารถเข้าถึงการพัฒนาเว็บได้โดยง่าย

1.1.2 วิวัฒนาการของ JavaScript

JavaScript ได้รับการพัฒนาและปรับปรุงอย่างมากตั้งแต่เริ่มต้น โดยภาษา JavaScript ได้ถูกมาตรฐานโดยองค์กร European Computer Manufacturers Association (ECMA) ในปี 1997 ภายใต้ชื่อ **ECMAScript (ES)** ซึ่งมีการอัปเดตอย่างต่อเนื่อง เช่น เวอร์ชัน ES5 ในปี 2009 ที่เพิ่มฟีเจอร์เช่น strict mode และการรองรับ JSON และเวอร์ชัน ES6 ในปี 2015 ที่เพิ่มฟีเจอร์เช่น classes, modules, arrow functions และ template literals การอัปเดตเหล่านี้ทำให้ JavaScript มีความทรงพลังและยืดหยุ่นมากขึ้นในการพัฒนาโปรแกรมทั้งฝั่งลูกข่ายและฝั่งเซิร์ฟเวอร์

การพัฒนาของ JavaScript ไม่ได้หยุดนิ่งอยู่กับที่ แต่มีการอัปเดตเพื่อให้สอดคล้องกับการเปลี่ยนแปลงของเทคโนโลยีที่รวดเร็ว การเพิ่มฟีเจอร์ต่างๆ เช่น arrow functions ช่วยให้โค้ดกระชับและอ่านง่ายขึ้น นอกจากนี้ ES6 ยังมีการแนะนำโครงสร้างการเขียนโปรแกรมใหม่ๆ ที่ช่วยให้นักพัฒนาสามารถจัดการโค้ดได้ดีขึ้น และช่วยให้การพัฒนาซอฟต์แวร์ที่ซับซ้อนทำได้ง่ายขึ้นและเป็นระบบมากขึ้น

1.1.3 JavaScript ในปัจจุบัน

ปัจจุบัน JavaScript เป็นส่วนสำคัญของการพัฒนาเว็บสมัยใหม่ โดยมันสามารถใช้สร้างทุกอย่างตั้งแต่การแอนิเมชันง่ายๆ บนหน้าเว็บ ไปจนถึงเฟรมเวิร์ก Front-End ที่ซับซ้อนเช่น React, Vue.js, และ Angular นอกจากนี้ ด้วยการเกิดขึ้นของ **Node.js** ทำให้ JavaScript ได้ขยายการใช้งานมาสู่การพัฒนาฝั่งเซิร์ฟเวอร์ ทำให้ JavaScript กลายเป็นภาษาที่สามารถใช้พัฒนาได้ทั้ง Front-End และ Back-End

JavaScript ในปัจจุบันไม่ได้จำกัดเฉพาะการพัฒนาในฝั่งลูกข่าย (Front-End) เท่านั้น แต่ยังสามารถใช้งานในฝั่งเซิร์ฟเวอร์ (Back-End) ได้ด้วยเครื่องมืออย่าง Node.js นอกจากนี้ยังมีเฟรมเวิร์กและไลบรารีที่ช่วยให้การพัฒนาเว็บง่ายและรวดเร็วขึ้น เช่น React, Vue.js, และ Angular ที่ใช้ในการสร้างเว็บแอปพลิเคชันที่ซับซ้อน JavaScript ยังสามารถเชื่อมต่อกับฐานข้อมูลและใช้สร้าง API เพื่อทำให้การสื่อสารระหว่างระบบเป็นไปได้อย่างมีประสิทธิภาพ

1.2 ไวยากรณ์และโครงสร้างพื้นฐาน: ตัวแปร ชนิดข้อมูล ตัวดำเนินการ และนิพจน์

การเข้าใจไวยากรณ์และโครงสร้างพื้นฐานของ JavaScript เป็นสิ่งสำคัญสำหรับการเขียนโค้ดที่มีประสิทธิภาพ องค์ประกอบเหล่านี้เป็นพื้นฐานสำคัญในการพัฒนาโครงสร้างที่ซับซ้อนมากขึ้นใน JavaScript

การเขียนโปรแกรมใน JavaScript จำเป็นต้องเข้าใจไวยากรณ์และองค์ประกอบพื้นฐาน เช่น ตัวแปร ชนิดข้อมูล ตัวดำเนินการ และนิพจน์ เพราะองค์ประกอบเหล่านี้จะช่วยในการจัดการข้อมูลและประมวลผลคำสั่งที่ซับซ้อน การเข้าใจการทำงานของตัวแปรอย่างถูกต้องจะช่วยให้การจัดการหน่วยความจำในโปรแกรมเป็นไปได้อย่างมีประสิทธิภาพ และลดข้อผิดพลาดที่อาจเกิดขึ้น

1.2.1 ตัวแปร

ตัวแปรใน JavaScript ใช้เพื่อเก็บข้อมูลที่สามารถอ้างอิงและปรับเปลี่ยนได้ในโปรแกรม JavaScript รองรับคีย์เวิร์ดสามตัวสำหรับการประกาศตัวแปร: `var`, `let`, และ `const`

- **`var`**: เป็นวิธีการประกาศตัวแปรแบบดั้งเดิม แต่มีข้อจำกัดบางอย่าง เช่น การถูกทำให้เป็นฟังก์ชันสโคป และการยกตัวแปร (hoisting) การใช้ var อาจทำให้เกิดความสับสนในกรณีที่ต้องการใช้งานตัวแปรในระดับบล็อก

- **`let`**: แนะนำใน ES6 โดย `let` อนุญาตให้ประกาศตัวแปรในระดับบล็อกสโคป ทำให้ปลอดภัยกว่า **`var`**เนื่องจากการประกาศตัวแปรแบบ let จะไม่ยกตัวแปรขึ้นไปอยู่ด้านบนสุดของสโคป และช่วยป้องกันข้อผิดพลาดที่เกิดจากการประกาศตัวแปรซ้ำ

- **`const`**: แนะนำใน ES6 เช่นกัน โดย `const` ใช้สำหรับประกาศตัวแปรที่ไม่ควรถูกเปลี่ยนค่าใหม่ และมีบล็อกสโคปเช่นเดียวกับ **`let`**ก ารใช้ const ช่วยให้โค้ดมีความชัดเจนมากขึ้นว่า ตัวแปรนั้นไม่ควรเปลี่ยนค่า ทำให้โค้ดอ่านง่ายและป้องกันข้อผิดพลาดที่เกิดจากการเปลี่ยนแปลงค่าที่ไม่คาดคิด

การเลือกใช้ตัวแปรที่เหมาะสม เช่น let และ const ที่แนะนำใน ES6 ทำให้โค้ดมีความปลอดภัยและมีความชัดเจนในการใช้งานมากขึ้น const เหมาะสำหรับการประกาศค่าที่คงที่ เช่น ค่า pi หรือค่าคงที่ทางฟิสิกส์ ส่วน let เหมาะสำหรับการใช้งานที่ต้องการเปลี่ยนแปลงค่าในโปรแกรม

```
1
2 var name = "John";
3
4 let age = 30;
5
6 const isStudent = true;
```

Listing 1.1: Variable Declaration Example

1.2.2 ชนิดของข้อมูล

JavaScript รองรับชนิดข้อมูลหลายประเภท ซึ่งสามารถแบ่งออกเป็นสองประเภทหลัก ได้แก่ ชนิดข้อมูลพื้นฐาน (primitive) และชนิดข้อมูลไม่พื้นฐาน (non-primitive) การเข้าใจชนิดข้อมูลแต่ละประเภทเป็นสิ่งสำคัญเนื่องจากมีผลต่อการจัดเก็บและการประมวลผลข้อมูลภายในโปรแกรมอย่างมาก

ชนิดข้อมูลพื้นฐานใน JavaScript คือชนิดข้อมูลที่ไม่สามารถแบ่งย่อยออกไปได้ ซึ่งหมายความว่าแต่ละตัวแปรจะเก็บเพียงค่าหนึ่งเดียว และค่าของตัวแปรจะถูกจัดเก็บในหน่วยความจำโดยตรง เมื่อคุณสร้างตัวแปรใหม่ที่เป็นชนิดข้อมูลพื้นฐาน ตัวแปรนั้นจะได้รับการกำหนดค่าและแยกออกจากตัวแปรอื่น ชนิดข้อมูลพื้นฐานใน JavaScript มีดังนี้:

String: ใช้เก็บข้อความหรือสตริงของอักขระ เช่น 'Hello' หรือ "World" ตัวแปรชนิด string สามารถรวมกันได้โดยใช้ตัวดำเนินการ "+" ทำให้สามารถสร้างข้อความที่ยาวขึ้นได้

Number: ใช้เก็บค่าตัวเลข ซึ่งอาจเป็นจำนวนเต็มหรือจำนวนทศนิยม JavaScript ใช้ชนิดข้อมูล number สำหรับตัวเลขทุกประเภท ไม่ว่าจะเป็นจำนวนเต็ม (integer) หรือจำนวนทศนิยม (float) นอกจากนี้ยังสามารถรองรับค่าพิเศษเช่น Infinity และ NaN (Not-a-Number)

Boolean: ใช้เก็บค่าจริง (true) หรือเท็จ (false) ชนิดข้อมูลนี้มักใช้ในคำสั่งเงื่อนไขและการควบคุมการทำงานของโปรแกรม

Null: ใช้แสดงค่าที่ไม่มี หรือการอ้างอิงที่ไม่มีอยู่จริง เมื่อกำหนดค่าตัวแปรเป็น null หมายความว่าตัวแปรนั้นไม่มีค่าใด ๆ

Undefined: ค่าของตัวแปรที่ถูกประกาศแต่ยังไม่ได้กำหนดค่า ตัวแปรที่มีค่า undefined มักเกิดขึ้นเมื่อคุณประกาศตัวแปรแต่ยังไม่ได้กำหนดค่าใด ๆ ให้กับมัน

Symbol: ชนิดข้อมูลที่ไม่ซ้ำกันและไม่สามารถเปลี่ยนแปลงได้ แนะนำใน ES6 ใช้สำหรับการสร้างตัวระบุที่ไม่ซ้ำกัน ซึ่งเป็นประโยชน์ในการป้องกันการชนกันของชื่อคุณสมบัติในออบเจกต์ขนาดใหญ่

ชนิดข้อมูลไม่พื้นฐาน (non-primitive) หรือที่เรียกว่าออบเจกต์ เป็นชนิดข้อมูลที่สามารถเก็บค่าหลายค่าและอ้างอิงไปยังค่าหลายค่าได้พร้อมกัน ชนิดข้อมูลไม่พื้นฐานสามารถมีโครงสร้างที่ซับซ้อนและสามารถเก็บข้อมูลที่หลากหลายได้ ตัวอย่างของชนิดข้อมูลไม่พื้นฐานใน JavaScript ได้แก่:

Object: เป็นชนิดข้อมูลที่สามารถเก็บข้อมูลที่หลากหลายภายใต้ชื่อคุณสมบัติ (properties) แต่ละตัว ออบเจกต์สามารถสร้างได้โดยใช้เครื่องหมายปีกกา เช่น name: "John", age: 30 โดยมีคุณสมบัติและค่าที่เกี่ยวข้อง ออบเจกต์สามารถใช้เพื่อเก็บข้อมูลที่ซับซ้อน เช่น ข้อมูลของผู้ใช้ หรือการตั้งค่าต่างๆ ในแอปพลิเคชัน

Array: เป็นชนิดข้อมูลที่สามารถเก็บข้อมูลหลายค่าในรูปแบบของลำดับ โดยที่แต่ละค่าอาจเป็นชนิดข้อมูลที่แตกต่างกันได้ อาเรย์ถูกสร้างขึ้นโดยใช้เครื่องหมายวงเล็บเหลี่ยม เช่น และสามารถเข้าถึงแต่ละค่าได้โดยการใช้ดัชนี (index)

Function: ใน JavaScript ฟังก์ชันถือเป็นชนิดข้อมูลเช่นเดียวกัน ฟังก์ชันคือบล็อกของโค้ดที่สามารถนำกลับมาใช้ซ้ำได้ และสามารถกำหนดให้เป็นค่าของตัวแปรหรือส่งผ่านในรูปของพารามิเตอร์ได้ การใช้ฟังก์ชันช่วยให้โค้ดมีความยืดหยุ่นและง่ายต่อการบำรุงรักษา

ชนิดข้อมูลไม่พื้นฐานเหล่านี้สามารถเปลี่ยนแปลงหรือปรับปรุงข้อมูลภายในได้ และมักใช้ในการจัดเก็บข้อมูลที่ซับซ้อนหรือการจัดการโครงสร้างข้อมูลที่มีหลายมิติ เช่น การเก็บข้อมูลของนักเรียนหลายคนภายในออบเจกต์หรืออาเรย์เพื่อการเข้าถึงและการใช้งานที่สะดวกยิ่งขึ้น

การทำความเข้าใจชนิดข้อมูลทั้งสองประเภทนี้จะช่วยให้คุณสามารถจัดการและประมวลผลข้อมูลได้อย่างมีประสิทธิภาพ นอกจากนี้ยังช่วยในการป้องกันข้อผิดพลาดที่อาจเกิดจากการใช้ชนิดข้อมูลไม่ถูกต้อง เช่น การพยายามดำเนินการทางคณิตศาสตร์กับตัวแปรที่เป็น string หรือการเข้าถึงค่าภายในออบเจกต์ที่ไม่มีอยู่จริง การใช้ชนิดข้อมูลที่เหมาะสมกับลักษณะงานยังทำให้โค้ดอ่านง่ายและบำรุงรักษาได้ง่ายยิ่งขึ้นอีกด้วย

- **ชนิดข้อมูลพื้นฐาน**: รวมถึง `string`, `number`, `boolean`, `null`, `undefined`, และ `symbol`

- **ชนิดข้อมูลไม่พื้นฐาน**: รวมถึง objects, arrays, และ functions ซึ่งเป็นข้อมูลที่สามารถเปลี่ยนแปลงได้และสามารถเก็บค่าหลายค่าได้

```
1
2  let str = "Hello";  // string
3
4  let num = 42;       // number
5
6  let bool = true;    // boolean
7
8  let obj = { name: "John", age: 30 };  // object
9
10 let arr = [1, 2, 3];  // array
11
12 let func = function() { return "Hello World"; };  // function
13
14 let sym = Symbol('unique');  // symbol
```

Listing 1.2: Data Types Example

1.2.3 ตัวดำเนินการและนิพจน์

ตัวดำเนินการใน JavaScript ใช้เพื่อดำเนินการกับตัวแปรและค่าต่างๆ ตัวดำเนินการที่พบบ่อยได้แก่ ตัวดำเนินการทางคณิตศาสตร์ (`+`, `-`, `*`, `/`), ตัวดำเนินการกำหนดค่า (`=`, `+=`, `-=`), ตัวดำเนินการเปรียบเทียบ (`==`, `===`, `!=`, `!==`), และตัวดำเนินการเชิงตรรกะ (`&&`, `||`, `!`)

```
1
2  let a = 10;
3
4  let b = 20;
5
6  let sum = a + b;  // 30
7
8  let isEqual = (a === b);  // false
9
10 let isBothTrue = (a < b && b > 10);  // true
```

Listing 1.3: Operators Example

นิพจน์คือนิพจน์ที่เกิดจากการรวมตัวแปร ตัวดำเนินการ และค่าต่างๆ เพื่อให้ได้ผลลัพธ์

```
1
2  let result = (a + b) * 2;  // 60
```

Listing 1.4: Expression Example

1.3 โครงสร้างการควบคุม: คำสั่งเงื่อนไข วนรอบ และฟังก์ชัน

โครงสร้างการควบคุมใน JavaScript ใช้เพื่อกำหนดการทำงานของโปรแกรม ให้สามารถตัดสินใจ วนรอบการทำงาน และสร้างโค้ดที่สามารถนำกลับมาใช้ใหม่ได้

1.3.1 คำสั่งเงื่อนไข

คำสั่งเงื่อนไขใน JavaScript ใช้เพื่อดำเนินการต่างๆ ตามเงื่อนไขที่แตกต่างกัน คำสั่งเงื่อนไขที่พบบ่อยได้แก่ `if`, `else if`, `else`, และ `switch`

```
1
2  let age = 20;
3
4
5
6  if (age < 18) {
7
8      console.log("You are a minor");
9
10 } else if (age < 65) {
11
12     console.log("You are an adult");
13
14 } else {
15
16     console.log("You are a senior");
17
18 }
```

Listing 1.5: Conditional Statement Example

1.3.2 วนรอบ

การวนรอบใช้เพื่อดำเนินการกับโค้ดบล็อกหนึ่งซ้ำๆ จนกว่าเงื่อนไขที่กำหนดจะเป็นจริง วนรอบที่พบบ่อยใน JavaScript ได้แก่ `for`, `while`, และ `do...while`

```
1
2  for (let i = 0; i < 5; i++) {
3
4      console.log(i);
5
6  }
7
8
9
10 let count = 0;
11
12 while (count < 5) {
13
14     console.log(count);
15
16     count++;
17
18 }
```

Listing 1.6: Loop Example

1.3.3 ฟังก์ชัน

ฟังก์ชันใน JavaScript เป็นบล็อกของโค้ดที่ออกแบบมาเพื่อทำงานเฉพาะอย่าง ฟังก์ชันสามารถรับค่าที่เรียกว่า parameters และคืนค่า output ได้

```
1
2  function greet(name) {
3
4      return "Hello, " + name;
5
6  }
7
8
9
10 let message = greet("Alice");
11
12 console.log(message);  // "Hello, Alice"
```

Listing 1.7: Function Example

JavaScript ยังรองรับฟังก์ชันที่ไม่ระบุชื่อ (anonymous functions), Arrow Function (แนะนำใน ES6), และนิพจน์ฟังก์ชันที่ถูกเรียกใช้งานทันที (IIFE)

```
1
2  const multiply = (a, b) => a * b;
3
4  console.log(multiply(2, 3));  // 6
```

Listing 1.8: Arrow Function Example

1.4 การแนะนำให้รู้จักกับสภาพแวดล้อมเบราว์เซอร์: การใช้ JavaScript ในเว็บเบราว์เซอร์

JavaScript ถูกใช้ในเว็บเบราว์เซอร์เพื่อสร้างส่วนติดต่อผู้ใช้ที่มีการโต้ตอบและไดนามิก สภาพแวดล้อมของเบราว์เซอร์มีออบเจกต์และ API ที่สร้างขึ้นมาในตัวหลายตัว

ที่นักพัฒนาสามารถใช้เพื่อจัดการกับ Document Object Model (DOM), จัดการเหตุการณ์ และโต้ตอบกับเบราว์เซอร์

1.4.1 Document Object Model (DOM)

DOM เป็นอินเตอร์เฟซโปรแกรมสำหรับเอกสาร HTML และ XML มันแสดงโครงสร้างของเอกสารเป็นต้นไม้ของออบเจกต์ ทำให้ JavaScript สามารถโต้ตอบและปรับเปลี่ยนเนื้อหาและโครงสร้างของหน้าเว็บได้

```
1
2  let heading = document.getElementById("myHeading");
3
4
5
6  heading.innerHTML = "Hello, World!";
```

Listing 1.9: DOM Manipulation Example

1.4.2 การจัดการเหตุการณ์

เหตุการณ์ใน JavaScript คือการกระทำหรือเหตุการณ์ที่เกิดขึ้นในเบราว์เซอร์ เช่น การคลิก การกดปุ่ม หรือการเคลื่อนเมาส์ JavaScript อนุญาตให้คุณกำหนด event listeners ที่จะทำการดำเนินการเมื่อเกิดเหตุการณ์

```
1
2  let button = document.getElementById("myButton");
3
4
5
6  button.addEventListener("click", function() {
7
8      alert("You clicked the button!");
9
10 });
```

Listing 1.10: Event Handling Example

1.4.3 การโต้ตอบกับ Browser APIs

เบราว์เซอร์สมัยใหม่มี API หลายตัวที่อนุญาตให้นักพัฒนาสามารถโต้ตอบกับส่วนต่างๆ ของเบราว์เซอร์และอุปกรณ์ของผู้ใช้ได้ เช่น Fetch API สำหรับการทำคำร้องขอเครือข่าย, Geolocation API สำหรับการเข้าถึงข้อมูลตำแหน่งที่ตั้ง และ Web Storage API สำหรับการเก็บข้อมูลในเครื่อง

```
1
2  fetch('https://api.example.com/data')
3
4      .then(response => response.json())
5
6      .then(data => console.log(data))
7
8      .catch(error => console.error('Error:', error));
```

Listing 1.11: Fetch API Example

สรุปท้ายบท

บทนี้ได้แนะนำพื้นฐานของ JavaScript ตั้งแต่ประวัติและวิวัฒนาการ ไวยากรณ์และโครงสร้างพื้นฐาน โครงสร้างการควบคุม ไปจนถึงการใช้ JavaScript ในสภาพแวดล้อมของเบราว์เซอร์ แนวคิดพื้นฐานเหล่านี้เป็นสิ่งสำคัญในการสร้างแอปพลิเคชันที่ซับซ้อนมากขึ้นในเส้นทางการเรียนรู้ JavaScript ของคุณ

คำถามทบทวน:

1. JavaScript คืออะไร และวิวัฒนาการมาอย่างไรตั้งแต่เริ่มแรก?
2. คุณจะประกาศตัวแปรใน JavaScript ได้อย่างไร และความแตกต่างระหว่าง `var`, `let`, และ `const` คืออะไร?
3. โครงสร้างการควบคุมหลักใน JavaScript มีอะไรบ้าง และทำงานอย่างไร?
4. คุณจะปรับเปลี่ยน DOM ด้วย JavaScript ในเว็บเบราว์เซอร์ได้อย่างไร?

การอ่านเพิ่มเติม:

- *JavaScript: The Definitive Guide* Flanagan *[1]*
- *Eloquent JavaScript* Haverbeke *[2]*
- *Mozilla Developer Network (MDN) Web Docs: JavaScript* MDN contributors *[3]*

บทที่ 2

ฟังก์ชันและออบเจกต์ใน JavaScript

2.1 ฟังก์ชันใน JavaScript: การประกาศฟังก์ชัน, นิพจน์ฟังก์ชัน, และฟังก์ชันลูกศร

ฟังก์ชันเป็นหนึ่งในองค์ประกอบหลักของ JavaScript ที่ช่วยให้นักพัฒนาสามารถรวบรวมโค้ดที่ใช้ซ้ำได้และดำเนินการเฉพาะเจาะจง การเข้าใจวิธีการประกาศ เรียกใช้งาน และใช้ฟังก์ชันอย่างมีประสิทธิภาพเป็นสิ่งสำคัญในการเขียนโค้ด JavaScript ที่สะอาดและมีประสิทธิภาพ

2.1.1 การประกาศฟังก์ชัน

การประกาศฟังก์ชัน เป็นวิธีพื้นฐานในการกำหนดฟังก์ชันใน JavaScript วิธีนี้ช่วยให้คุณสามารถสร้างฟังก์ชันที่สามารถเรียกใช้ได้ทุกที่ในสคริปต์หลังจากที่มันถูกประกาศ เนื่องจากกระบวนการที่เรียกว่า "hoisting"

ไวยากรณ์ของการประกาศฟังก์ชัน:

```
function functionName(parameters) {
    // Code to be executed
}
```

Listing 2.1: Function Declaration Syntax

ตัวอย่างของการประกาศฟังก์ชันอย่างง่าย:

```
function greet(name) {
    return "Hello, " + name + "!";
}

console.log(greet("Alice")); // Output: Hello, Alice!
```

Listing 2.2: Simple Function Declaration Example

ในตัวอย่างนี้ ฟังก์ชัน `greet` ถูกประกาศด้วยพารามิเตอร์เดียวคือ `name` เมื่อเรียกใช้ ฟังก์ชันนี้จะคืนค่าสตริงข้อความต้อนรับ การประกาศฟังก์ชันเหมาะสำหรับการกำหนดฟังก์ชันมาตรฐานที่ใช้หลายครั้งในโค้ดของคุณ

2.1.2 นิพจน์ฟังก์ชัน

นิพจน์ฟังก์ชัน คือการสร้างฟังก์ชันและกำหนดให้กับตัวแปร ไม่เหมือนการประกาศฟังก์ชัน นิพจน์ฟังก์ชันไม่ได้ถูกยกขึ้น (hoisted) ซึ่งหมายความว่ามันไม่สามารถเรียกใช้ได้ก่อนที่จะถูกกำหนด

ไวยากรณ์ของนิพจน์ฟังก์ชัน:

```
const functionName = function(parameters) {
    // Code to be executed
};
```

Listing 2.3: Function Expression Syntax

ตัวอย่างของนิพจน์ฟังก์ชัน:

```
const greet = function(name) {
    return "Hello, " + name + "!";
};

console.log(greet("Bob")); // Output: Hello, Bob!
```

Listing 2.4: Function Expression Example

ในตัวอย่างนี้ ฟังก์ชัน `greet` ถูกกำหนดเป็นนิพจน์และกำหนดให้กับตัวแปร `greet` วิธีนี้ช่วยให้มีความยืดหยุ่นมากขึ้น เช่น การสร้างฟังก์ชันตามเงื่อนไขหรือแบบไดนามิกภายในฟังก์ชันอื่นๆ

นิพจน์ฟังก์ชันยังสามารถเป็นนิพจน์ฟังก์ชันที่ไม่ระบุชื่อ (anonymous function) ซึ่งหมายถึงฟังก์ชันนั้นไม่มีชื่อและถูกอ้างอิงโดยตัวแปรที่มันถูกกำหนดให้ วิธีนี้มักใช้ในตัวจัดการเหตุการณ์และ callback

ตัวอย่างของนิพจน์ฟังก์ชันที่ไม่ระบุชื่อ:

```
setTimeout(function() {
    console.log("This message will display after 3 seconds");
}, 3000);
```

Listing 2.5: Anonymous Function Expression Example

ในตัวอย่างนี้ ฟังก์ชันที่ไม่ระบุชื่อถูกส่งผ่านเป็นพารามิเตอร์ให้กับฟังก์ชัน `setTimeout` ซึ่งจะดำเนินการโค้ดภายในฟังก์ชันที่ไม่ระบุชื่อหลังจากเวลาที่กำหนด

2.1.3 ฟังก์ชันลูกศร

ฟังก์ชันลูกศร ถูกแนะนำใน ES6 (ECMAScript 2015) เป็นวิธีที่กระชับกว่าในการเขียนฟังก์ชันใน JavaScript ฟังก์ชันลูกศรช่วยทำให้ไวยากรณ์ของฟังก์ชันเรียบง่ายขึ้น โดยเฉพาะเมื่อฟังก์ชันมีขนาดเล็ก นอกจากนี้ยังจัดการกับคีย์เวิร์ด `this` แตกต่างจากฟังก์ชันทั่วไป ซึ่งสามารถเป็นประโยชน์ในบางสถานการณ์

ไวยากรณ์ของฟังก์ชันลูกศร:

```
const functionName = (parameters) => {
    // Code to be executed
};
```

Listing 2.6: Arrow Function Syntax

สำหรับฟังก์ชันที่มีพารามิเตอร์เดียว สามารถละเว้นวงเล็บได้:

```
const greet = name => "Hello, " + name + "!";
```

Listing 2.7: Concise Arrow Function Syntax

ตัวอย่างของฟังก์ชันลูกศร:

```
const multiply = (a, b) => a * b;

console.log(multiply(2, 3)); // Output: 6
```

Listing 2.8: Arrow Function Example

ในตัวอย่างนี้ ฟังก์ชัน **multiply** ถูกกำหนดด้วยไวยากรณ์ฟังก์ชันลูกศร โดยรับพารามิเตอร์สองตัวคือ **a** และ **b** และคืนค่าผลคูณของพวกมัน

ฟังก์ชันลูกศรมีประโยชน์อย่างยิ่งในสถานการณ์ที่คุณต้องส่งผ่านฟังก์ชันเป็นอาร์กิวเมนต์ เช่นในเมธอดของอาเรย์เช่น **map**, **filter**, และ **reduce**

ตัวอย่างของฟังก์ชันลูกศรในเมธอดของอาเรย์:

```
const numbers = [1, 2, 3, 4, 5];
const squares = numbers.map(number => number * number);

console.log(squares); // Output: [1, 4, 9, 16, 25]
```

Listing 2.9: Arrow Function in Array Methods Example

ในตัวอย่างนี้ เมธอด **map** นำฟังก์ชันลูกศรไปใช้กับแต่ละองค์ประกอบในอาเรย์ **numbers** ส่งผลให้เกิดอาเรย์ใหม่ที่ประกอบด้วยค่าที่ถูกยกกำลังสอง

ฟังก์ชันลูกศรต่างจากฟังก์ชันทั่วไปในวิธีการจัดการกับคีย์เวิร์ด **this** ในฟังก์ชันปกติ **this** จะอ้างถึงออบเจกต์ที่เรียกใช้ฟังก์ชัน ซึ่งอาจแตกต่างกันไปขึ้นอยู่กับบริบท อย่างไรก็ตามในฟังก์ชันลูกศร **this** ถูกผูกติดตามตัวอักษร ซึ่งหมายความว่ามันจะเก็บค่าของ **this** จากบริบทโดยรอบ

ตัวอย่างของ this ในฟังก์ชันลูกศร:

```
function Person(name) {
    this.name = name;
    this.sayName = () => {
        console.log(this.name);
    };
}

const person1 = new Person("Charlie");
person1.sayName(); // Output: Charlie
```

Listing 2.10: Arrow Function ‘this‘ Binding Example

ในตัวอย่างนี้ ฟังก์ชันลูกศรใน **sayName** ใช้ค่า **this** จากตัวสร้าง **Person** ทำให้มันอ้างอิงถึงอินสแตนซ์ของ **Person** ได้อย่างถูกต้อง

ฟังก์ชันลูกศรเป็นฟีเจอร์ที่มีพลังใน JavaScript โดยมอบไวยากรณ์ที่สะอาดและอ่านง่ายสำหรับการกำหนดฟังก์ชัน โดยเฉพาะในกรณีที่ฟังก์ชันปกติจะต้องใช้โค้ดที่ซับซ้อนมากขึ้น

2.2 การเขียนโปรแกรมเชิงวัตถุใน JavaScript: ออบเจกต์, คุณสมบัติ, เมธอด, และการสืบทอด

JavaScript เป็นภาษาที่มีความหลากหลายซึ่งสนับสนุนการเขียนโปรแกรมเชิงวัตถุ (OOP) ใน OOP ออบเจกต์เป็นองค์ประกอบหลักที่ประกอบด้วยคุณสมบัติ (ข้อมูล) และเมธอด (ฟังก์ชัน) ที่ทำงานกับข้อมูล การเข้าใจวิธีการสร้างและจัดการกับออบเจกต์เป็นสิ่งสำคัญสำหรับการเขียนโปรแกรม JavaScript อย่างชำนาญ

2.2.1 ออบเจกต์ใน JavaScript

ออบเจกต์ ใน JavaScript เป็นกลุ่มของคุณสมบัติที่แต่ละคุณสมบัติประกอบด้วยคู่คีย์-ค่า ออบเจกต์ช่วยให้คุณสามารถจำลองสิ่งที่เกิดขึ้นจริงได้โดยการรวมข้อมูลและพฤติกรรมที่เกี่ยวข้องไว้ในโครงสร้างเดียว

การสร้างออบเจกต์ด้วย Object Literals:

วิธีที่พบบ่อยที่สุดในการสร้างออบเจกต์ใน JavaScript คือการใช้ Object Literals วิธีนี้เป็นการสร้างที่ตรงไปตรงมาและช่วยให้คุณสามารถกำหนดคุณสมบัติและเมธอดภายในออบเจกต์ได้โดยตรง

```
const person = {
    name: "David",
    age: 25,
    greet: function() {
        console.log("Hello, my name is " + this.name);
    }
};

console.log(person.name); // Output: David
person.greet(); // Output: Hello, my name is David
```

Listing 2.11: Object Literal Example

ในตัวอย่างนี้ ออบเจกต์ `person` มีคุณสมบัติสองอย่าง (`name` และ `age`) และมีเมธอดหนึ่งอย่าง (`greet`) เมธอดนี้ใช้คีย์เวิร์ด `this` เพื่ออ้างอิงถึงคุณสมบัติของออบเจกต์

2.2.2 คุณสมบัติและเมธอด

คุณสมบัติ ในออบเจกต์ใช้ในการเก็บค่าที่บ่งบอกลักษณะของออบเจกต์ ขณะที่ **เมธอด** เป็นฟังก์ชันที่กำหนดพฤติกรรมของออบเจกต์

คุณสามารถเข้าถึงคุณสมบัติของออบเจกต์ได้โดยใช้ `dot notation` หรือ `bracket notation`:

ตัวอย่างของการเข้าถึงคุณสมบัติ:

```
console.log(person.name); // Output: David
console.log(person["age"]); // Output: 25
```

Listing 2.12: Accessing Object Properties Example

คุณยังสามารถเพิ่ม, อัปเดต, หรือลบคุณสมบัติได้อย่างไดนามิก:

ตัวอย่างของการปรับเปลี่ยนคุณสมบัติ:

```
person.name = "John";
person.job = "Engineer";

delete person.age;

console.log(person); // Output: { name: "John", greet: [Function: greet],
    job: "Engineer" }
```

Listing 2.13: Modifying Object Properties Example

ในตัวอย่างนี้ คุณสมบัติ **name** ถูกอัปเดต, คุณสมบัติ **job** ใหม่ถูกเพิ่ม และคุณสมบัติ **age** ถูกลบ

เมธอด เป็นฟังก์ชันที่เป็นส่วนหนึ่งของออบเจกต์และมักใช้ในการดำเนินการกับคุณสมบัติของออบเจกต์ เมธอดสามารถกำหนดได้โดยตรงภายใน Object Literal หรือเพิ่มเข้าไปอย่างไดนามิก:

ตัวอย่างของการกำหนดเมธอด:

```
const car = {
    brand: "Toyota",
    model: "Corolla",
    start: function() {
        console.log("The car is starting");
    }
};

car.start(); // Output: The car is starting
```

Listing 2.14: Defining Methods Example

ในตัวอย่างนี้ ออบเจกต์ **car** มีเมธอด **start** ที่พิมพ์ข้อความเมื่อถูกเรียกใช้

2.2.3 การสืบทอดใน JavaScript

การสืบทอดเป็นแนวคิดสำคัญใน OOP ที่ช่วยให้ออบเจกต์หนึ่งสามารถสืบทอดคุณสมบัติและเมธอดจากออบเจกต์อื่นได้ ใน JavaScript การสืบทอดถูกนำมาใช้ผ่านโปรโตไทป์

การสร้างออบเจกต์ด้วย Constructor Functions:

Constructor functions เป็นวิธีที่ใช้บ่อยในการสร้างออบเจกต์ที่มีโครงสร้างเดียวกัน พวกมันช่วยให้คุณสามารถกำหนดโครงร่างสำหรับออบเจกต์และสร้างอินสแตนซ์หลายตัวที่มีคุณสมบัติและเมธอดที่คล้ายกัน

```
function Animal(name, species) {
    this.name = name;
    this.species = species;
}

Animal.prototype.speak = function() {
    console.log(this.name + " says hello!");
};

const dog = new Animal("Buddy", "Dog");
const cat = new Animal("Whiskers", "Cat");

dog.speak(); // Output: Buddy says hello!
cat.speak(); // Output: Whiskers says hello!
```

Listing 2.15: Constructor Function Example

ในตัวอย่างนี้ ฟังก์ชัน **Animal** เป็นฟังก์ชันสร้างที่กำหนดโครงร่างสำหรับการสร้างสัตว์ เมธอด **speak** ถูกเพิ่มเข้าไปใน **Animal.prototype** ทำให้อินสแตนซ์ของ **Animal** ทุกตัวสามารถแชร์เมธอดนี้ได้

การสืบทอดด้วย Object.create

อีกวิธีหนึ่งในการสืบทอดใน JavaScript คือผ่านเมธอด **Object.create** ซึ่งช่วยให้คุณสามารถสร้างออบเจกต์ใหม่ที่สืบทอดจากออบเจกต์โปรโตไทป์ที่กำหนด

```
const animal = {
    speak: function() {
        console.log(this.name + " makes a sound");
    }
};

const dog = Object.create(animal);
dog.name = "Buddy";
dog.speak(); // Output: Buddy makes a sound
```

Listing 2.16: Inheritance with Object.create Example

ในตัวอย่างนี้ ออบเจกต์ **dog** ถูกสร้างด้วย **animal** เป็นโปรโตไทป์ของมัน โดยสืบทอดเมธอด **speak** จากออบเจกต์ **animal**

2.3 การทำความเข้าใจ Prototype Chain: วิธีที่ JavaScript จัดการกับการสืบทอด

โมเดลการสืบทอดของ JavaScript อิงจากโปรโตไทป์มากกว่าการสืบทอดแบบคลาสสิค การเข้าใจ **Prototype Chain** เป็นสิ่งสำคัญสำหรับการทำความเข้าใจวิธีการทำงานของการสืบทอดและความสัมพันธ์ของออบเจกต์ใน JavaScript

2.3.1 Prototype Chain คืออะไร?

Prototype Chain เป็นกลไกที่ JavaScript ใช้ในการสืบทอดคุณสมบัติและเมธอดจากออบเจกต์หนึ่งไปยังอีกออบเจกต์หนึ่ง ออบเจกต์ทุกตัวใน JavaScript มีโปรโตไทป์ซึ่งเป็นออบเจกต์อีกตัวหนึ่งที่สามารถสืบทอดคุณสมบัติและเมธอดได้ โซ่ของโปรโตไทป์นี้จะดำเนินต่อไปจนถึงโปรโตไทป์ **null** ซึ่งเป็นจุดสิ้นสุดของโซ่

ตัวอย่างของ Prototype Chain:

```
function Person(name) {
    this.name = name;
}

Person.prototype.greet = function() {
    console.log("Hello, " + this.name);
};

const alice = new Person("Alice");

console.log(alice.hasOwnProperty("name")); // true
console.log(alice.hasOwnProperty("greet")); // false
console.log(Object.getPrototypeOf(alice)); // Person { greet: [Function] }
```

Listing 2.17: Prototype Chain Example

ในตัวอย่างนี้ ออบเจกต์ `alice` เป็นอินสแตนซ์ของ `Person` เมธอด `greet` ไม่ได้อยู่ใน `alice` โดยตรง แต่ในโปรโตไทป์ของมันซึ่งก็คือ `Person.prototype` ตัวอย่างนี้แสดงให้เห็นว่าคุณสมบัติและเมธอดถูกสืบทอดผ่านโซ่โปรโตไทป์ได้อย่างไร

2.3.2 บทบาทของ `__proto__` และ `prototype`

ออบเจกต์ JavaScript ทุกตัวมีคุณสมบัติภายในที่เรียกว่า `[[Prototype]]` ซึ่งสามารถเข้าถึงได้ผ่าน `__proto__` คุณสมบัตินี้ชี้ไปยังโปรโตไทป์ของออบเจกต์ สำหรับฟังก์ชัน JavaScript จะเพิ่มคุณสมบัติ `prototype` โดยอัตโนมัติซึ่งอ้างอิงถึงออบเจกต์ที่อินสแตนซ์ของฟังก์ชันนั้นจะสืบทอดมา

ตัวอย่างของ `__proto__` และ `prototype`:

```
const person = {
    name: "John",
    greet: function() {
        console.log("Hello, " + this.name);
    }
};

const student = {
    course: "Math"
};

student.__proto__ = person;

console.log(student.name); // Output: John
student.greet(); // Output: Hello, John
```

Listing 2.18: `__proto__` and `prototype` Example

ในตัวอย่างนี้ ออบเจกต์ `student` สืบทอดจากออบเจกต์ `person` โดยตั้งค่า `student.__proto__` เป็น `person` ผลลัพธ์คือ `student` สามารถเข้าถึงคุณสมบัติ `name` และเมธอด `greet` จาก `person` ได้

2.3.3 การปิดบังคุณสมบัติใน Prototype Chain

เมื่อออบเจกต์มีคุณสมบัติที่มีชื่อเดียวกับหนึ่งในคุณสมบัติของโปรโตไทป์ของมัน คุณสมบัติของออบเจกต์จะ **ปิดบัง** คุณสมบัติของโปรโตไทป์ ซึ่งหมายความว่าคุณสมบัติของออบเจกต์เองจะถูกเข้าถึงแทนคุณสมบัติของโปรโตไทป์

ตัวอย่างของการปิดบังคุณสมบัติ:

```
const animal = {
    type: "Mammal"
};

const dog = Object.create(animal);
dog.type = "Canine";

console.log(dog.type); // Output: Canine
console.log(animal.type); // Output: Mammal
```

Listing 2.19: Property Shadowing Example

ในตัวอย่างนี้ ออบเจกต์ `dog` มีคุณสมบัติ `type` ที่ปิดบังคุณสมบัติ `type` ในโปรโตไทป์ของมันซึ่งก็คือ `animal` ผลลัพธ์คือ `dog.type` คืนค่า "Canine" ขณะที่ `animal.type` ยังคงเป็น "Mammal"

2.3.4 การแก้ไขโปรโตไทป์

คุณสามารถเพิ่มคุณสมบัติหรือเมธอดใหม่ให้กับโปรโตไทป์ได้ ซึ่งจะทำให้พร้อมใช้งานกับอินสแตนซ์ทั้งหมดที่สืบทอดจากโปรโตไทป์นั้น

ตัวอย่างของการแก้ไขโปรโตไทป์:

```
function Car(brand) {
    this.brand = brand;
}

Car.prototype.start = function() {
    console.log(this.brand + " is starting");
};

const myCar = new Car("Toyota");

myCar.start(); // Output: Toyota is starting

Car.prototype.stop = function() {
    console.log(this.brand + " is stopping");
};

myCar.stop(); // Output: Toyota is stopping
```

Listing 2.20: Modifying Prototype Example

ในตัวอย่างนี้ โปรโตไทป์ของฟังก์ชันสร้าง `Car` ถูกแก้ไขหลังจากที่อินสแตนซ์ถูกสร้างขึ้น เมธอด `stop` ใหม่พร้อมใช้งานทันทีสำหรับอินสแตนซ์ทั้งหมดของ `Car` รวมถึง `myCar`

2.4 การทำงานกับอาเรย์และสตริง: เมธอดและเทคนิคการจัดการ

อาเรย์และสตริงเป็นชนิดข้อมูลพื้นฐานใน JavaScript ที่มาพร้อมกับเมธอดในตัวมากมายสำหรับการจัดการ การเข้าใจและการใช้เมธอดเหล่านี้อย่างชำนาญเป็นสิ่งสำคัญสำหรับการจัดการและประมวลผลข้อมูลในแอปพลิเคชัน JavaScript

2.4.1 อาเรย์ใน JavaScript

อาเรย์ เป็นชนิดข้อมูลพิเศษใน JavaScript ที่ใช้ในการเก็บค่าหลายค่าในตัวแปรเดียว อาเรย์มีการจัดการดัชนีแบบศูนย์ซึ่งหมายความว่าองค์ประกอบแรกจะเข้าถึงได้ที่ดัชนี 0

การสร้างและการเข้าถึงอาเรย์:

```
const fruits = ["Apple", "Banana", "Cherry"];

console.log(fruits[0]); // Output: Apple
console.log(fruits.length); // Output: 3
```

Listing 2.21: Array Creation and Access Example

2.4.2 เมธอดทั่วไปของอาเรย์

JavaScript มีเมธอดหลายตัวสำหรับการจัดการอาเรย์ เช่น:

- **push**: เพิ่มหนึ่งหรือมากกว่าองค์ประกอบไปยังส่วนท้ายของอาเรย์
- **pop**: ลบองค์ประกอบสุดท้ายออกจากอาเรย์
- **shift**: ลบองค์ประกอบแรกออกจากอาเรย์
- **unshift**: เพิ่มหนึ่งหรือมากกว่าองค์ประกอบไปยังส่วนต้นของอาเรย์
- **slice**: คืนค่าชิ้นส่วนตื้นของอาเรย์
- **splice**: เปลี่ยนเนื้อหาของอาเรย์โดยการลบ, แทนที่ หรือเพิ่มองค์ประกอบ
- **map**: สร้างอาเรย์ใหม่ที่ประกอบด้วยผลลัพธ์จากการเรียกใช้ฟังก์ชันในทุกองค์ประกอบ
- **filter**: สร้างอาเรย์ใหม่ด้วยองค์ประกอบทั้งหมดที่ผ่านการทดสอบที่ใช้ฟังก์ชัน
- **reduce**: ดำเนินการฟังก์ชัน reducer ในแต่ละองค์ประกอบของอาเรย์ ส่งผลให้เกิดค่าผลลัพธ์เดียว

ตัวอย่างของเมธอดอาเรย์:

```
let numbers = [1, 2, 3, 4, 5];

numbers.push(6);
console.log(numbers); // Output: [1, 2, 3, 4, 5, 6]

numbers.pop();
console.log(numbers); // Output: [1, 2, 3, 4, 5]

let firstNumber = numbers.shift();
console.log(firstNumber); // Output: 1
console.log(numbers); // Output: [2, 3, 4, 5]

numbers.unshift(0);
console.log(numbers); // Output: [0, 2, 3, 4, 5]

let slicedNumbers = numbers.slice(1, 3);
console.log(slicedNumbers); // Output: [2, 3]

numbers.splice(2, 1);
console.log(numbers); // Output: [0, 2, 4, 5]

let squaredNumbers = numbers.map(num => num * num);
console.log(squaredNumbers); // Output: [0, 4, 16, 25]

let evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Output: [0, 2, 4]

let sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // Output: 11
```

Listing 2.22: Array Methods Example

2.4.3 สตริงใน JavaScript

สตริง เป็นลำดับของอักขระที่ใช้ในการแทนข้อความ สตริงใน JavaScript เป็นข้อมูลที่เปลี่ยนแปลงไม่ได้ ซึ่งหมายความว่าเมื่อสร้างสตริงขึ้นแล้วจะไม่สามารถเปลี่ยนแปลงได้ อย่างไรก็ตาม คุณสามารถจัดการสตริงและสร้างสตริงใหม่ได้โดยใช้เมธอดต่างๆ

การสร้างและการเข้าถึงสตริง:

```
let greeting = "Hello, World!";

console.log(greeting[0]); // Output: H
console.log(greeting.length); // Output: 13
```

Listing 2.23: String Creation and Access Example

2.4.4 เมธอดทั่วไปของสตริง

JavaScript มีเมธอดหลายตัวสำหรับการจัดการสตริง เช่น:

- **charAt**: คืนค่าอักขระที่ตำแหน่งดัชนีที่กำหนด
- **concat**: รวมสตริงสองหรือมากกว่าเข้าด้วยกัน

- **includes**: ตรวจสอบว่าสตริงหนึ่งมีอีกสตริงหนึ่งอยู่หรือไม่
- **indexOf**: คืนค่าดัชนีของการพบครั้งแรกของค่าที่กำหนด
- **replace**: แทนที่การพบของสตริงย่อยด้วยสตริงใหม่
- **split**: แบ่งสตริงเป็นอาเรย์ของสตริงย่อยตามตัวแบ่งที่กำหนด
- **substring**: คืนค่าส่วนหนึ่งของสตริงระหว่างสองดัชนี
- **toLowerCase**: เปลี่ยนสตริงให้เป็นตัวพิมพ์เล็ก
- **toUpperCase**: เปลี่ยนสตริงให้เป็นตัวพิมพ์ใหญ่
- **trim**: ลบช่องว่างที่อยู่ทั้งสองข้างของสตริง

ตัวอย่างของเมธอดสตริง:

```
let message = "  JavaScript is fun!  ";

console.log(message.charAt(0)); // Output: J

let newMessage = message.concat(" Let's learn more");
console.log(newMessage); // Output: "  JavaScript is fun!  Let's learn
    more"

console.log(message.includes("fun")); // Output: true

console.log(message.indexOf("fun")); // Output: 15

let replacedMessage = message.replace("fun", "awesome");
console.log(replacedMessage); // Output: "  JavaScript is awesome!  "

let words = message.trim().split(" ");
console.log(words); // Output: ["JavaScript", "is", "fun!"]

console.log(message.substring(2, 11)); // Output: "JavaScript"

console.log(message.toLowerCase()); // Output: "  javascript is fun!  "
console.log(message.toUpperCase()); // Output: "  JAVASCRIPT IS FUN!  "

console.log(message.trim()); // Output: "JavaScript is fun!"
```

Listing 2.24: String Methods Example

2.4.5 การผสมผสานอาเรย์และสตริง

อาเรย์และสตริงสามารถรวมกันในหลายวิธีเพื่อจัดการข้อมูลได้อย่างมีประสิทธิภาพ ตัวอย่างเช่น คุณสามารถรวมองค์ประกอบของอาเรย์เป็นสตริงหรือแบ่งสตริงเป็นอาเรย์

ตัวอย่างของการผสมผสานอาเรย์และสตริง:

```
let sentence = "The quick brown fox jumps over the lazy dog";

let wordsArray = sentence.split(" ");
console.log(wordsArray); // Output: ["The", "quick", "brown", "fox",
    "jumps", "over", "the", "lazy", "dog"]

let newSentence = wordsArray.join("-");
console.log(newSentence); // Output:
    "The-quick-brown-fox-jumps-over-the-lazy-dog"
```

Listing 2.25: Combining Arrays and Strings Example

ในตัวอย่างนี้ เมธอด **split** ถูกใช้เพื่อแบ่งประโยคออกเป็นอาเรย์ของคำ และเมธอด **join** ถูกใช้ในการรวมองค์ประกอบของอาเรย์กลับเป็นสตริงโดยมีเครื่องหมายขีดกลางเป็นตัวคั่น

สรุปท้ายบท

บทนี้ได้เสนอข้อมูลที่ครอบคลุมเกี่ยวกับฟังก์ชันและออบเจกต์ใน JavaScript เราเริ่มต้นด้วยการให้ภาพรวมของฟังก์ชัน รวมถึงการประกาศฟังก์ชัน, นิพจน์ฟังก์ชัน, และฟังก์ชันลูกศร โดยเน้นไวยากรณ์ การใช้งาน และความแตกต่างระหว่างกัน จากนั้นบทนี้ได้ลงลึกในเรื่องของการเขียนโปรแกรมเชิงวัตถุใน JavaScript โดยครอบคลุมวิธีการสร้างและจัดการกับออบเจกต์ คุณสมบัติ และเมธอด รวมถึงการใช้การสืบทอดผ่านโปรโตไทป์และฟังก์ชันสร้าง

การทำความเข้าใจ Prototype Chain และวิธีที่ JavaScript จัดการกับการสืบทอดเป็นสิ่งสำคัญสำหรับการทำความเข้าใจวิธีการทำงานของความสัมพันธ์ระหว่างออบเจกต์ในภาษา นอกจากนี้ เรายังได้สำรวจการทำงานกับอาเรย์และสตริง โดยเน้นเมธอดต่างๆ ที่มีสำหรับการจัดการโครงสร้างข้อมูลเหล่านี้ การชำนาญในแนวคิดเหล่านี้เป็นสิ่งสำคัญสำหรับการพัฒนาแอปพลิเคชัน JavaScript ที่ซับซ้อนและมีประสิทธิภาพ

คำถามทบทวน:

1. อะไรคือความแตกต่างระหว่างการประกาศฟังก์ชัน, นิพจน์ฟังก์ชัน, และฟังก์ชันลูกศรใน JavaScript?
2. คุณจะกำหนดและจัดการคุณสมบัติและเมธอดภายในออบเจกต์ JavaScript ได้อย่างไร?
3. อธิบาย Prototype Chain และบทบาทของมันในกระบวนการสืบทอดของ JavaScript?
4. คุณจะใช้เมธอดของอาเรย์และสตริงใน JavaScript เพื่อจัดการข้อมูลได้อย่างไร?

การอ่านเพิ่มเติม:

- *JavaScript: The Good Parts by Douglas Crockford* Crockford [4]
- *You Don't Know JS: Scope and Closures* Simpson [5]
- *Eloquent JavaScript* Haverbeke [2]

บทที่ 3

แนะนำ Node.js

3.1 Node.js คืออะไร?: การทำความเข้าใจสภาพแวดล้อมรันไทม์และสถาปัตยกรรมของมัน

Node.js เป็นสภาพแวดล้อมรันไทม์ที่สร้างขึ้นบนเอนจิน **V8** ของ Google Chrome ซึ่งออกแบบมาเพื่อรันโค้ด JavaScript นอกเว็บเบราว์เซอร์ การใช้ JavaScript ไม่ได้จำกัดเพียงแค่บนฝั่งลูกข่าย (Client-Side) เท่านั้น แต่ยังสามารถใช้ในฝั่งเซิร์ฟเวอร์ (Server-Side) ได้อีกด้วย Node.js นำเสนอความสามารถในการสร้างแอปพลิเคชันเครือข่ายที่มีประสิทธิภาพสูงและสามารถรองรับการเชื่อมต่อแบบไม่จำกัดจำนวนพร้อม ๆ กันได้

3.1.1 สถาปัตยกรรมของ Node.js

Node.js ใช้ **event-driven architecture** ซึ่งหมายความว่าโค้ดจะทำงานในลักษณะที่เน้นการตอบสนองต่อเหตุการณ์ เช่น การร้องขอ HTTP การตอบสนองแบบนี้ทำให้ Node.js มีความสามารถในการจัดการกับการเชื่อมต่อหลาย ๆ การเชื่อมต่อในเวลาเดียวกันได้อย่างมีประสิทธิภาพ โดยไม่ต้องสร้างเธรดใหม่สำหรับการเชื่อมต่อแต่ละครั้ง

Non-blocking I/O เป็นหนึ่งในคุณสมบัติหลักของ Node.js ซึ่งหมายถึงการที่โค้ดสามารถทำงานต่อไปได้โดยไม่ต้องรอให้การดำเนินการ I/O เสร็จสิ้น การดำเนินการ I/O เช่น การอ่านไฟล์จากดิสก์หรือการดึงข้อมูลจากฐานข้อมูลนั้นมักจะใช้เวลานาน ด้วยวิธี non-blocking I/O, Node.js สามารถจัดการกับการร้องขอได้หลาย ๆ การร้องขอพร้อมกันโดยไม่ต้องรอการดำเนินการเหล่านี้เสร็จสิ้น

Single-threaded event loop เป็นหัวใจหลักของ Node.js ซึ่งทำให้สามารถจัดการกับการเชื่อมต่อหลาย ๆ การเชื่อมต่อพร้อมกันได้โดยใช้เธรดเดียว การทำงานของมันเหมือนกับการที่เรามีรายการงานที่ต้องทำ (task queue) และเมื่อมีเหตุการณ์ใด ๆ เกิดขึ้น Node.js จะดำเนินการตามรายการงานนี้ทีละงานโดยไม่ต้องรอให้งานใดงานหนึ่งเสร็จสิ้นก่อนที่จะดำเนินการงานถัดไป

3.1.2 การนำ Node.js มาใช้งานในแอปพลิเคชัน

Node.js ถูกนำมาใช้ในหลายด้านของการพัฒนาเว็บ เช่น การสร้าง API, เว็บเซิร์ฟเวอร์, และการพัฒนาแอปพลิเคชันแบบเรียลไทม์ ความสามารถในการจัดการกับการร้องขอพร้อม ๆ กันอย่างมีประสิทธิภาพทำให้ Node.js เป็นเครื่องมือที่เหมาะสมสำหรับการพัฒนาเว็บแอปพลิเคชันที่มีการโหลดสูงและต้องการประสิทธิภาพสูง

การใช้ JavaScript ในฝั่งเซิร์ฟเวอร์ทำให้การพัฒนา Full-Stack ง่ายขึ้นมาก เนื่องจากนักพัฒนาสามารถใช้ภาษาเดียวกันทั้งในฝั่งลูกข่ายและฝั่งเซิร์ฟเวอร์ สิ่งนี้ช่วยลดความซับซ้อนในการพัฒนาซอฟต์แวร์และทำให้ทีมพัฒนาสามารถทำงานร่วมกันได้ดีขึ้น

Node.js ยังมีเอกสารประกอบและชุมชนที่เติบโตอย่างรวดเร็ว ซึ่งทำให้เป็นทางเลือกที่น่าสนใจสำหรับนักพัฒนา ทั้งนี้เพราะการเรียนรู้และการนำไปใช้เป็นไปได้ง่าย เนื่องจากมีทรัพยากรที่หลากหลายและมีการสนับสนุนอย่างกว้างขวางจากชุมชน

3.2 การตั้งค่าสภาพแวดล้อม Node.js: การติดตั้ง Node.js, npm, และการตั้งค่าโปรเจกต์

การเริ่มต้นใช้งาน Node.js จำเป็นต้องมีการตั้งค่าสภาพแวดล้อมการพัฒนา ซึ่งรวมถึงการติดตั้ง Node.js และ npm (Node Package Manager) รวมถึงการตั้งค่าโปรเจกต์เบื้องต้น ในส่วนนี้ เราจะได้เรียนรู้วิธีการติดตั้งและการเริ่มต้นใช้งานโปรเจกต์ Node.js อย่างถูกต้อง

3.2.1 การติดตั้ง Node.js และ npm

Node.js มาพร้อมกับ **npm** ซึ่งเป็นเครื่องมือที่ใช้ในการจัดการแพ็กเกจหรือโมดูลที่จำเป็นสำหรับการพัฒนาแอปพลิเคชัน Node.js npm ช่วยให้นักพัฒนาสามารถติดตั้ง, อัปเดต, และลบแพ็กเกจจากโปรเจกต์ได้อย่างง่ายดาย

การติดตั้ง Node.js บน Windows/MacOS:

1. ไปที่เว็บไซต์ทางการของ Node.js https://nodejs.org/ และดาวน์โหลดตัวติดตั้งที่ตรงกับระบบปฏิบัติการของคุณ
2. รันตัวติดตั้งและทำตามขั้นตอนการติดตั้ง
3. หลังจากการติดตั้งเสร็จสิ้น คุณสามารถตรวจสอบว่า Node.js และ npm ถูกติดตั้งเรียบร้อยแล้วหรือไม่โดยการใช้คำสั่งต่อไปนี้ใน command prompt หรือ terminal:

```
node -v
npm -v
```

คำสั่งนี้จะแสดงหมายเลขเวอร์ชันของ Node.js และ npm หากการติดตั้งสำเร็จ

การติดตั้ง Node.js บน Linux:

บนระบบ Linux คุณสามารถติดตั้ง Node.js และ npm ได้โดยใช้ package manager ของระบบ ตัวอย่างเช่น บน Ubuntu หรือ Debian คุณสามารถใช้คำสั่งต่อไปนี้:

```
sudo apt update
sudo apt install nodejs npm
```

หลังจากการติดตั้ง คุณสามารถตรวจสอบเวอร์ชันของ Node.js และ npm ได้เช่นเดียวกับใน Windows/MacOS

3.2.2 การตั้งค่าโปรเจกต์ Node.js

หลังจากที่ติดตั้ง Node.js และ npm เรียบร้อยแล้ว ขั้นตอนถัดไปคือการตั้งค่าโปรเจกต์ Node.js ซึ่งเริ่มต้นด้วยการสร้างไฟล์ `package.json` ไฟล์นี้เป็นไฟล์ที่ใช้ในการเก็บข้อมูลเกี่ยวกับโปรเจกต์ เช่น ชื่อ, เวอร์ชัน, สคริปต์ที่ใช้ในการรันโปรเจกต์, และรายการของแพ็กเกจที่โปรเจกต์ใช้งาน

การสร้างโปรเจกต์ Node.js เบื้องต้น:

1. สร้างโฟลเดอร์ใหม่สำหรับโปรเจกต์ของคุณ:

```
mkdir my-nodejs-project
cd my-nodejs-project
```

1. ใช้คำสั่ง `npm init` เพื่อสร้างไฟล์ `package.json`:

```
npm init
```

คำสั่งนี้จะนำคุณเข้าสู่กระบวนการตั้งค่าไฟล์ `package.json` โดยคุณจะถูกถามข้อมูลเกี่ยวกับโปรเจกต์ เช่น ชื่อ, เวอร์ชัน, คำอธิบาย, และสคริปต์ที่ใช้ในการรันโปรเจกต์ หลังจากที่คุณตอบคำถามทั้งหมดแล้ว ไฟล์ `package.json` จะถูกสร้างขึ้นในโฟลเดอร์โปรเจกต์ของคุณ

1. หลังจากที่คุณสร้างไฟล์ `package.json` เรียบร้อยแล้ว คุณสามารถเริ่มติดตั้งแพ็กเกจที่จำเป็นสำหรับโปรเจกต์ของคุณโดยใช้คำสั่ง `npm install` ตามด้วยชื่อแพ็กเกจ

ตัวอย่าง:

```
npm install express
```

คำสั่งนี้จะติดตั้งแพ็กเกจ **Express** ซึ่งเป็นเฟรมเวิร์กที่ใช้ในการสร้างเว็บแอปพลิเคชันใน Node.js และบันทึกข้อมูลของแพ็กเกจลงในไฟล์ `package.json`

3.2.3 การสร้างโครงสร้างไฟล์โปรเจกต์:

เมื่อคุณเริ่มต้นโปรเจกต์ Node.js คุณควรจัดโครงสร้างไฟล์และโฟลเดอร์ให้ชัดเจน เพื่อให้การจัดการโปรเจกต์เป็นไปได้อย่างมีประสิทธิภาพ โครงสร้างพื้นฐานที่มักใช้กันทั่วไปคือ:

```
my-nodejs-project/
 node_modules/       # Directory that stores installed packages via npm
 public/             # Directory for public files like HTML, CSS, JS
 src/                # Directory for project source code
    index.js        # Entry point of the project
    routes/         # Directory for route files
 .gitignore          # File specifying files/folders to be ignored by Git
 package.json        # File storing project information and packages
 README.md           # Project documentation file
```

Listing 3.1: Example of a Node.js Project Structure

โครงสร้างนี้ช่วยให้โปรเจกต์ของคุณมีการจัดระเบียบที่ดีและง่ายต่อการขยายเพิ่มเติม

3.3 แนวคิดพื้นฐานของ Node.js: โมดูล, `require`, และ `exports`

Node.js ใช้ระบบโมดูล (Module System) ซึ่งช่วยให้คุณสามารถจัดโครงสร้างโค้ดเป็นส่วน ๆ ที่แยกออกจากกันและสามารถนำกลับมาใช้ใหม่ได้ โมดูลเหล่านี้สามารถนำเข้ามาใช้ในไฟล์อื่น ๆ ผ่านฟังก์ชัน `require` และคุณสามารถทำให้ฟังก์ชัน, ออบเจกต์, หรือค่าต่าง ๆ พร้อมใช้งานในไฟล์อื่น ๆ ผ่าน `exports`

3.3.1 โมดูลใน Node.js

โมดูลใน Node.js คือไฟล์ JavaScript ที่มีฟังก์ชัน, ออบเจกต์, หรือค่าต่าง ๆ ที่สามารถนำไปใช้ในไฟล์อื่น ๆ ได้ การใช้โมดูลช่วยให้โค้ดของคุณมีความยืดหยุ่นและสามารถจัดการได้ง่ายขึ้น โดยการแยกโค้ดออกเป็นส่วน ๆ ที่มีความรับผิดชอบเฉพาะ

ตัวอย่างของโมดูล:

สร้างไฟล์ `math.js` เพื่อใช้เป็นโมดูลที่มีฟังก์ชันคำนวณทางคณิตศาสตร์:

```
 // math.js

function add(a, b) {
    return a + b;
}

function subtract(a, b) {
    return a - b;
}

module.exports = {
    add,
    subtract
};
```

Listing 3.2: Example of a Math Module with Exported Functions

ในตัวอย่างนี้ ฟังก์ชัน `add` และ `subtract` ถูกกำหนดในไฟล์ `math.js` และทำให้พร้อมใช้งานในไฟล์อื่น ๆ โดยใช้ `module.exports`

3.3.2 การใช้ require ใน Node.js

ฟังก์ชัน **require** ใน Node.js ใช้สำหรับนำเข้าฟังก์ชัน, ออบเจกต์, หรือค่าต่าง ๆ จากโมดูลอื่นเข้ามาใช้ในไฟล์ปัจจุบัน

ตัวอย่างการใช้ require:

สร้างไฟล์ **app.js** เพื่อใช้ฟังก์ชันจากโมดูล **math.js**:

```
// app.js

const math = require('./math');

const result = math.add(5, 3);
console.log("The addition result is:", result); // Result: 8
```

Listing 3.3: Example of Using a Math Module in an Application

ในตัวอย่างนี้ ฟังก์ชัน **add** จากโมดูล **math.js** ถูกนำเข้าและใช้ในไฟล์ **app.js** โดยใช้ **require**

3.3.3 การใช้ exports ใน Node.js

exports เป็นออบเจกต์ที่ใช้ในการกำหนดว่าฟังก์ชัน, ออบเจกต์, หรือค่าต่าง ๆ ใดที่ควรจะทำให้พร้อมใช้งานสำหรับไฟล์อื่น ๆ

คุณสามารถใช้ **exports** ได้สองวิธีหลัก:

1. การใช้ **exports** เพื่อกำหนดแต่ละฟังก์ชันหรือค่าที่ต้องการส่งออก:

```
// math.js

exports.add = function(a, b) {
    return a + b;
};

exports.subtract = function(a, b) {
    return a - b;
};
```

Listing 3.4: Example of a Math Module Using Exports

ในตัวอย่างนี้ ฟังก์ชัน **add** และ **subtract** ถูกกำหนดโดยตรงบนออบเจกต์ **exports**

1. การใช้ **module.exports** เพื่อส่งออกออบเจกต์ทั้งหมด:

```
// math.js

function add(a, b) {
    return a + b;
}

function subtract(a, b) {
    return a - b;
}

module.exports = {
    add,
    subtract
};
```

Listing 3.5: Example of a Simple Math Module

วิธีนี้ใช้เมื่อคุณต้องการส่งออกหลายฟังก์ชันหรือคุณสมบัติในรูปแบบของออบเจกต์

3.4 การสร้างเว็บเซิร์ฟเวอร์แบบง่าย: การใช้โมดูล HTTP ใน Node.js

หนึ่งในความสามารถที่น่าสนใจของ Node.js คือการสร้างเว็บเซิร์ฟเวอร์ด้วยการใช้โมดูล `HTTP` ซึ่งเป็นโมดูลในตัวที่ให้เครื่องมือพื้นฐานสำหรับการจัดการกับการร้องขอ HTTP และการตอบสนอง HTTP

3.4.1 การสร้างเว็บเซิร์ฟเวอร์เบื้องต้น

การสร้างเว็บเซิร์ฟเวอร์เบื้องต้นใน Node.js ทำได้โดยใช้โมดูล `HTTP` ซึ่งมีฟังก์ชัน `createServer` สำหรับสร้างเซิร์ฟเวอร์ที่สามารถจัดการกับการร้องขอและการตอบสนอง

```
const http = require('http');

// Create a server
const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello, World!\n');
});

// Specify the port the server will listen to
const port = 3000;
server.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 3.6: Example of Creating a Web Server

ในตัวอย่างนี้ เซิร์ฟเวอร์ถูกสร้างขึ้นโดยใช้ `http.createServer` ซึ่งรับฟังก์ชัน callback ที่มีพารามิเตอร์ `req` (คำร้องขอ) และ `res` (การตอบสนอง) เซิร์ฟเวอร์จะตอบสนองด้วยข้อความ "Hello, World!" ทุกครั้งที่มีการร้องขอเข้ามา

3.4.2 การจัดการเส้นทาง (Routing) ใน Node.js

การจัดการเส้นทาง (Routing) เป็นกระบวนการที่เว็บเซิร์ฟเวอร์ใช้ในการกำหนดว่าเซิร์ฟเวอร์ควรตอบสนองอย่างไรเมื่อได้รับคำร้องขอสำหรับเส้นทาง (URL) ต่าง ๆ

```
const http = require('http');

const server = http.createServer((req, res) => {
    if (req.url === '/') {
        res.statusCode = 200;
        res.setHeader('Content-Type', 'text/plain');
        res.end('Welcome to the homepage!\n');
    } else if (req.url === '/about') {
        res.statusCode = 200;
        res.setHeader('Content-Type', 'text/plain');
        res.end('This is the about page.\n');
    } else {
        res.statusCode = 404;
        res.setHeader('Content-Type', 'text/plain');
        res.end('Page not found.\n');
    }
});

const port = 3000;
server.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 3.7: Example of Route Handling

ในตัวอย่างนี้ เซิร์ฟเวอร์ถูกตั้งค่าให้จัดการกับเส้นทาง `/` และ `/about` เมื่อผู้ใช้ร้องขอเส้นทางใดเส้นทางหนึ่ง เซิร์ฟเวอร์จะตอบสนองด้วยข้อความที่กำหนดไว้ หากผู้ใช้ร้องขอเส้นทางอื่นที่ไม่ได้กำหนด เซิร์ฟเวอร์จะตอบสนองด้วยข้อความ "Page not found."

3.4.3 การใช้งานโมดูลเสริมเช่น Express

แม้ว่า Node.js จะมีความสามารถในการสร้างเว็บเซิร์ฟเวอร์และจัดการเส้นทาง แต่การจัดการโค้ดที่ซับซ้อนอาจจะทำได้ยากและมีข้อจำกัด ดังนั้นนักพัฒนามักจะใช้เฟรมเวิร์กเช่น **Express** เพื่อทำให้กระบวนการนี้ง่ายขึ้นและมีประสิทธิภาพมากขึ้น

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Welcome to the homepage!');
});

app.get('/about', (req, res) => {
    res.send('This is the about page.');
});

app.use((req, res) => {
    res.status(404).send('Page not found.');
});

const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 3.8: Creating a Web Server with Express

ในตัวอย่างนี้ เราใช้ `Express` เพื่อสร้างเว็บเซิร์ฟเวอร์และจัดการเส้นทางได้ง่ายขึ้น `Express` ทำให้การจัดการเส้นทางมีความเรียบง่ายและมีประสิทธิภาพมากขึ้น และยังมีฟีเจอร์เสริมอื่น ๆ ที่สามารถใช้ในการพัฒนาเว็บแอปพลิเคชันได้อย่างมีประสิทธิภาพ

สรุปท้ายบท

บทนี้ได้แนะนำคุณเกี่ยวกับ Node.js และความสามารถในการสร้างเว็บแอปพลิเคชันฝั่งเซิร์ฟเวอร์ เริ่มจากการทำความเข้าใจสภาพแวดล้อมรันไทม์และสถาปัตยกรรมของ Node.js ซึ่งประกอบด้วยสถาปัตยกรรมแบบ event-driven, non-blocking I/O และ single-threaded event loop

จากนั้นคุณได้เรียนรู้วิธีการตั้งค่าสภาพแวดล้อม Node.js ตั้งแต่การติดตั้ง Node.js และ NPM ไปจนถึงการตั้งค่าโปรเจกต์เบื้องต้น นอกจากนี้ยังได้ทำความรู้จักกับแนวคิดพื้นฐานของ Node.js เช่น การใช้โมดูล การใช้ฟังก์ชัน `require` และ `exports` เพื่อจัดโครงสร้างโค้ดให้มีประสิทธิภาพมากขึ้น

ในส่วนสุดท้ายของบทนี้ คุณได้เรียนรู้วิธีการสร้างเว็บเซิร์ฟเวอร์แบบง่าย ๆ โดยใช้โมดูล `HTTP` ใน Node.js และการจัดการเส้นทางภายในเซิร์ฟเวอร์ พร้อมทั้งการแนะนำการใช้เฟรมเวิร์ก `Express` เพื่อทำให้การพัฒนาเว็บแอปพลิเคชันด้วย Node.js มีประสิทธิภาพมากขึ้น

คำถามทบทวน:

1. Node.js มีสถาปัตยกรรมแบบใดที่ทำให้สามารถจัดการกับการเชื่อมต่อหลาย ๆ การเชื่อมต่อพร้อมกันได้อย่างมีประสิทธิภาพ?
2. คุณจะตั้งค่าสภาพแวดล้อมการพัฒนา Node.js และสร้างโปรเจกต์เบื้องต้นได้อย่างไร?
3. โมดูลใน Node.js คืออะไร และคุณสามารถนำเข้าฟังก์ชันหรือออบเจกต์จากโมดูลอื่นได้อย่างไร?
4. คุณจะสร้างเว็บเซิร์ฟเวอร์แบบง่าย ๆ ใน Node.js ได้อย่างไร และทำอย่างไรในการจัดการเส้นทางในเว็บเซิร์ฟเวอร์นั้น?

การอ่านเพิ่มเติม:

- *Node.js Documentation* Node.js Documentation Team [6]
- *Express.js Guide* Express.js Documentation Team [7]
- *Node.js Design Patterns* Casanova and Mammino [8]

บทที่ 4

การเขียนโปรแกรมแบบ Asynchronous ใน JavaScript และ Node.js

การเขียนโปรแกรมแบบ Asynchronous Programming เป็นส่วนสำคัญของ JavaScript และ Node.js ที่ช่วยให้นักพัฒนาสามารถดำเนินการงานต่าง ๆ เช่น การดำเนินการ I/O โดยไม่ต้องทำให้กระบวนการหลักของโปรแกรมหยุดชะงัก บทนี้จะอธิบายถึงวิธีการและรูปแบบต่าง ๆ ในการเขียนโค้ดแบบ Asynchronous รวมถึงการใช้ Callbacks, Promises, และ Async/Await นอกจากนี้ยังครอบคลุมถึงการเขียนโปรแกรมแบบ Event-Driven ใน Node.js และการทำงานกับ Timers รวมถึงการดำเนินการกับระบบไฟล์ใน Node.js อย่าง Asynchronous

4.1 การทำความเข้าใจการเขียนโปรแกรมแบบ Asynchronous: Callbacks, Promises, และ Async/Await

การเขียนโปรแกรมแบบ Asynchronous เป็นรูปแบบการเขียนโปรแกรมที่ช่วยให้การดำเนินการบางอย่างเกิดขึ้นได้โดยไม่ต้องรอให้การดำเนินการอื่นเสร็จสิ้นก่อน ซึ่งเป็นประโยชน์อย่างยิ่งในกรณีที่ต้องดำเนินการที่ใช้เวลานาน เช่น การร้องขอข้อมูลผ่านเครือข่ายหรือการทำงานกับระบบไฟล์

4.1.1 Callbacks

Callback คือฟังก์ชันที่ถูกส่งเป็นอาร์กิวเมนต์ให้กับฟังก์ชันอื่น และจะถูกเรียกใช้เมื่อการดำเนินการที่กำหนดเสร็จสิ้นแล้ว Callbacks เป็นวิธีการหลักในการจัดการกับการดำเนินการแบบ Asynchronous ใน JavaScript ก่อนที่ Promise และ Async/Await จะถูกแนะนำ

ตัวอย่างของ Callback:

```
function fetchData(callback) {
    setTimeout(() => {
        const data = { name: "John", age: 30 };
        callback(data);
    }, 2000);
}

function displayData(data) {
    console.log("Data received:", data);
}

fetchData(displayData);
```

Listing 4.1: ตัวอย่างของ Callback

ในตัวอย่างนี้ ฟังก์ชัน `fetchData` จะใช้เวลาประมาณ 2 วินาทีในการดำเนินการ จากนั้นจะเรียกใช้ฟังก์ชัน `displayData` พร้อมกับส่งข้อมูลที่ได้มา ฟังก์ชัน `displayData` จะถูกเรียกใช้เมื่อการดึงข้อมูลเสร็จสิ้น

แม้ว่าการใช้ Callbacks จะช่วยในการจัดการกับการเขียนโปรแกรมแบบ Asynchronous ได้ แต่บางครั้งก็อาจนำไปสู่สถานการณ์ที่เรียกว่า **Callback Hell** ซึ่งเกิดขึ้นเมื่อมีการใช้ Callbacks ซ้อนกันหลายชั้นจนทำให้โค้ดอ่านและจัดการได้ยาก

ตัวอย่างของ Callback Hell:

```
doSomething(function(result) {
    doSomethingElse(result, function(newResult) {
        doAnotherThing(newResult, function(finalResult) {
            console.log("Final result:", finalResult);
        });
    });
});
```

Listing 4.2: ตัวอย่างของ Callback Hell

โค้ดลักษณะนี้เป็นตัวอย่างของ Callback Hell ที่ทำให้โค้ดยากต่อการอ่านและบำรุงรักษา เนื่องจากมีการใช้ฟังก์ชันซ้อนกันหลายชั้น ซึ่งส่งผลให้เกิดโค้ดที่มีลักษณะเป็นโครงสร้างแบบ ”บันได” (pyramid structure)

4.1.2 Promises

Promises ถูกแนะนำเข้ามาใน ES6 เพื่อแก้ไขปัญหาที่เกิดจากการใช้ Callbacks โดย Promises ทำให้โค้ดที่เกี่ยวข้องกับการเขียนโปรแกรมแบบ Asynchronous อ่านง่ายขึ้น และช่วยลดความซับซ้อนของ Callback Hell

Promise คือออบเจกต์ที่แทนค่าของการดำเนินการแบบ Asynchronous ซึ่งอาจจะสำเร็จ (fulfilled) หรือไม่สำเร็จ (rejected) ในอนาคต และมีสองเมธอดหลักคือ `then` และ `catch` เพื่อจัดการกับผลลัพธ์ของการดำเนินการนั้น

ตัวอย่างของ Promises:

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const data = { name: "John", age: 30 };
            resolve(data);
        }, 2000);
    });
}

fetchData()
    .then(data => {
        console.log("Data received:", data);
    })
    .catch(error => {
        console.error("Error:", error);
    });
```

Listing 4.3: ตัวอย่างของ Promises

ในตัวอย่างนี้ ฟังก์ชัน `fetchData` จะคืนค่าเป็น Promise ซึ่งจะถูก `resolve` พร้อมกับข้อมูลหลังจากการดำเนินการเสร็จสิ้น จากนั้นฟังก์ชัน `then` จะถูกเรียกใช้เพื่อจัดการกับข้อมูลที่ได้รับ และฟังก์ชัน `catch` จะถูกเรียกใช้เมื่อเกิดข้อผิดพลาด

Promises ช่วยให้โค้ด Asynchronous อ่านง่ายขึ้นเนื่องจากการใช้ `then` และ `catch` เพื่อจัดการกับผลลัพธ์และข้อผิดพลาด ทำให้โค้ดมีโครงสร้างที่เรียบง่ายขึ้นและจัดการได้ง่าย

4.1.3 Async/Await

Async/Await เป็นคุณสมบัติที่ถูกเพิ่มเข้ามาใน ES2017 ซึ่งช่วยให้การเขียนโปรแกรมแบบ Asynchronous ใน JavaScript ง่ายขึ้น โดยใช้คำสั่ง `async` และ `await` เพื่อจัดการกับ Promises ทำให้โค้ดมีลักษณะคล้ายกับการเขียนโปรแกรมแบบ Synchronous

ตัวอย่างของ Async/Await:

```
async function fetchData() {
    try {
        const data = await new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve({ name: "John", age: 30 });
            }, 2000);
        });
        console.log("Data received:", data);
    } catch (error) {
        console.error("Error:", error);
    }
}

fetchData();
```

Listing 4.4: ตัวอย่างของ Async/Await

ในตัวอย่างนี้ ฟังก์ชัน `fetchData` ถูกประกาศด้วยคำสั่ง `async` และใช้ `await` เพื่อรอให้ Promise สำเร็จก่อนที่จะดำเนินการต่อ คำสั่ง `try/catch` ถูกใช้ในการจัดการข้อผิดพลาดที่อาจเกิดขึ้น

การใช้ Async/Await ช่วยให้โค้ดที่เป็น Asynchronous ดูเรียบง่ายและอ่านง่ายมากขึ้น เนื่องจากสามารถใช้รูปแบบการเขียนโค้ดแบบ Synchronous ได้ ทำให้ไม่จำเป็นต้องใช้ `then` หรือ `catch` ในการจัดการผลลัพธ์หรือข้อผิดพลาด

4.2 การเขียนโปรแกรมแบบ Event-Driven: Event Loop และ Event Emitters ใน Node.js

การเขียนโปรแกรมแบบ Event-Driven เป็นหลักการสำคัญของ Node.js ซึ่งใช้ Event Loop ในการจัดการการดำเนินการแบบ Asynchronous และ Event Emitters ในการจัดการเหตุการณ์ต่าง ๆ

4.2.1 Event Loop ใน Node.js

Event Loop เป็นกลไกที่อยู่เบื้องหลังการทำงานของ Node.js ที่ช่วยให้สามารถจัดการกับการดำเนินการแบบ Asynchronous ได้อย่างมีประสิทธิภาพ โดยใช้เพียงเธรดเดียวในการดำเนินการ

Node.js ใช้ Event Loop ในการจัดการงานที่เป็น I/O และการดำเนินการอื่น ๆ ที่ใช้เวลานาน โดยไม่ทำให้เธรดหลักต้องหยุดชะงัก Event Loop จะตรวจสอบว่างานใดเสร็จสิ้นแล้วและพร้อมที่จะดำเนินการต่อ โดยการใช้ callback หรือการจัดการเหตุการณ์

ขั้นตอนการทำงานของ Event Loop:

1. **Timers**: ตรวจสอบงานที่ถูกกำหนดให้ทำในเวลาที่กำหนด เช่น งานที่ถูกเรียกใช้ด้วย `setTimeout` หรือ `setInterval`
2. **I/O Callbacks**: จัดการกับการดำเนินการ I/O ที่เสร็จสิ้นแล้ว
3. **Idle, Prepare**: ใช้ภายในระบบ Node.js สำหรับการเตรียมการดำเนินการ
4. **Poll**: ดำเนินการ I/O callbacks ที่รออยู่

5. **Check**: จัดการกับงานที่ถูกกำหนดให้ทำใน
   ทันทีด้วย `setImmediate`

6. **Close Callbacks**: ดำเนินการ callback ที่เกี่ยวข้องกับการปิดการเชื่อมต่อ

ตัวอย่างของการทำงานของ Event Loop:

```
console.log("Start");

setTimeout(() => {
    console.log("Timeout");
}, 0);

Promise.resolve().then(() => {
    console.log("Promise");
});

console.log("End");
```

Listing 4.5: ตัวอย่างของการทำงานของ Event Loop

ในตัวอย่างนี้ โค้ดจะทำงานในลำดับดังนี้:

1. พิมพ์ `"Start"`
2. สร้าง `setTimeout` ซึ่งจะถูกจัดคิวใน Event Loop
3. สร้าง Promise ซึ่งจะถูกจัดการทันทีหลังจากโค้ด Synchronous ทั้งหมดเสร็จสิ้น
4. พิมพ์ `"End"`
5. พิมพ์ `"Promise"` (เนื่องจาก Promise อยู่ใน microtask queue ซึ่งจะถูกดำเนินการก่อน tasks อื่น ๆ)
6. สุดท้ายพิมพ์ `"Timeout"`

4.2.2 Event Emitters ใน Node.js

EventEmitter เป็นออบเจกต์ใน Node.js ที่ช่วยให้เราสามารถจัดการกับเหตุการณ์ต่าง ๆ โดยการปล่อยเหตุการณ์ (emit events) และฟังเหตุการณ์ (listen for events) ซึ่งเป็นหัวใจของการเขียนโปรแกรมแบบ Event-Driven ใน Node.js

ตัวอย่างของ Event Emitters:

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();

// Function to listen for the event
eventEmitter.on('start', (message) => {
    console.log(`Event received: ${message}`);
});

// Emit the event
eventEmitter.emit('start', 'Hello, World!');
```

Listing 4.6: Example of Event Emitters

ในตัวอย่างนี้ เราได้สร้างออบเจกต์ EventEmitter และกำหนดฟังก์ชันฟังเหตุการณ์สำหรับเหตุการณ์ `start` เมื่อเหตุการณ์ `start` ถูกปล่อย (emit) ข้อความที่กำหนดจะถูกพิมพ์ออกมา

Event Emitters มักถูกใช้ในการจัดการกับการสื่อสารระหว่างโมดูลต่าง ๆ หรือในการสร้าง API ที่สามารถจัดการกับเหตุการณ์ที่กำหนดเองได้

4.3 การทำงานกับ Timers: setTimeout, setInterval, และ Immediate

Timers เป็นฟังก์ชันพื้นฐานใน JavaScript ที่ช่วยให้เราสามารถกำหนดให้โค้ดถูกเรียกใช้หลังจากเวลาที่กำหนด (`setTimeout`), เรียกใช้ซ้ำๆ ทุกช่วงเวลาที่กำหนด (`setInterval`), หรือเรียกใช้ทันทีหลังจากการดำเนินการในรอบ Event Loop ปัจจุบันเสร็จสิ้น (`setImmediate`)

4.3.1 setTimeout

setTimeout เป็นฟังก์ชันที่ใช้ในการกำหนดให้โค้ดถูกเรียกใช้หลังจากเวลาที่กำหนด ซึ่งเป็นการกำหนดเวลาแบบหน่วยมิลลิวินาที

ตัวอย่างของ setTimeout:

```
1  console.log("Start");
2
3  setTimeout(() => {
4      console.log("This will run after 2 seconds");
5  }, 2000);
6
7  console.log("End");
```

Listing 4.7: ตัวอย่างของ setTimeout

ในตัวอย่างนี้ ข้อความ `"This will run after 2 seconds"` จะถูกพิมพ์ออกมาหลังจาก 2 วินาทีหลังจากที่โค้ดถูกดำเนินการ ข้อความ `"Start"` และ `"End"` จะถูกพิมพ์ออกมาก่อน เนื่องจาก `setTimeout` เป็นการดำเนินการแบบ Asynchronous

4.3.2 setInterval

setInterval เป็นฟังก์ชันที่ใช้ในการกำหนดให้โค้ดถูกเรียกใช้ซ้ำๆ ทุกช่วงเวลาที่กำหนด

ตัวอย่างของ setInterval:

```
1  let count = 0;
2
3  const intervalId = setInterval(() => {
4      count += 1;
5      console.log(`Count: ${count}`);
6
7      if (count === 5) {
8          clearInterval(intervalId);
9          console.log("Interval cleared");
10     }
11 }, 1000);
```

Listing 4.8: ตัวอย่างของ setInterval

ในตัวอย่างนี้ ฟังก์ชันที่กำหนดจะถูกเรียกใช้ทุกๆ 1 วินาที (`1000ms`) และเพิ่มค่า `count` ทีละ 1 หลังจาก `count` ถึง 5 การเรียกใช้ `setInterval` จะถูกหยุดโดยการเรียก `clearInterval`

4.3.3 setImmediate

setImmediate เป็นฟังก์ชันที่ใช้ในการกำหนดให้โค้ดถูกเรียกใช้ทันทีหลังจากการดำเนินการในรอบ Event Loop ปัจจุบันเสร็จสิ้น ซึ่งต่างจาก `setTimeout` ที่กำหนดเวลา `setImmediate` จะถูกดำเนินการทันทีที่ Event Loop มีเวลาว่าง

ตัวอย่างของ setImmediate:

```
1 console.log("Start");
2
3 setImmediate(() => {
4     console.log("This will run immediately after the current Event Loop
          cycle");
5 });
6
7 console.log("End");
```

Listing 4.9: ตัวอย่างของ setImmediate

ในตัวอย่างนี้ ข้อความ `"This will run immediately after the current Event Loop cycle"` จะถูกพิมพ์หลังจาก `"End"` เนื่องจาก `setImmediate` จะถูกดำเนินการหลังจากรอบ Event Loop ปัจจุบันเสร็จสิ้น

การใช้ Timers อย่างมีประสิทธิภาพเป็นสิ่งสำคัญในการจัดการการดำเนินการแบบ Asynchronous ใน JavaScript และ Node.js

4.4 การดำเนินการกับระบบไฟล์ใน Node.js: การอ่าน, เขียน, และจัดการไฟล์แบบ Asynchronous

ระบบไฟล์เป็นหนึ่งในส่วนสำคัญของการพัฒนาแอปพลิเคชัน โดย Node.js มีโมดูล `fs` ที่ช่วยในการดำเนินการกับระบบไฟล์อย่างมีประสิทธิภาพ ซึ่งสามารถดำเนินการแบบ Synchronous หรือ Asynchronous ได้

4.4.1 การอ่านไฟล์แบบ Asynchronous

การอ่านไฟล์ เป็นกระบวนการที่ใช้ในการดึงข้อมูลจากไฟล์ในระบบ โดยการอ่านไฟล์แบบ Asynchronous ช่วยให้โปรแกรมไม่หยุดชะงักขณะรอการอ่านไฟล์เสร็จสิ้น

ตัวอย่างของการอ่านไฟล์แบบ Asynchronous:

```
1 const fs = require('fs');
2
3 fs.readFile('example.txt', 'utf8', (err, data) => {
4     if (err) {
5         console.error('Error reading file:', err);
6         return;
7     }
8     console.log('File content:', data);
9 });
```

Listing 4.10: การอ่านไฟล์แบบ Asynchronous

ในตัวอย่างนี้ เราใช้ `fs.readFile` เพื่ออ่านไฟล์ `example.txt` แบบ Asynchronous โดยใช้ callback เพื่อจัดการกับผลลัพธ์ เมื่อการอ่านไฟล์เสร็จสิ้น ข้อมูลในไฟล์จะถูกส่งผ่านไปยัง callback เพื่อดำเนินการต่อ

4.4.2 การเขียนไฟล์แบบ Asynchronous

การเขียนไฟล์ เป็นกระบวนการที่ใช้ในการบันทึกข้อมูลลงในไฟล์ โดยการเขียนไฟล์แบบ Asynchronous ช่วยให้โปรแกรมสามารถดำเนินการต่อไปได้ในขณะที่ข้อมูลกำลังถูกเขียนลงในไฟล์

ตัวอย่างของการเขียนไฟล์แบบ Asynchronous:

```
const fs = require('fs');

const content = 'This is the content to be written into the file';

fs.writeFile('example.txt', content, 'utf8', (err) => {
    if (err) {
        console.error('Error writing file:', err);
        return;
    }
    console.log('File has been written');
});
```

Listing 4.11: การเขียนไฟล์แบบ Asynchronous

ในตัวอย่างนี้ เราใช้ `fs.writeFile` เพื่อเขียนข้อมูลลงในไฟ

ล์ `example.txt` แบบ Asynchronous ฟังก์ชัน callback จะถูกเรียกใช้หลังจากการเขียนไฟล์เสร็จสิ้น หากเกิดข้อผิดพลาด จะมีการแสดงข้อความข้อผิดพลาด

4.4.3 การจัดการไฟล์อื่น ๆ แบบ Asynchronous

นอกจากการอ่านและเขียนไฟล์แล้ว Node.js ยังมีความสามารถในการดำเนินการอื่น ๆ กับไฟล์ เช่น การลบไฟล์, การเปลี่ยนชื่อไฟล์, และการคัดลอกไฟล์ แบบ Asynchronous

ตัวอย่างของการลบไฟล์แบบ Asynchronous:

```
const fs = require('fs');

fs.unlink('example.txt', (err) => {
    if (err) {
        console.error('Error deleting file:', err);
        return;
    }
    console.log('File has been deleted');
});
```

Listing 4.12: การลบไฟล์แบบ Asynchronous

ในตัวอย่างนี้ เราใช้ `fs.unlink` เพื่อลบไฟล์ `example.txt` แบบ Asynchronous ฟังก์ชัน callback จะถูกเรียกใช้หลังจากการลบไฟล์เสร็จสิ้น

การดำเนินการกับระบบไฟล์แบบ Asynchronous ใน Node.js ช่วยให้โปรแกรมสามารถทำงานได้อย่างลื่นไหลโดยไม่ถูกขัดจังหวะจากการดำเนินการที่ใช้เวลานาน การใช้โมดูล `fs` อย่างมีประสิทธิภาพเป็นสิ่งสำคัญสำหรับการพัฒนาแอปพลิเคชันที่มีประสิทธิภาพ

สรุปท้ายบท

บทนี้ได้เสนอแนวคิดและหลักการของการเขียนโปรแกรมแบบ Asynchronous ใน JavaScript และ Node.js โดยเริ่มจากการอธิบายแนวคิดพื้นฐานของ Callbacks, Promises, และ Async/Await ซึ่งเป็นเครื่องมือสำคัญในการจัดการกับการดำเนินการแบบ Asynchronous จากนั้นได้ทำความเข้าใจการเขียนโปรแกรมแบบ Event-Driven ใน Node.js ซึ่งใช้ Event Loop และ Event Emitters เป็นหลัก

นอกจากนี้ ยังได้อธิบายถึงการทำงานกับ Timers เช่น `setTimeout`, `setInterval`, และ `setImmediate` ซึ่งเป็นฟังก์ชันพื้นฐานในการจัดการกับการดำเนินการแบบ Asynchronous ใน JavaScript และ Node.js ในส่วนสุดท้ายได้

ครอบคลุมถึงการดำเนินการกับระบบไฟล์ใน Node.js อย่าง Asynchronous ซึ่งรวมถึงการอ่าน, เขียน, และจัดการไฟล์อย่างมีประสิทธิภาพ

การเข้าใจและใช้เครื่องมือเหล่านี้อย่างถูกต้องเป็นสิ่งสำคัญสำหรับการพัฒนาแอปพลิเคชันที่มีประสิทธิภาพและสามารถจัดการกับการดำเนินการแบบ Asynchronous ได้อย่างเหมาะสม

คำถามทบทวน:

1. อะไรคือความแตกต่างระหว่าง Callbacks, Promises, และ Async/Await ในการจัดการกับการดำเนินการแบบ Asynchronous?
2. Event Loop ใน Node.js ทำงานอย่างไรในการจัดการกับการดำเนินการแบบ Asynchronous?
3. คุณจะใช้ Timers เช่น `setTimeout`, `setInterval`, และ `setImmediate` ในการจัดการการดำเนินการแบบ Asynchronous ได้อย่างไร?
4. คุณจะดำเนินการกับระบบไฟล์ใน Node.js อย่าง Asynchronous ได้อย่างไร?

การอ่านเพิ่มเติม:

- *Node.js Documentation* Node.js Documentation Team *[6]*
- *JavaScript: The Definitive Guide* Flanagan *[9]*
- *Asynchronous Programming with JavaScript* Burnham *[10]*

บทที่ 5

การแนะนำการพัฒนาเว็บแอปพลิเคชัน

เว็บแอปพลิเคชันเป็นหนึ่งในประเภทของซอฟต์แวร์ที่มีความสำคัญและถูกใช้งานอย่างกว้างขวางในปัจจุบัน ทั้งในด้านธุรกิจและการใช้งานส่วนบุคคล บทนี้จะนำเสนอพื้นฐานของการพัฒนาเว็บแอปพลิเคชัน รวมถึงสถาปัตยกรรมของการพัฒนาเว็บ, การแนะนำเฟรมเวิร์กที่นิยมใช้, การตั้งค่าเว็บเซิร์ฟเวอร์ และการจัดการเส้นทางในเว็บแอปพลิเคชัน

5.1 พื้นฐานของเว็บแอปพลิเคชัน: สถาปัตยกรรม Client-Server และบริการ RESTful

เว็บแอปพลิเคชันถูกสร้างขึ้นมาเพื่อตอบสนองความต้องการของผู้ใช้ที่ต้องการเข้าถึงข้อมูลและบริการผ่านเว็บเบราว์เซอร์ ซึ่งการพัฒนาเว็บแอปพลิเคชันมักใช้สถาปัตยกรรม Client-Server และการสร้างบริการ RESTful เพื่อให้การสื่อสารระหว่างฝั่งลูกข่าย (Client) และฝั่งเซิร์ฟเวอร์ (Server) เป็นไปได้อย่างมีประสิทธิภาพ

5.1.1 สถาปัตยกรรม Client-Server

สถาปัตยกรรม **Client-Server** เป็นรูปแบบการจัดการข้อมูลและการประมวลผลที่แยกฝั่งลูกข่าย (Client) และฝั่งเซิร์ฟเวอร์ (Server) ออกจากกัน โดยที่ Client จะเป็นผู้ร้องขอข้อมูลหรือบริการผ่านโปรโตคอล HTTP และ Server จะเป็นผู้ตอบสนองต่อคำขอเหล่านั้น

Client คือโปรแกรมหรืออุปกรณ์ที่ผู้ใช้สามารถเข้าถึงและใช้งานได้ เช่น เว็บเบราว์เซอร์, แอปพลิเคชันบนมือถือ หรือแอปพลิเคชันเดสก์ท็อป

Server คือโปรแกรมหรือระบบที่ทำงานบนเครื่องแม่ข่าย (host) ซึ่งรับผิดชอบในการจัดการข้อมูล, ประมวลผลคำขอจาก Client, และส่งข้อมูลกลับไปยัง Client ตามคำขอที่ได้รับ

การทำงานของ Client-Server มักจะประกอบด้วยขั้นตอนดังนี้:

1. Client ส่งคำขอ (Request) ไปยัง Server ผ่านโปรโตคอล HTTP โดยระบุข้อมูลหรือบริการที่ต้องการ
2. Server รับคำขอและประมวลผลข้อมูลที่เกี่ยวข้อง เช่น การเข้าถึงฐานข้อมูลหรือการคำนวณทางธุรกิจ
3. Server ส่งคำตอบ (Response) กลับไปยัง Client ซึ่งอาจเป็นข้อมูลในรูปแบบ JSON, HTML, หรือ XML
4. Client แสดงผลข้อมูลที่ได้รับให้ผู้ใช้ดูในรูปแบบของหน้าเว็บหรือ UI ที่ออกแบบมา

การแยกฝั่ง Client และ Server ช่วยให้การพัฒนาเว็บแอปพลิเคชันมีความยืดหยุ่นมากขึ้น เนื่องจากสามารถพัฒนาและปรับปรุงแต่ละฝั่งแยกกันได้โดยไม่ส่งผลกระทบต่ออีกฝั่งหนึ่ง

5.1.2 บริการ RESTful

REST (Representational State Transfer) เป็นรูปแบบการออกแบบสถาปัตยกรรมที่ใช้ในการพัฒนาเว็บแอปพลิเคชันที่มีความยืดหยุ่นและมีประสิทธิภาพ REST ถูกนำมาใช้ในการสร้าง **RESTful services** หรือ **RESTful APIs** ซึ่งเป็นจุดเชื่อมต่อที่ช่วยให้ Client และ Server สามารถสื่อสารและแลกเปลี่ยนข้อมูลกันได้อย่างมีประสิทธิภาพ

หลักการสำคัญของ RESTful services คือ:

- **Stateless**: แต่ละคำขอจาก Client จะต้องมีข้อมูลที่จำเป็นทั้งหมดสำหรับการดำเนินการ Server จะไม่เก็บข้อมูลของสถานะใด ๆ ระหว่างคำขอที่ต่างกัน
- **Uniform Interface**: การสื่อสารระหว่าง Client และ Server จะต้องมีรูปแบบที่สม่ำเสมอ เช่น การใช้ HTTP methods (GET, POST, PUT, DELETE) ในการจัดการกับทรัพยากรต่าง ๆ
- **Client-Server Separation**: การแยกหน้าที่ของ Client และ Server ชัดเจน Client เป็นผู้จัดการการนำเสนอข้อมูล ส่วน Server เป็นผู้จัดการข้อมูลและตรรกะของธุรกิจ
- **Cacheable**: การใช้ caching ช่วยเพิ่มประสิทธิภาพในการรับส่งข้อมูล Client สามารถเก็บข้อมูลที่ได้รับจาก Server ไว้ใช้ซ้ำได้
- **Layered System**: สถาปัตยกรรมของระบบควรเป็นแบบชั้นๆ (layers) เพื่อให้สามารถขยายระบบได้ง่าย เช่น การเพิ่ม proxy servers หรือ load balancers

ตัวอย่างของ RESTful API:
สมมติว่ามี RESTful API สำหรับจัดการข้อมูลผู้ใช้ ซึ่งมีเส้นทาง (endpoint) ต่าง ๆ ดังนี้:

- `GET /users`: รับข้อมูลของผู้ใช้ทั้งหมด
- `GET /users/:id`: รับข้อมูลของผู้ใช้ที่มี ID ระบุ
- `POST /users`: สร้างผู้ใช้ใหม่
- `PUT /users/:id`: อัปเดตข้อมูลของผู้ใช้ที่มี ID ระบุ
- `DELETE /users/:id`: ลบผู้ใช้ที่มี ID ระบุ

ตัวอย่างนี้แสดงให้เห็นถึงวิธีการใช้ HTTP methods ที่ต่างกันเพื่อจัดการกับทรัพยากร (resources) ซึ่งในที่นี้คือข้อมูลของผู้ใช้ การออกแบบ RESTful services ที่ดีจะช่วยให้การพัฒนาเว็บแอปพลิเคชันเป็นไปได้อย่างมีประสิทธิภาพและยืดหยุ่น

5.2 ภาพรวมของเฟรมเวิร์กเว็บ: การแนะนำเฟรมเวิร์กยอดนิยมเช่น Express.js

การพัฒนาเว็บแอปพลิเคชันในปัจจุบันมักใช้ **เฟรมเวิร์กเว็บ** เพื่อช่วยลดความซับซ้อนของกระบวนการพัฒนาและเพิ่มประสิทธิภาพในการทำงาน เฟรมเวิร์กเว็บเป็นชุดของเครื่องมือและไลบรารีที่ช่วยในการสร้างเว็บแอปพลิเคชัน โดยทำให้การจัดการกับเส้นทาง, การจัดการข้อมูล, และการทำงานอื่น ๆ ที่เกี่ยวข้องกับการพัฒนาเว็บเป็นไปได้อย่างง่ายดาย

5.2.1 การแนะนำ Express.js

Express.js เป็นเฟรมเวิร์กเว็บที่มีน้ำหนักเบาและยืดหยุ่น ถูกออกแบบมาเพื่อใช้กับ Node.js ในการพัฒนาเว็บแอปพลิเคชัน Express.js ช่วยให้การสร้างและจัดการเส้นทาง, การจัดการคำขอและคำตอบ, และการใช้งาน middleware เป็นไปได้อย่างง่ายดาย

คุณสมบัติของ Express.js:

- **ความเรียบง่าย**: Express.js มีโครงสร้างที่เรียบง่ายและไม่ซับซ้อน ทำให้การเริ่มต้นใช้งานและการพัฒนาเป็นไปได้อย่างรวดเร็ว
- **ความยืดหยุ่น**: Express.js สามารถปรับแต่งได้ตามความต้องการของผู้ใช้ และสามารถรวมกับไลบรารีอื่น ๆ ได้อย่างง่ายดาย
- **Middleware**: Express.js รองรับการใช้งาน middleware ซึ่งช่วยให้การจัดการคำขอและคำตอบเป็นไปได้อย่างเป็นระบบและมีประสิทธิภาพ
- **Routing**: Express.js มีเครื่องมือสำหรับการจัดการเส้นทางที่หลากหลาย ทำให้การสร้าง API และการจัดการกับเส้นทางเป็นไปได้อย่างง่ายดาย

ตัวอย่างของ Express.js:
สมมติว่าเราต้องการสร้างเว็บเซิร์ฟเวอร์ง่าย ๆ ด้วย Express.js ที่สามารถจัดการกับเส้นทางพื้นฐานได้

```
const express = require('express');
const app = express();

// Handling routes
app.get('/', (req, res) => {
    res.send('Welcome to the homepage!');
});

app.get('/about', (req, res) => {
    res.send('This is the about page.');
});

// Handling routes not found
app.use((req, res) => {
    res.status(404).send('Page not found.');
});

// Set the server to listen on port 3000
const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 5.1: Creating a Server with Express.js

ในตัวอย่างนี้ เราได้สร้างเว็บเซิร์ฟเวอร์ง่าย ๆ ด้วย Express.js ซึ่งสามารถจัดการกับเส้นทาง `"/"` และ `"/about"` ได้ รวมถึงการจัดการกับเส้นทางที่ไม่พบ โดยการใช้ middleware

การใช้ Express.js ช่วยให้การพัฒนาเว็บแอปพลิเคชันเป็นไปได้อย่างรวดเร็วและมีประสิทธิภาพ เนื่องจากสามารถปรับแต่งและขยายเฟรมเวิร์กได้ตามความต้องการของโปรเจกต์

5.3 การตั้งค่าเว็บเซิร์ฟเวอร์: การใช้ Express.js ในการสร้างเว็บเซิร์ฟเวอร์พื้นฐาน

การตั้งค่าเว็บเซิร์ฟเวอร์เป็นขั้นตอนสำคัญในการพัฒนาเว็บแอปพลิเคชัน บทนี้จะนำเสนอขั้นตอนและวิธีการตั้งค่าเว็บเซิร์ฟเวอร์พื้นฐานด้วย Express.js ซึ่งเป็นเฟรมเวิร์กยอดนิยมสำหรับการพัฒนาเว็บใน Node.js

5.3.1 การติดตั้ง Express.js

ก่อนที่จะเริ่มต้นการสร้างเว็บเซิร์ฟเวอร์ด้วย Express.js เราจำเป็นต้องติดตั้ง Express.js ในโปรเจกต์ Node.js ของเรา

ขั้นตอนการติดตั้ง Express.js:

1. สร้างโฟลเดอร์ใหม่สำหรับโปรเจกต์:

```
    mkdir my-express-server
    cd my-express-server
```

2. ใช้คำสั่ง `npm init` เพื่อสร้างไฟล์ `package.json`:

```
    npm init
```

คำสั่งนี้จะช่วยให้คุณสร้างไฟล์ `package.json` สำหรับโปรเจกต์ Node.js ของคุณ

3. ติดตั้ง Express.js โดยใช้คำสั่ง `npm install`:

```
npm install express
```

หลังจากที่ติดตั้ง Express.js เรียบร้อยแล้ว คุณก็สามารถเริ่มต้นการพัฒนาเว็บเซิร์ฟเวอร์ได้

5.3.2 การสร้างเว็บเซิร์ฟเวอร์ด้วย Express.js

การสร้างเว็บเซิร์ฟเวอร์ด้วย Express.js นั้นเรียบง่ายและตรงไปตรงมา คุณสามารถใช้ Express.js ในการจัดการคำขอและตอบกลับคำขอจาก Client ได้อย่างมีประสิทธิภาพ

```
const express = require('express');
const app = express();

// Handling the main route
app.get('/', (req, res) => {
    res.send('Hello, world!');
});

// Handling other routes
app.get('/about', (req, res) => {
    res.send('This is the about page.');
});

// Handling routes not found
app.use((req, res) => {
    res.status(404).send('Page not found.');
});

// Set the server to listen on port 3000
const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 5.2: Creating a Basic Web Server

ในตัวอย่างนี้ เราได้สร้างเว็บเซิร์ฟเวอร์พื้นฐานด้วย Express.js ซึ่งสามารถจัดการกับเส้นทาง `"/"` และ `"/about"` ได้ เมื่อมีการร้องขอจาก Client มายังเส้นทางเหล่านี้ เซิร์ฟเวอร์จะตอบกลับด้วยข้อความที่กำหนดไว้

หากมีการร้องขอไปยังเส้นทางที่ไม่พบในเซิร์ฟเวอร์ Express.js จะตอบกลับด้วยข้อความ `Page not found.` และส่งรหัสสถานะ HTTP 404 เพื่อบอกว่าเส้นทางที่ร้องขอไม่ถูกต้อง

5.3.3 การใช้ Middleware ใน Express.js

Middleware เป็นฟังก์ชันที่ถูกดำเนินการในระหว่างการร้องขอและการตอบกลับ Middleware ใน Express.js ถูกใช้เพื่อจัดการกับการตรวจสอบสิทธิ์, การบันทึกข้อมูล, การจัดการข้อมูล JSON, และอื่น ๆ

```
const express = require('express');
const app = express();

// Middleware to log request details
app.use((req, res, next) => {
    console.log(`Request received: ${req.method} ${req.url}`);
    next(); // Call the next middleware
});

// Handling the main route
app.get('/', (req, res) => {
    res.send('Hello, world!');
});

// Middleware for handling routes not found
app.use((req, res) => {
    res.status(404).send('Page not found.');
});

// Set the server to listen on port 3000
const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 5.3: Example of Using Middleware

ในตัวอย่างนี้ เราได้เพิ่ม Middleware ที่จะบันทึกข้อมูลของการร้องขอทุกครั้งที่เซิร์ฟเวอร์ได้รับ เมื่อได้รับการร้องขอแล้ว Middleware จะพิมพ์ข้อความแสดงคำขอที่ได้รับและเรียกใช้ Middleware ถัดไป (โดยใช้ `next()`)

การใช้ Middleware ใน Express.js ช่วยให้คุณสามารถจัดการกับการดำเนินการที่ซับซ้อนในระหว่างการร้องขอได้อย่างเป็นระบบและมีประสิทธิภาพ

5.4 การจัดการเส้นทางในเว็บแอปพลิเคชัน: การสร้างและจัดการเส้นทางต่าง ๆ

Routing หรือการจัดการเส้นทาง เป็นกระบวนการที่เว็บเซิร์ฟเวอร์ใช้ในการกำหนดวิธีการตอบสนองต่อคำร้องขอที่มาจาก URL ที่แตกต่างกัน การจัดการเส้นทางในเว็บแอปพลิเคชันมีความสำคัญในการสร้าง API และการจัดการกับเนื้อหาต่าง ๆ ในแอปพลิเคชัน

5.4.1 การสร้างเส้นทางใน Express.js

Express.js ช่วยให้การจัดการเส้นทางในเว็บแอปพลิเคชันเป็นไปได้อย่างง่ายดายและยืดหยุ่น คุณสามารถสร้างเส้นทางต่าง ๆ เพื่อจัดการกับคำร้องขอที่มาจาก URL ที่แตกต่างกัน

```
const express = require('express');
const app = express();

// Handling the main route
app.get('/', (req, res) => {
    res.send('Welcome to the homepage!');
});

// Handling routes related to user information
app.get('/users', (req, res) => {
    res.send('Here is the list of users.');
});

app.get('/users/:id', (req, res) => {
    const userId = req.params.id;
    res.send(`Details of user ${userId}`);
});

// Handling routes not found
app.use((req, res) => {
    res.status(404).send('Page not found.');
});

// Set the server to listen on port 3000
const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 5.4: Creating Routes in Express.js

ในตัวอย่างนี้ เราได้สร้างเส้นทางหลายเส้นทางใน Express.js:

- เส้นทาง `"/"` สำหรับหน้าแรก
- เส้นทาง `"/users"` สำหรับรายการผู้ใช้
- เส้นทาง `"/users/:id"` สำหรับรายละเอียดของผู้ใช้ที่มี ID ระบุ

เมื่อมีการร้องขอไปยัง URL เหล่านี้ เซิร์ฟเวอร์จะตอบกลับด้วยข้อความที่กำหนดไว้ การใช้ `req.params` ช่วยให้เราสามารถเข้าถึงพารามิเตอร์ที่ถูกส่งผ่าน URL ได้ เช่น `userId` ในเส้นทาง `"/users/:id"`

5.4.2 การใช้ Router ใน Express.js

Router ใน Express.js เป็นเครื่องมือที่ช่วยในการจัดการเส้นทางให้มีโครงสร้างที่ดีขึ้น โดยสามารถแยกเส้นทางที่เกี่ยวข้องกันออกเป็นกลุ่มและจัดการแยกกันได้

```
const express = require('express');
const app = express();
const router = express.Router();

// Handling routes related to users
router.get('/', (req, res) => {
    res.send('User homepage');
});

router.get('/:id', (req, res) => {
    const userId = req.params.id;
    res.send(`Details of user ${userId}`);
});

// Using the Router for the `/users` path
app.use('/users', router);

// Handling routes not found
app.use((req, res) => {
    res.status(404).send('Page not found.');
});

// Set the server to listen on port 3000
const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 5.5: Using Router in Express.js

ในตัวอย่างนี้ เราได้สร้าง **Router** เพื่อจัดการกับเส้นทางที่เกี่ยวข้องกับผู้ใช้ทั้งหมด จากนั้นเรานำ **Router** นี้มาใช้กับเส้นทาง **"/users"** ในแอปพลิเคชัน การใช้ **Router** ช่วยให้โค้ดมีโครงสร้างที่เป็นระเบียบมากขึ้นและง่ายต่อการบำรุงรักษา

การจัดการเส้นทางในเว็บแอปพลิเคชันด้วย Express.js ช่วยให้การพัฒนาเป็นไปได้อย่างง่ายดายและยืดหยุ่น คุณสามารถสร้างเส้นทางต่าง ๆ และจัดการกับคำร้องขอจาก Client ได้อย่างมีประสิทธิภาพ

สรุปท้ายบท

บทนี้ได้เสนอแนวคิดพื้นฐานในการพัฒนาเว็บแอปพลิเคชัน เริ่มต้นจากการทำความเข้าใจสถาปัตยกรรม Client-Server และบริการ RESTful ที่เป็นพื้นฐานในการพัฒนาเว็บแอปพลิเคชันสมัยใหม่ จากนั้นได้แนะนำ Express.js ซึ่งเป็นเฟรมเวิร์กยอดนิยมสำหรับการพัฒนาเว็บใน Node.js และวิธีการตั้งค่าเว็บเซิร์ฟเวอร์ด้วย Express.js

นอกจากนี้ ยังได้ครอบคลุมถึงการจัดการเส้นทางในเว็บแอปพลิเคชันด้วย Express.js ซึ่งเป็นกระบวนการสำคัญในการสร้าง API และการจัดการเนื้อหาต่าง ๆ ในแอปพลิเคชัน

การเข้าใจและใช้เครื่องมือเหล่านี้อย่างถูกต้องเป็นสิ่งสำคัญสำหรับการพัฒนาเว็บแอปพลิเคชันที่มีประสิทธิภาพและสามารถจัดการกับการร้องขอจาก Client ได้อย่างเหมาะสม

คำถามทบทวน:

1. สถาปัตยกรรม Client-Server คืออะไร และมีบทบาทอย่างไรในเว็บแอปพลิเคชัน?
2. บริการ RESTful มีหลักการอย่างไรในการออกแบบ API สำหรับเว็บแอปพลิเคชัน?
3. คุณจะสร้างเว็บเซิร์ฟเวอร์พื้นฐานด้วย Express.js ได้อย่างไร?
4. การจัดการเส้นทางใน Express.js มีความสำคัญอย่างไรในการพัฒนาเว็บแอปพลิเคชัน?

การอ่านเพิ่มเติม:

- *Express.js Guide* *Express.js Documentation Team [7]*
- *RESTful Web APIs by Leonard Richardson and Mike Amundsen* *Richardson, Amundsen, and Ruby [11]*
- *JavaScript and jQuery: Interactive Front-End Web Development by Jon Duckett* *Duckett [12]*

บทที่ 6

การพัฒนา API ด้วย Node.js

ในยุคสมัยที่เว็บแอปพลิเคชันและแอปพลิเคชันบนมือถือมีการเติบโตอย่างรวดเร็ว **API** (Application Programming Interface) ได้กลายเป็นส่วนสำคัญของการพัฒนาแอปพลิเคชันสมัยใหม่ โดยเฉพาะอย่างยิ่ง **RESTful APIs** ที่มีบทบาทสำคัญในการทำให้ระบบสามารถทำงานร่วมกันได้อย่างราบรื่นและมีประสิทธิภาพ บทนี้จะนำเสนอแนวคิดพื้นฐานของการพัฒนา API ด้วย Node.js รวมถึงการสร้าง API ที่สามารถทำการเพิ่ม, อ่าน, แก้ไข, และลบทรัพยากร (resources) การใช้ middleware ใน Express.js สำหรับการจัดการคำขอ และการจัดการข้อผิดพลาดใน APIs

6.1 การทำความเข้าใจ API: RESTful APIs และความสำคัญในแอปพลิเคชันสมัยใหม่

API ย่อมาจาก Application Programming Interface เป็นชุดของฟังก์ชันและโปรโตคอลที่ช่วยให้นักพัฒนาสามารถสร้างแอปพลิเคชันที่สื่อสารกันได้ API เป็นเสมือนสัญญาณการติดต่อระหว่างส่วนต่าง ๆ ของซอฟต์แวร์ ที่ทำให้การแลกเปลี่ยนข้อมูลและการดำเนินการต่าง ๆ เป็นไปได้อย่างราบรื่น

6.1.1 RESTful APIs

REST (Representational State Transfer) เป็นสถาปัตยกรรมที่ถูกออกแบบมาเพื่อการสร้าง API ที่เรียกว่า **RESTful APIs** ซึ่งใช้โปรโตคอล HTTP ในการสื่อสารและการจัดการทรัพยากร

หลักการสำคัญของ RESTful APIs คือ:

- **Stateless**: แต่ละคำขอจากลูกข่าย (client) จะต้องมีข้อมูลที่จำเป็นทั้งหมดสำหรับการดำเนินการ เซิร์ฟเวอร์จะไม่เก็บข้อมูลของสถานะใด ๆ ระหว่างคำขอที่ต่างกัน
- **Uniform Interface**: การสื่อสารระหว่างลูกข่ายและเซิร์ฟเวอร์จะต้องมีรูปแบบที่สม่ำเสมอ เช่น การใช้ HTTP methods (GET, POST, PUT, DELETE) ในการจัดการกับทรัพยากรต่าง ๆ
- **Client-Server Separation**: การแยกหน้าที่ของลูกข่ายและเซิร์ฟเวอร์ชัดเจน ลูกข่ายเป็นผู้จัดการการนำเสนอข้อมูล ส่วนเซิร์ฟเวอร์เป็นผู้จัดการข้อมูลและตรรกะของธุรกิจ
- **Cacheable**: การใช้ caching ช่วยเพิ่มประสิทธิภาพในการรับส่งข้อมูล ลูกข่ายสามารถเก็บข้อมูลที่ได้รับจากเซิร์ฟเวอร์ไว้ใช้ซ้ำได้
- **Layered System**: สถาปัตยกรรมของระบบควรเป็นแบบชั้นๆ (layers) เพื่อให้สามารถขยายระบบได้ง่าย เช่น การเพิ่ม proxy servers หรือ load balancers

การพัฒนา RESTful APIs ช่วยให้การพัฒนาแอปพลิเคชันเป็นไปได้อย่างมีประสิทธิภาพและยืดหยุ่น นักพัฒนาสามารถสร้าง API ที่รองรับการทำงานร่วมกันระหว่างแอปพลิเคชันต่าง ๆ ได้ง่ายขึ้น

6.2 การสร้าง RESTful APIs: การสร้าง, อ่าน, แก้ไข, และลบทรัพยากร

การพัฒนา RESTful APIs ด้วย Node.js และ Express.js เป็นกระบวนการที่สามารถทำได้อย่างง่ายดายและมีประสิทธิภาพ ในส่วนนี้ เราจะได้เรียนรู้วิธีการสร้าง API ที่สามารถทำการเพิ่ม, อ่าน, แก้ไข, และลบทรัพยากรต่าง ๆ

6.2.1 การตั้งค่าโปรเจกต์และการติดตั้ง Express.js

ก่อนที่เราจะเริ่มต้นการสร้าง API เราจำเป็นต้องตั้งค่าโปรเจกต์และติดตั้ง Express.js

ขั้นตอนการตั้งค่าโปรเจกต์และติดตั้ง Express.js:

1. สร้างโฟลเดอร์ใหม่สำหรับโปรเจกต์:

```
mkdir my-rest-api
cd my-rest-api
```

Listing 6.1: การสร้างโฟลเดอร์สำหรับโปรเจกต์

2. ใช้คำสั่ง `npm init` เพื่อสร้างไฟล์ `package.json`:

```
npm init
```

Listing 6.2: การใช้คำสั่ง npm init

3. ติดตั้ง Express.js และไลบรารีอื่น ๆ ที่จำเป็น:

```
npm install express body-parser mongoose
```

Listing 6.3: การติดตั้ง Express.js และไลบรารีอื่น ๆ

`express` เป็นเฟรมเวิร์กที่ใช้ในการสร้างเว็บแอปพลิเคชันและ API `body-parser` เป็น middleware ที่ใช้ในการแยกข้อมูลจากคำขอ `mongoose` เป็นไลบรารีที่ใช้ในการเชื่อมต่อและจัดการกับฐานข้อมูล MongoDB

6.2.2 การสร้าง API สำหรับการจัดการทรัพยากร

ในส่วนนี้ เราจะสร้าง API ที่สามารถทำการเพิ่ม, อ่าน, แก้ไข, และลบข้อมูลของผู้ใช้ (users)

การตั้งค่า Express.js และการสร้างเส้นทาง:

```
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const app = express();

app.use(bodyParser.json());

// Connecting to the MongoDB database
mongoose.connect('mongodb://localhost:27017/myrestapi', { useNewUrlParser: true, useUnifiedTopology: true });

// Defining the User model
const User = mongoose.model('User', {
    name: String,
    email: String,
    age: Number
});

// Creating a new user (Create)
app.post('/users', async (req, res) => {
    try {
        const user = new User(req.body);
        await user.save();
        res.status(201).send(user);
    } catch (error) {
        res.status(400).send(error);
    }
});

// Reading all users (Read)
app.get('/users', async (req, res) => {
    try {
        const users = await User.find({});
        res.status(200).send(users);
    } catch (error) {
        res.status(500).send(error);
    }
});

// Reading a user by ID (Read)
app.get('/users/:id', async (req, res) => {
    try {
        const user = await User.findById(req.params.id);
        if (!user) {
            return res.status(404).send();
        }
        res.status(200).send(user);
    } catch (error) {
        res.status(500).send(error);
    }
});

// Updating a user (Update)
app.put('/users/:id', async (req, res) => {
    try {
        const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true, runValidators: true });
        if (!user) {
            return res.status(404).send();
        }
        res.status(200).send(user);
    } catch (error) {
        res.status(400).send(error);
    }
});
```

ในตัวอย่างนี้ เราได้สร้าง API ที่สามารถทำการเพิ่ม, อ่าน, แก้ไข, และลบข้อมูลของผู้ใช้ โดยใช้ Express.js และ MongoDB

การเพิ่มผู้ใช้ใหม่ (Create):

- เส้นทาง `POST /users` จะรับข้อมูลผู้ใช้จากคำขอและสร้างผู้ใช้ใหม่ในฐานข้อมูล

การอ่านข้อมูลผู้ใช้ทั้งหมด (Read):

- เส้นทาง `GET /users` จะอ่านข้อมูลผู้ใช้ทั้งหมดจากฐานข้อมูลและส่งกลับไปยังลูกข่าย

การอ่านข้อมูลผู้ใช้โดย ID (Read):

- เส้นทาง `GET /users/:id` จะอ่านข้อมูลผู้ใช้ที่มี ID ระบุจากฐานข้อมูลและส่งกลับไปยังลูกข่าย

การอัปเดตข้อมูลผู้ใช้ (Update):

- เส้นทาง `PUT /users/:id` จะอัปเดตข้อมูลผู้ใช้ที่มี ID ระบุในฐานข้อมูล

การลบผู้ใช้ (Delete):

- เส้นทาง `DELETE /users/:id` จะลบผู้ใช้ที่มี ID ระบุจากฐานข้อมูล

6.3 Middleware ใน Express.js: การเขียนและการใช้ Middleware สำหรับการจัดการคำขอ

Middleware ใน Express.js เป็นฟังก์ชันที่ถูกดำเนินการในระหว่างการร้องขอและการตอบกลับ Middleware สามารถใช้ในการจัดการกับการตรวจสอบสิทธิ์, การบันทึกข้อมูล, การจัดการข้อมูล JSON, และอื่น ๆ

6.3.1 การเขียน Middleware ใน Express.js

การเขียน Middleware ใน Express.js เป็นเรื่องง่าย โดยใช้ฟังก์ชันที่รับพารามิเตอร์ **`req`** (request), **`res`** (response), และ **`next`** ซึ่ง **`next`** เป็นฟังก์ชันที่ใช้ในการเรียก Middleware ถัดไปในสายโซ่ (chain) ของ Middleware

ตัวอย่างของการเขียน Middleware:

```
const express = require('express');
const app = express();

// Middleware to log request details
app.use((req, res, next) => {
    console.log(`Request received: ${req.method} ${req.url}`);
    next(); // Call the next middleware
});

// Handling the main route
app.get('/', (req, res) => {
    res.send('Hello, world!');
});

// Set the server to listen on port 3000
const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 6.5: Writing Middleware in Express.js

ในตัวอย่างนี้ เราได้สร้าง Middleware ที่จะบันทึกข้อมูลการร้องขอทุกครั้งที่เซิร์ฟเวอร์ได้รับ โดยจะพิมพ์ข้อความแสดงคำขอที่ได้รับและเรียกใช้ Middleware ถัดไป

6.3.2 การใช้ Middleware ที่มีอยู่ใน Express.js

Express.js มี Middleware ที่มีอยู่แล้วจำนวนมากที่สามารถนำมาใช้ได้อย่างง่ายดาย

ตัวอย่างของการใช้ Middleware `body-parser`:

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

// Use body-parser to handle JSON data
app.use(bodyParser.json());

// Handling the main route
app.post('/data', (req, res) => {
    const data = req.body;
    res.send(`Data received: ${JSON.stringify(data)}`);
});

// Set the server to listen on port 3000
const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 6.6: Using the body-parser Middleware in Express.js

ในตัวอย่างนี้ เราได้ใช้ `body-parser` เพื่อจัดการกับข้อมูล JSON ที่ถูกส่งมาจากลูกข่ายในคำขอ POST โดย `body-parser` จะช่วยแยกข้อมูล JSON ออกจากคำขอและทำให้พร้อมใช้งานใน `req.body`

6.4 การจัดการข้อผิดพลาดใน API: การนำข้อผิดพลาดและการตรวจสอบข้อมูลมาใช้งาน

การจัดการข้อผิดพลาดเป็นส่วนสำคัญของการพัฒนา API ที่มีคุณภาพ การตรวจสอบข้อมูลและการจัดการข้อผิดพลาดอย่างถูกต้องช่วยให้ API ของคุณมีความเสถียรและสามารถจัดการกับสถานการณ์ที่ไม่คาดคิดได้อย่างมีประสิทธิภาพ

6.4.1 การจัดการข้อผิดพลาดใน Express.js

Express.js มีวิธีการจัดการข้อผิดพลาดที่ง่ายและมีประสิทธิภาพ โดยใช้ Middleware สำหรับการจัดการข้อผิดพลาด

ตัวอย่างของการจัดการข้อผิดพลาด:

```
const express = require('express');
const app = express();

// Route that may cause an error
app.get('/error', (req, res, next) => {
    try {
        throw new Error('Something went wrong!');
    } catch (error) {
        next(error); // Pass the error to the error-handling middleware
    }
});

// Error-handling middleware
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Something broke!');
});

// Set the server to listen on port 3000
const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 6.7: Error Handling in Express.js

ในตัวอย่างนี้ เราได้สร้าง Middleware สำหรับการจัดการข้อผิดพลาด ซึ่งจะถูกเรียกใช้เมื่อเกิดข้อผิดพลาดในระหว่างการดำเนินการ โดย Middleware นี้จะพิมพ์ข้อความข้อผิดพลาดลงในคอนโซลและส่งกลับไปยังลูกข่ายด้วยรหัสสถานะ HTTP 500

6.4.2 การตรวจสอบข้อมูลใน API

การตรวจสอบข้อมูลที่ถูกส่งเข้ามาใน API เป็นสิ่งสำคัญในการป้องกันข้อผิดพลาดและการโจมตีที่อาจเกิดขึ้น การตรวจสอบข้อมูลช่วยให้แน่ใจว่าข้อมูลที่ถูกส่งเข้ามานั้นถูกต้องและปลอดภัย

ตัวอย่างของการตรวจสอบข้อมูล:

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

app.use(bodyParser.json());

const validateUserData = (req, res, next) => {
    const { name, email, age } = req.body;
    if (!name || typeof name !== 'string') {
        return res.status(400).send('Invalid name');
    }
    if (!email || typeof email !== 'string') {
        return res.status(400).send('Invalid email');
    }
    if (!age || typeof age !== 'number') {
        return res.status(400).send('Invalid age');
    }
    next();
};

// Use middleware to validate data before creating a new user
app.post('/users', validateUserData, (req, res) => {
    const user = req.body;
    // Create a new user in the database (hypothetical)
    res.status(201).send(`User created: ${JSON.stringify(user)}`);
});

// Set the server to listen on port 3000
const port = 3000;
app.listen(port, () => {
    console.log(`Server running at http://localhost:${port}/`);
});
```

Listing 6.8: Data Validation in an API with Express.js

ในตัวอย่างนี้ เราได้สร้าง Middleware **validateUserData** เพื่อทำการตรวจสอบข้อมูลที่ถูกส่งเข้ามาในคำขอ POST หากข้อมูลไม่ถูกต้อง Middleware จะส่งข้อผิดพลาดกลับไปยังลูกข่ายด้วยรหัสสถานะ HTTP 400

การตรวจสอบข้อมูลใน API ช่วยให้ระบบของคุณปลอดภัยและเชื่อถือได้ โดยป้องกันไม่ให้ข้อมูลที่ไม่ถูกต้องหรือเป็นอันตรายถูกส่งเข้ามาในระบบ

สรุปท้ายบท

บทนี้ได้เสนอแนวคิดและขั้นตอนพื้นฐานในการพัฒนา API ด้วย Node.js และ Express.js โดยเริ่มจากการทำความเข้าใจเกี่ยวกับ RESTful API และบทบาทสำคัญของมันในแอปพลิเคชันสมัยใหม่ จากนั้นได้แนะนำวิธีการสร้าง API ที่สามารถทำการเพิ่ม, อ่าน, แก้ไข, และลบทรัพยากรต่าง ๆ ในฐานข้อมูล

นอกจากนี้ ยังได้อธิบายถึงการใช้ Middleware ใน Express.js เพื่อจัดการกับการร้องขอและการตอบกลับ รวมถึงการจัดการข้อผิดพลาดใน API ซึ่งเป็นส่วนสำคัญในการพัฒนา API ที่มีคุณภาพ การจัดการข้อผิดพลาดและการตรวจสอบข้อมูลอย่างถูกต้องช่วยให้ API ของคุณมีความเสถียรและปลอดภัยมากขึ้น

การเข้าใจและใช้เครื่องมือเหล่านี้อย่างถูกต้องเป็นสิ่งสำคัญสำหรับการพัฒนา API ที่มีประสิทธิภาพและสามารถจัดการกับการร้องขอจากลูกข่ายได้อย่างเหมาะสม

คำถามทบทวน:

1. RESTful APIs คืออะไร และมีบทบาทอย่างไรในแอปพลิเคชันสมัยใหม่?

2. คุณจะสร้าง API สำหรับการเพิ่ม, อ่าน, แก้ไข, และลบทรัพยากรได้อย่างไรใน Node.js และ Express.js?
3. Middleware ใน Express.js มีบทบาทอย่างไรในการจัดการคำขอและการตอบกลับ?
4. คุณจะจัดการข้อผิดพลาดและตรวจสอบข้อมูลใน API อย่างไรเพื่อให้ API มีคุณภาพและปลอดภัย?

การอ่านเพิ่มเติม:

- *Node.js Documentation* *Node.js Documentation Team [6]*
- *Express.js Guide* *Express.js Documentation Team [7]*
- *Building APIs with Node.js by Caio Ribeiro Pereira* *Pereira [13]*

บทที่ 7

การทำงานกับฐานข้อมูล SQL - แนะนำการใช้งานฐานข้อมูล SQL

ฐานข้อมูล SQL (Structured Query Language) เป็นเครื่องมือที่สำคัญในการจัดการข้อมูลในแอปพลิเคชันซอฟต์แวร์ โดยเฉพาะแอปพลิเคชันที่มีการเก็บข้อมูลจำนวนมาก ฐานข้อมูล SQL มีความสามารถในการจัดการกับข้อมูลในรูปแบบที่มีโครงสร้างชัดเจน (Structured Data) ซึ่งเป็นที่นิยมในธุรกิจและองค์กรต่างๆ ทั่วโลก บทนี้จะนำเสนอภาพรวมของฐานข้อมูลเชิงสัมพันธ์ (Relational Databases) และการใช้งาน SQL เบื้องต้น การบูรณาการฐานข้อมูล SQL เข้ากับ Node.js ด้วยไลบรารีต่าง ๆ การดำเนินการ CRUD (Create, Read, Update, Delete) ในฐานข้อมูล SQL และการจัดการการย้ายและการกำหนดโครงสร้างฐานข้อมูล (Database Migrations)

7.1 ภาพรวมของฐานข้อมูลเชิงสัมพันธ์และ SQL พื้นฐาน

ฐานข้อมูลเชิงสัมพันธ์ (Relational Databases) เป็นรูปแบบฐานข้อมูลที่ใช้โครงสร้างตาราง (Table) ในการจัดเก็บข้อมูล แต่ละตารางจะมีความสัมพันธ์กันผ่านคีย์ (Keys) ที่ใช้ในการเชื่อมโยงข้อมูล ตารางในฐานข้อมูลเชิงสัมพันธ์จะประกอบด้วยคอลัมน์ (Columns) และแถว (Rows) ซึ่งคอลัมน์จะระบุประเภทของข้อมูล และแถวจะเป็นการบันทึกข้อมูลแต่ละรายการ

SQL (Structured Query Language) เป็นภาษาที่ใช้ในการจัดการและจัดการข้อมูลในฐานข้อมูลเชิงสัมพันธ์ SQL ถูกใช้ในการดำเนินการต่าง ๆ เช่น การดึงข้อมูล (SELECT), การเพิ่มข้อมูล (INSERT), การอัปเดตข้อมูล (UPDATE), และการลบข้อมูล (DELETE) SQL เป็นภาษามาตรฐานที่ใช้ในระบบฐานข้อมูลเชิงสัมพันธ์ที่มีการใช้งานกันอย่างแพร่หลาย เช่น MySQL, PostgreSQL, Oracle, และ Microsoft SQL Server

7.1.1 คีย์หลักและคีย์ต่างประเทศ (Primary and Foreign Keys)

คีย์หลัก (Primary Key) เป็นคอลัมน์หรือชุดของคอลัมน์ที่ใช้ในการระบุแต่ละแถวในตารางให้มีเอกลักษณ์ คีย์หลักจะต้องไม่ซ้ำกันและไม่มีค่าที่เป็น null ซึ่งหมายความว่าแต่ละแถวในตารางจะต้องมีค่าที่แตกต่างกันในคอลัมน์คีย์หลัก

คีย์ต่างประเทศ (Foreign Key) เป็นคอลัมน์ที่ใช้ในการเชื่อมโยงข้อมูลระหว่างสองตาราง คีย์ต่างประเทศจะอ้างอิงถึงคีย์หลักในตารางอื่นเพื่อสร้างความสัมพันธ์ระหว่างข้อมูล เช่น การเชื่อมโยงข้อมูลระหว่างตารางคำสั่งซื้อ (Orders) และตารางลูกค้า (Customers) โดยที่แต่ละคำสั่งซื้อจะอ้างอิงถึงลูกค้าที่ทำการสั่งซื้อ

ตัวอย่างการสร้างตารางด้วยคีย์หลักและคีย์ต่างประเทศ:

```sql
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Email VARCHAR(255) NOT NULL
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Listing 7.1: ตัวอย่างการสร้างตารางด้วยคีย์หลักและคีย์ต่างประเทศ

ในตัวอย่างนี้ ตาราง ‘Customers‘ มีคีย์หลักคือ ‘CustomerID‘ ซึ่งใช้ระบุลูกค้าแต่ละราย และตาราง ‘Orders‘ มีคีย์หลักคือ ‘OrderID‘ และคีย์ต่างประเทศคือ ‘CustomerID‘ ที่อ้างอิงถึง ‘CustomerID‘ ในตาราง ‘Customers‘ เพื่อระบุว่าคำสั่งซื้อแต่ละรายการมาจากลูกค้ารายใด

7.1.2 การดำเนินการพื้นฐานใน SQL

SELECT: คำสั่ง ‘SELECT‘ ใช้ในการดึงข้อมูลจากตารางในฐานข้อมูล

```sql
SELECT * FROM Customers;
```

Listing 7.2: ตัวอย่างของการใช้คำสั่ง SELECT

คำสั่งนี้จะดึงข้อมูลทั้งหมดจากตาราง ‘Customers‘

INSERT: คำสั่ง ‘INSERT‘ ใช้ในการเพิ่มข้อมูลใหม่ลงในตาราง

```sql
INSERT INTO Customers (CustomerID, Name, Email)
VALUES (1, 'John Doe', 'john.doe@example.com');
```

Listing 7.3: ตัวอย่างของการใช้คำสั่ง INSERT

คำสั่งนี้จะเพิ่มลูกค้าใหม่ลงในตาราง ‘Customers‘

UPDATE: คำสั่ง ‘UPDATE‘ ใช้ในการอัปเดตข้อมูลในตาราง

```sql
UPDATE Customers
SET Email = 'john.newemail@example.com'
WHERE CustomerID = 1;
```

Listing 7.4: ตัวอย่างของการใช้คำสั่ง UPDATE

คำสั่งนี้จะอัปเดตอีเมลของลูกค้าที่มี ‘CustomerID‘ เท่ากับ 1

DELETE: คำสั่ง ‘DELETE‘ ใช้ในการลบข้อมูลจากตาราง

```sql
DELETE FROM Customers
WHERE CustomerID = 1;
```

Listing 7.5: ตัวอย่างของการใช้คำสั่ง DELETE

คำสั่งนี้จะลบลูกค้าที่มี ‘CustomerID‘ เท่ากับ 1 ออกจากตาราง ‘Customers‘

7.2 การบูรณาการฐานข้อมูล SQL เข้ากับ Node.js: การใช้ไลบรารีอย่าง Sequelize หรือ Knex

การบูรณาการฐานข้อมูล SQL เข้ากับ Node.js เป็นขั้นตอนสำคัญในการพัฒนาแอปพลิเคชันที่มีความซับซ้อน Node.js มีไลบรารีหลายตัวที่ช่วยให้การเชื่อมต่อและจัดการกับฐานข้อมูล SQL เป็นไปได้อย่างง่ายดาย เช่น **Sequelize** และ **Knex**

7.2.1 Sequelize

Sequelize เป็น Object-Relational Mapping (ORM) ไลบรารีสำหรับ Node.js ที่ช่วยให้นักพัฒนาสามารถทำงานกับฐานข้อมูลเชิงสัมพันธ์ได้อย่างสะดวกสบาย โดยการแปลงข้อมูลจากตารางในฐานข้อมูลให้เป็นออบเจกต์ใน JavaScript และการจัดการกับข้อมูลเหล่านี้เหมือนกับการจัดการกับออบเจกต์ในโค้ด

การติดตั้ง Sequelize:

```
npm install sequelize sequelize-cli pg pg-hstore
```

Listing 7.6: การติดตั้ง Sequelize

ในตัวอย่างนี้ เราได้ติดตั้ง ‘sequelize‘, ‘sequelize-cli‘ (เครื่องมือ Command Line สำหรับ Sequelize), ‘pg‘ (ไลบรารีสำหรับ PostgreSQL), และ ‘pg-hstore‘ (การจัดการกับข้อมูล JSON ใน PostgreSQL)

การตั้งค่า Sequelize:

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize('database', 'username', 'password', {
    host: 'localhost',
    dialect: 'postgres' // or 'mysql', 'sqlite', 'mariadb', 'mssql'
});

// Defining the User model
const User = sequelize.define('User', {
    name: {
        type: DataTypes.STRING,
        allowNull: false
    },
    email: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: true
    },
    age: {
        type: DataTypes.INTEGER,
        allowNull: true
    }
});

// Connecting to the database and creating tables
sequelize.sync().then(() => {
    console.log('Database & tables created!');
});
```

Listing 7.7: Setting Up Sequelize

ในตัวอย่างนี้ เราได้ใช้ Sequelize ในการสร้างโมเดล ‘User‘ ซึ่งมีฟิลด์ ‘name‘, ‘email‘, และ ‘age‘ และทำการซิงโครไนซ์โมเดลกับฐานข้อมูลเพื่อสร้างตาราง

7.2.2 Knex

Knex เป็น SQL query builder สำหรับ Node.js ที่ให้ความยืดหยุ่นในการเขียนคำสั่ง SQL โดยการใช้ JavaScript Knex สามารถเชื่อมต่อกับฐานข้อมูล SQL หลายประเภท เช่น PostgreSQL, MySQL, และ SQLite

การติดตั้ง Knex:

```
npm install knex pg
```

Listing 7.8: การติดตั้ง Knex

ในตัวอย่างนี้ เราได้ติดตั้ง ‘knex‘ และ ‘pg‘ สำหรับการเชื่อมต่อกับ PostgreSQL

การตั้งค่า Knex:

```
const knex = require('knex')({
    client: 'pg',
    connection: {
        host: 'localhost',
        user: 'username',
        password: 'password',
        database: 'mydb'
    }
});

//
knex.schema.createTable('users', (table) => {
    table.increments('id').primary();
    table.string('name');
    table.string('email').unique();
    table.integer('age');
}).then(() => {
    console.log('Table created');
}).catch((err) => {
    console.error('Error creating table:', err);
});
```

Listing 7.9: การตั้งค่า Knex

ในตัวอย่างนี้ เราได้ใช้ Knex ในการสร้างตาราง ‘users‘ ในฐานข้อมูล PostgreSQL โดยใช้ JavaScript

7.3 การดำเนินการ CRUD ใน SQL: การสร้าง, อ่าน, แก้ไข, และลบข้อมูล

CRUD (Create, Read, Update, Delete) เป็นกระบวนการพื้นฐานในการจัดการกับข้อมูลในฐานข้อมูล SQL การดำเนินการ CRUD ช่วยให้นักพัฒนาสามารถเพิ่ม, อ่าน, แก้ไข, และลบข้อมูลในฐานข้อมูลได้อย่างมีประสิทธิภาพ

7.3.1 การสร้างข้อมูล (Create)

การสร้างข้อมูลในฐานข้อมูล SQL สามารถทำได้โดยการใช้คำสั่ง ‘INSERT INTO‘ เพื่อเพิ่มข้อมูลใหม่ลงในตาราง

ตัวอย่างของการสร้างข้อมูล:

```sql
INSERT INTO Users (name, email, age)
VALUES ('John Doe', 'john.doe@example.com', 30);
```

Listing 7.10: ตัวอย่างของการสร้างข้อมูล

คำสั่งนี้จะเพิ่มผู้ใช้ใหม่ลงในตาราง 'Users' ด้วยข้อมูล 'name', 'email', และ 'age'

การสร้างข้อมูลด้วย Sequelize:

```
User.create({
    name: 'John Doe',
    email: 'john.doe@example.com',
    age: 30
}).then(user => {
    console.log('User created:', user);
});
```

Listing 7.11: การสร้างข้อมูลด้วย Sequelize

ในตัวอย่างนี้ เราได้ใช้ Sequelize ในการเพิ่มผู้ใช้ใหม่ลงในฐานข้อมูลโดยการใช้เมธอด 'create'

การสร้างข้อมูลด้วย Knex:

```
knex('users').insert({
    name: 'John Doe',
    email: 'john.doe@example.com',
    age: 30
}).then(() => {
    console.log('User created');
});
```

Listing 7.12: การสร้างข้อมูลด้วย Knex

ในตัวอย่างนี้ เราได้ใช้ Knex ในการเพิ่มผู้ใช้ใหม่ลงในฐานข้อมูลโดยการใช้เมธอด 'insert'

7.3.2 การอ่านข้อมูล (Read)

การอ่านข้อมูลในฐานข้อมูล SQL สามารถทำได้โดยการใช้คำสั่ง 'SELECT' เพื่อดึงข้อมูลจากตาราง

ตัวอย่างของการอ่านข้อมูล:

```sql
SELECT * FROM Users;
```

Listing 7.13: ตัวอย่างของการอ่านข้อมูล

คำสั่งนี้จะดึงข้อมูลทั้งหมดจากตาราง 'Users'

การอ่านข้อมูลด้วย Sequelize:

```
User.findAll().then(users => {
    console.log('Users:', users);
});
```

Listing 7.14: การอ่านข้อมูลด้วย Sequelize

ในตัวอย่างนี้ เราได้ใช้ Sequelize ในการดึงข้อมูลผู้ใช้ทั้งหมดจากฐานข้อมูลโดยการใช้เมธอด 'findAll'

การอ่านข้อมูลด้วย Knex:

```
knex.select('*').from('users').then(users => {
    console.log('Users:', users);
});
```

Listing 7.15: การอ่านข้อมูลด้วย Knex

ในตัวอย่างนี้ เราได้ใช้ Knex ในการดึงข้อมูลผู้ใช้ทั้งหมดจากฐานข้อมูล

7.3.3 การแก้ไขข้อมูล (Update)

การแก้ไขข้อมูลในฐานข้อมูล SQL สามารถทำได้โดยการใช้คำสั่ง 'UPDATE' เพื่ออัปเดตข้อมูลในตาราง

ตัวอย่างของการแก้ไขข้อมูล:

```
UPDATE Users
SET email = 'john.newemail@example.com'
WHERE id = 1;
```

Listing 7.16: ตัวอย่างของการแก้ไขข้อมูล

คำสั่งนี้จะอัปเดตอีเมลของผู้ใช้ที่มี 'id' เท่ากับ 1

การแก้ไขข้อมูลด้วย Sequelize:

```
User.update({ email: 'john.newemail@example.com' }, {
    where: { id: 1 }
}).then(() => {
    console.log('User updated');
});
```

Listing 7.17: การแก้ไขข้อมูลด้วย Sequelize

ในตัวอย่างนี้ เราได้ใช้ Sequelize ในการอัปเดตข้อมูลผู้ใช้โดยการใช้เมธอด 'update'

การแก้ไขข้อมูลด้วย Knex:

```
knex('users')
    .where('id', 1)
    .update({ email: 'john.newemail@example.com' })
    .then(() => {
        console.log('User updated');
    });
```

Listing 7.18: การแก้ไขข้อมูลด้วย Knex

ในตัวอย่างนี้ เราได้ใช้ Knex ในการอัปเดตข้อมูลผู้ใช้ในฐานข้อมูล

7.3.4 การลบข้อมูล (Delete)

การลบข้อมูลในฐานข้อมูล SQL สามารถทำได้โดยการใช้คำสั่ง 'DELETE' เพื่อลบข้อมูลจากตาราง

ตัวอย่างของการลบข้อมูล:

```
DELETE FROM Users
WHERE id = 1;
```

Listing 7.19: ตัวอย่างของการลบข้อมูล

คำสั่งนี้จะลบผู้ใช้ที่มี ‘id‘ เท่ากับ 1 จากตาราง ‘Users‘

การลบข้อมูลด้วย Sequelize:

```
User.destroy({
    where: { id: 1 }
}).then(() => {
    console.log('User deleted');
});
```

Listing 7.20: การลบข้อมูลด้วย Sequelize

ในตัวอย่างนี้ เราได้ใช้ Sequelize ในการลบข้อมูลผู้ใช้โดยการใช้เมธอด ‘destroy‘

การลบข้อมูลด้วย Knex:

```
knex('users')
    .where('id', 1)
    .del()
    .then(() => {
        console.log('User deleted');
    });
```

Listing 7.21: การลบข้อมูลด้วย Knex

ในตัวอย่างนี้ เราได้ใช้ Knex ในการลบข้อมูลผู้ใช้ในฐานข้อมูล

7.4 การย้ายและการกำหนดโครงสร้างฐานข้อมูล: การกำหนดและย้ายสคีมาของฐานข้อมูล

การย้ายฐานข้อมูล (Database Migration) เป็นกระบวนการในการเปลี่ยนแปลงโครงสร้างของฐานข้อมูล เช่น การสร้างตารางใหม่ การเพิ่มคอลัมน์ หรือการลบคอลัมน์ โดยไม่ทำให้ข้อมูลที่มีอยู่แล้วสูญหาย การกำหนดและการย้ายโครงสร้างฐานข้อมูลช่วยให้นักพัฒนาสามารถจัดการกับการเปลี่ยนแปลงโครงสร้างของฐานข้อมูลได้อย่างมีประสิทธิภาพ

7.4.1 การย้ายฐานข้อมูลด้วย Sequelize

Sequelize มีระบบการย้ายฐานข้อมูลที่ช่วยให้นักพัฒนาสามารถจัดการกับการเปลี่ยนแปลงโครงสร้างของฐานข้อมูลได้อย่างง่ายดาย

การสร้างการย้ายฐานข้อมูล (Migration):

```
npx sequelize-cli migration:generate --name create-users-table
```

Listing 7.22: การสร้างการย้ายฐานข้อมูลด้วย Sequelize

คำสั่งนี้จะสร้างไฟล์การย้ายฐานข้อมูลใหม่ที่สามารถแก้ไขเพื่อกำหนดโครงสร้างของตาราง

ตัวอย่างของไฟล์การย้ายฐานข้อมูล:

```
'use strict';

module.exports = {
    up: async (queryInterface, Sequelize) => {
        await queryInterface.createTable('Users', {
            id: {
                allowNull: false,
                autoIncrement: true,
                primaryKey: true,
                type: Sequelize.INTEGER
            },
            name: {
                type: Sequelize.STRING,
                allowNull: false
            },
            email: {
                type: Sequelize.STRING,
                allowNull: false,
                unique: true
            },
            age: {
                type: Sequelize.INTEGER
            },
            createdAt: {
                allowNull: false,
                type: Sequelize.DATE
            },
            updatedAt: {
                allowNull: false,
                type: Sequelize.DATE
            }
        });
    },

    down: async (queryInterface, Sequelize) => {
        await queryInterface.dropTable('Users');
    }
};
```

Listing 7.23: ตัวอย่างของไฟล์การย้ายฐานข้อมูลใน Sequelize

ในตัวอย่างนี้ การย้ายฐานข้อมูลจะสร้างตาราง ‘Users‘ ในฟังก์ชัน ‘up‘ และลบตาราง ‘Users‘ ในฟังก์ชัน ‘down‘

การดำเนินการย้ายฐานข้อมูล:

```
npx sequelize-cli db:migrate
```

Listing 7.24: การดำเนินการย้ายฐานข้อมูลด้วย Sequelize

คำสั่งนี้จะดำเนินการย้ายฐานข้อมูลตามที่กำหนดในไฟล์การย้ายฐานข้อมูล

7.4.2 การย้ายฐานข้อมูลด้วย Knex

Knex มีระบบการย้ายฐานข้อมูลที่ช่วยให้นักพัฒนาสามารถจัดการกับการเปลี่ยนแปลงโครงสร้างของฐานข้อมูลได้

การสร้างการย้ายฐานข้อมูล (Migration):

```
npx knex migrate:make create_users_table
```

Listing 7.25: การสร้างการย้ายฐานข้อมูลด้วย Knex

คำสั่งนี้จะสร้างไฟล์การย้ายฐานข้อมูลใหม่ที่สามารถแก้ไขเพื่อกำหนดโครงสร้างของตาราง
ตัวอย่างของไฟล์การย้ายฐานข้อมูล:

```
exports.up = function(knex) {
    return knex.schema.createTable('users', function(table) {
        table.increments('id').primary();
        table.string('name').notNullable();
        table.string('email').unique().notNullable();
        table.integer('age');
        table.timestamps(true, true);
    });
};

exports.down = function(knex) {
    return knex.schema.dropTable('users');
};
```

Listing 7.26: ตัวอย่างของไฟล์การย้ายฐานข้อมูลใน Knex

ในตัวอย่างนี้ การย้ายฐานข้อมูลจะสร้างตาราง ‘users‘ ในฟังก์ชัน ‘up‘ และลบตาราง ‘users‘ ในฟังก์ชัน ‘down‘
การดำเนินการย้ายฐานข้อมูล:

```
npx knex migrate:latest
```

Listing 7.27: การดำเนินการย้ายฐานข้อมูลด้วย Knex

คำสั่งนี้จะดำเนินการย้ายฐานข้อมูลตามที่กำหนดในไฟล์การย้ายฐานข้อมูล

สรุปท้ายบท

บทนี้ได้เสนอแนวคิดและขั้นตอนพื้นฐานในการทำงานกับฐานข้อมูล SQL ใน Node.js โดยเริ่มต้นจากการทำความเข้าใจเกี่ยวกับฐานข้อมูลเชิงสัมพันธ์และการใช้งาน SQL เบื้องต้น จากนั้นได้แนะนำการบูรณาการฐานข้อมูล SQL เข้ากับ Node.js ด้วยไลบรารีอย่าง Sequelize และ Knex และวิธีการดำเนินการ CRUD (Create, Read, Update, Delete) ในฐานข้อมูล SQL

นอกจากนี้ ยังได้ครอบคลุมถึงการย้ายและการกำหนดโครงสร้างฐานข้อมูลโดยการใช้ Sequelize และ Knex ซึ่งเป็นส่วนสำคัญในการจัดการกับการเปลี่ยนแปลงโครงสร้างฐานข้อมูลในการพัฒนาแอปพลิเคชัน

การเข้าใจและใช้เครื่องมือเหล่านี้อย่างถูกต้องเป็นสิ่งสำคัญสำหรับการพัฒนาแอปพลิเคชันที่มีประสิทธิภาพและสามารถจัดการกับข้อมูลในฐานข้อมูลได้อย่างเหมาะสม

คำถามทบทวน:

1. ฐานข้อมูลเชิงสัมพันธ์คืออะไร และมีบทบาทอย่างไรในแอปพลิเคชันสมัยใหม่?
2. คุณจะบูรณาการฐานข้อมูล SQL เข้ากับ Node.js ได้อย่างไรโดยใช้ Sequelize หรือ Knex?
3. การดำเนินการ CRUD ในฐานข้อมูล SQL มีวิธีการอย่างไร?
4. การย้ายและการกำหนดโครงสร้างฐานข้อมูลมีบทบาทอย่างไรในการพัฒนาแอปพลิเคชัน?

การอ่านเพิ่มเติม:

- *Sequelize Documentation* Sequelize Documentation Team [14]
- *Knex.js Documentation* Knex.js Documentation Team [15]
- *SQL for Everyone* O'Donovan [16]

บทที่ 8

การทำงานร่วมกับ Object-Relational Mapping (ORM)

ในบทนี้ เราจะสำรวจแนวคิดของ Object-Relational Mapping (ORM) และวิธีการใช้ ORM ในการพัฒนาฐานข้อมูลใน Node.js ด้วยการใช้ไลบรารี Sequelize คุณจะได้เรียนรู้เกี่ยวกับการกำหนดโมเดลและความสัมพันธ์ระหว่างตารางในฐานข้อมูล และวิธีการเรียกค้นและจัดการข้อมูลผ่าน ORM ซึ่งเป็นเครื่องมือที่ทำให้การพัฒนาแอปพลิเคชันที่ต้องทำงานร่วมกับฐานข้อมูลเป็นไปได้อย่างง่ายดายและมีประสิทธิภาพ

8.1 Object-Relational Mapping (ORM) คืออะไร?

Object-Relational Mapping (ORM) เป็นเทคนิคในการเชื่อมโยงระหว่างออบเจกต์ในโปรแกรมที่เขียนด้วยภาษาการเขียนโปรแกรมเชิงวัตถุ (OOP) กับฐานข้อมูลเชิงสัมพันธ์ (Relational Database) โดยมีเป้าหมายเพื่อทำให้การจัดการข้อมูลในฐานข้อมูลเป็นไปได้ง่ายขึ้นและไม่ต้องมีการเขียนคำสั่ง SQL โดยตรง

8.1.1 แนวคิดพื้นฐานของ ORM

ORM เป็นเครื่องมือที่ทำหน้าที่เป็นสะพานเชื่อมระหว่างสองโลกที่ต่างกันอย่างสิ้นเชิง ได้แก่ โลกของโปรแกรมที่ใช้การเขียนโค้ดแบบเชิงวัตถุ (Object-Oriented Programming) และโลกของฐานข้อมูลที่ใช้รูปแบบเชิงสัมพันธ์ (Relational Database) ORM ช่วยให้โปรแกรมเมอร์สามารถทำงานกับข้อมูลในฐานข้อมูลผ่านการจัดการออบเจกต์ โดยไม่ต้องเขียนคำสั่ง SQL โดยตรง

ตัวอย่าง:

หากคุณมีตารางในฐานข้อมูลที่ชื่อว่า `users` และมีคอลัมน์ `id`, `name`, และ `email` ORM จะช่วยให้คุณสามารถสร้างออบเจกต์ `User` ในโค้ดที่สอดคล้องกับตารางนั้น จากนั้นคุณสามารถใช้ออบเจกต์นี้ในการจัดการข้อมูลในฐานข้อมูล เช่น การเพิ่ม, แก้ไข, หรือลบข้อมูล โดยไม่ต้องเขียนคำสั่ง SQL

ORM ช่วยให้โค้ดของคุณดูเรียบร้อยและง่ายต่อการบำรุงรักษา เนื่องจากคุณสามารถจัดการกับข้อมูลในฐานข้อมูลผ่านการเรียกใช้งานเมธอดของออบเจกต์ที่ถูกสร้างขึ้นมาแทนการเขียน SQL ดิบ

8.1.2 ข้อดีของการใช้ ORM

การใช้ ORM มีข้อดีหลายประการสำหรับนักพัฒนาซอฟต์แวร์ ดังนี้:

- **การเพิ่มประสิทธิภาพในการพัฒนา**: ORM ช่วยลดความซับซ้อนในการจัดการฐานข้อมูลโดยการซ่อนรายละเอียดที่ซับซ้อนของ SQL และให้เครื่องมือที่ใช้งานง่ายสำหรับการดำเนินการที่ต้องทำงานกับฐานข้อมูล
- **การลดข้อผิดพลาดที่เกี่ยวข้องกับ SQL**: เนื่องจาก ORM ช่วยให้คุณสามารถทำงานกับฐานข้อมูลโดยไม่ต้องเขียน SQL โดยตรง ข้อผิดพลาดที่เกี่ยวข้องกับการเขียนคำสั่ง SQL จึงลดลง
- **การจัดการการโยกย้ายฐานข้อมูลได้ง่ายขึ้น**: ORM มักมาพร้อมกับเครื่องมือในการจัดการการโยกย้ายฐานข้อมูล (Database Migration) ซึ่งช่วยให้คุณสามารถจัดการการเปลี่ยนแปลงโครงสร้างฐานข้อมูลได้ง่ายและเป็นระบบ

- **การพัฒนาแบบ Cross-Database**: ORM ช่วยให้คุณสามารถพัฒนาแอปพลิเคชันที่รองรับฐานข้อมูลหลายประเภทได้ง่ายขึ้น โดยไม่ต้องเปลี่ยนแปลงโค้ดมากนัก
- **การรักษาความสมบูรณ์ของข้อมูล**: ORM ช่วยให้คุณสามารถกำหนดความสัมพันธ์และข้อจำกัดระหว่างตารางในฐานข้อมูลได้ง่ายขึ้น ซึ่งช่วยให้การรักษาความสมบูรณ์ของข้อมูลเป็นไปได้อย่างมีประสิทธิภาพ

ORM ไม่ได้มีข้อดีเพียงแต่ในแง่ของการลดความซับซ้อนในการพัฒนา แต่ยังช่วยให้โค้ดของคุณดูสวยงามและเข้าใจง่ายขึ้น เนื่องจากการทำงานกับฐานข้อมูลจะถูกแปลงเป็นการทำงานกับออบเจกต์ในภาษาการเขียนโปรแกรมที่คุณใช้

8.2 การใช้ ORM กับ Node.js: การตั้งค่าและการกำหนดค่า Sequelize

ใน Node.js มีหลายไลบรารีที่สามารถใช้เป็น ORM ได้ แต่หนึ่งในไลบรารียอดนิยมที่นักพัฒนามักใช้คือ **Sequelize** Sequelize เป็นไลบรารี ORM สำหรับ Node.js ที่ช่วยให้คุณสามารถเชื่อมต่อกับฐานข้อมูลเชิงสัมพันธ์ เช่น MySQL, PostgreSQL, SQLite, และ MSSQL และจัดการกับข้อมูลในฐานข้อมูลผ่านการใช้งานออบเจกต์

8.2.1 การติดตั้ง Sequelize และการตั้งค่าโปรเจกต์

ก่อนที่คุณจะเริ่มต้นใช้งาน Sequelize คุณจำเป็นต้องติดตั้ง Sequelize และไลบรารีที่เชื่อมต่อกับฐานข้อมูล (Database Driver) ที่คุณต้องการใช้

ขั้นตอนการติดตั้ง Sequelize:

1. สร้างโปรเจกต์ Node.js ใหม่และสร้างไฟล์ `package.json`:

```
mkdir my-sequelize-app
cd my-sequelize-app
npm init -y
```

Listing 8.1: การสร้างโปรเจกต์และไฟล์ package.json

2. ติดตั้ง Sequelize และ Database Driver ที่คุณต้องการใช้ ตัวอย่างเช่น หากคุณใช้ PostgreSQL:

```
npm install sequelize pg pg-hstore
```

Listing 8.2: การติดตั้ง Sequelize และ PostgreSQL Driver

3. สร้างไฟล์ `index.js` เพื่อเป็นไฟล์หลักของโปรเจกต์:

```
touch index.js
```

Listing 8.3: การสร้างไฟล์ index.js

8.2.2 การตั้งค่า Sequelize

หลังจากที่คุณติดตั้ง Sequelize แล้ว ขั้นตอนถัดไปคือการตั้งค่า Sequelize เพื่อเชื่อมต่อกับฐานข้อมูลที่คุณเลือกใช้ โดยในที่นี้เราจะใช้ PostgreSQL เป็นตัวอย่าง

ตัวอย่างของการตั้งค่า Sequelize:

```
const { Sequelize } = require('sequelize');

// Connecting to the database
const sequelize = new Sequelize('database_name', 'username', 'password', {
    host: 'localhost',
    dialect: 'postgres' // or 'mysql', 'sqlite', 'mssql'
});

// Testing the connection
sequelize.authenticate()
    .then(() => {
        console.log('Connection has been established successfully.');
    })
    .catch(err => {
        console.error('Unable to connect to the database:', err);
    });

module.exports = sequelize;
```

Listing 8.4: Setting Up Sequelize

ในตัวอย่างนี้ เราได้สร้างการเชื่อมต่อกับฐานข้อมูล PostgreSQL โดยใช้ Sequelize เมื่อการเชื่อมต่อเสร็จสิ้น คุณสามารถทดสอบการเชื่อมต่อเพื่อให้แน่ใจว่าการตั้งค่าของคุณถูกต้อง

พารามิเตอร์ที่ใช้ในการตั้งค่า Sequelize:

- `database_name`: ชื่อของฐานข้อมูลที่คุณต้องการเชื่อมต่อ
- `username`: ชื่อผู้ใช้สำหรับการเชื่อมต่อฐานข้อมูล
- `password`: รหัสผ่านสำหรับการเชื่อมต่อฐานข้อมูล
- `host`: ที่อยู่ของเซิร์ฟเวอร์ฐานข้อมูล (ในกรณีนี้คือ `localhost`)
- `dialect`: ประเภทของฐานข้อมูลที่คุณใช้ เช่น `postgres`, `mysql`, `sqlite`, `mssql`

การตั้งค่า Sequelize ช่วยให้คุณสามารถเชื่อมต่อกับฐานข้อมูลและเริ่มต้นการทำงานกับ ORM ได้

8.3 การกำหนดโมเดลและความสัมพันธ์: ความสัมพันธ์ระหว่างตารางใน ORM

ใน ORM โมเดล (Model) เป็นตัวแทนของตารางในฐานข้อมูล แต่ละโมเดลจะถูกสร้างขึ้นเพื่อทำหน้าที่ในการจัดการข้อมูลในตารางนั้น ๆ โมเดลใน Sequelize สามารถกำหนดความสัมพันธ์ระหว่างตารางได้อย่างง่ายดาย เช่น ความสัมพันธ์แบบหนึ่งต่อหนึ่ง (One-to-One), หนึ่งต่อหลาย (One-to-Many), และหลายต่อหลาย (Many-to-Many)

8.3.1 การสร้างโมเดลใน Sequelize

การสร้างโมเดลใน Sequelize นั้นทำได้ง่ายและตรงไปตรงมา คุณสามารถใช้ Sequelize เพื่อกำหนดโครงสร้างของตารางในฐานข้อมูลและจัดการกับข้อมูลในตารางนั้น

ตัวอย่างของการสร้างโมเดลใน Sequelize:

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = require('./index'); // The pre-configured connection

// Defining the User model
const User = sequelize.define('User', {
    id: {
        type: DataTypes.INTEGER,
        autoIncrement: true,
        primaryKey: true
    },
    name: {
        type: DataTypes.STRING,
        allowNull: false
    },
    email: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: true
    }
}, {
    timestamps: true // Enable timestamps for createdAt and updatedAt
});

module.exports = User;
```

Listing 8.5: Creating a Model in Sequelize

ในตัวอย่างนี้ เราได้สร้างโมเดล `User` ที่สอดคล้องกับตาราง `users` ในฐานข้อมูล โมเดลนี้มีฟิลด์ `id`, `name`, และ `email` ซึ่งแต่ละฟิลด์จะถูกกำหนดด้วยประเภทข้อมูลและคุณสมบัติที่จำเป็น เช่น `allowNull`, `unique`, และ `primaryKey`

รายละเอียดของฟิลด์ในโมเดล:

- `type`: กำหนดประเภทข้อมูลของฟิลด์ เช่น `INTEGER`, `STRING`, `BOOLEAN`
- `allowNull`: กำหนดว่าอนุญาตให้ฟิลด์นี้เป็นค่าว่าง (null) ได้หรือไม่
- `unique`: กำหนดว่าค่าของฟิลด์นี้ต้องไม่ซ้ำกันในตาราง
- `primaryKey`: กำหนดว่าฟิลด์นี้เป็นคีย์หลักของตาราง

8.3.2 การกำหนดความสัมพันธ์ระหว่างโมเดล

Sequelize ช่วยให้คุณสามารถกำหนดความสัมพันธ์ระหว่างโมเดลต่าง ๆ ได้อย่างง่ายดาย เช่น การกำหนดความสัมพันธ์แบบหนึ่งต่อหนึ่ง (One-to-One), หนึ่งต่อหลาย (One-to-Many), และหลายต่อหลาย (Many-to-Many)

ตัวอย่างของการกำหนดความสัมพันธ์แบบหนึ่งต่อหลาย (One-to-Many):

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = require('./index'); // The pre-configured connection

// Defining the User model
const User = sequelize.define('User', {
    id: {
        type: DataTypes.INTEGER,
        autoIncrement: true,
        primaryKey: true
    },
    name: {
        type: DataTypes.STRING,
        allowNull: false
    }
}, {
    timestamps: true
});

// Defining the Post model
const Post = sequelize.define('Post', {
    id: {
        type: DataTypes.INTEGER,
        autoIncrement: true,
        primaryKey: true
    },
    title: {
        type: DataTypes.STRING,
        allowNull: false
    },
    content: {
        type: DataTypes.TEXT,
        allowNull: false
    }
}, {
    timestamps: true
});

// Defining the relationship between User and Post
User.hasMany(Post, { foreignKey: 'userId' });
Post.belongsTo(User, { foreignKey: 'userId' });

module.exports = { User, Post };
```

Listing 8.6: Defining One-to-Many Relationships in Sequelize

ในตัวอย่างนี้ เราได้สร้างโมเดล `User` และ `Post` และกำหนดความสัมพันธ์ระหว่างโมเดลทั้งสองเป็นความสัมพันธ์แบบหนึ่งต่อหลาย (One-to-Many) โดยที่ผู้ใช้หนึ่งคนสามารถมีโพสต์หลายโพสต์ ความสัมพันธ์นี้ถูกกำหนดด้วย `User.hasMany(Post)` และ `Post.belongsTo(User)` พร้อมกับการกำหนด `foreignKey` ที่เชื่อมโยงโมเดลทั้งสองเข้าด้วยกัน

8.3.3 ความสัมพันธ์แบบหลายต่อหลาย (Many-to-Many)

ในบางกรณี คุณอาจต้องการกำหนดความสัมพันธ์แบบหลายต่อหลาย (Many-to-Many) ระหว่างโมเดลต่าง ๆ เช่น ความสัมพันธ์ระหว่างผู้ใช้และบทบาท (roles) ที่ผู้ใช้หนึ่งคนสามารถมีหลายบทบาท และบทบาทหนึ่งบทบาทสามารถมีผู้ใช้หลายคน

ตัวอย่างของการกำหนดความสัมพันธ์แบบหลายต่อหลาย (Many-to-Many):

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = require('./index'); // The pre-configured connection

// Defining the User model
const User = sequelize.define('User', {
    id: {
        type: DataTypes.INTEGER,
        autoIncrement: true,
        primaryKey: true
    },
    name: {
        type: DataTypes.STRING,
        allowNull: false
    }
}, {
    timestamps: true
});

// Defining the Role model
const Role = sequelize.define('Role', {
    id: {
        type: DataTypes.INTEGER,
        autoIncrement: true,
        primaryKey: true
    },
    roleName: {
        type: DataTypes.STRING,
        allowNull: false
    }
}, {
    timestamps: true
});

// Defining the many-to-many relationship between User and Role
User.belongsToMany(Role, { through: 'UserRole' });
Role.belongsToMany(User, { through: 'UserRole' });

module.exports = { User, Role };
```

Listing 8.7: Defining Many-to-Many Relationships in Sequelize

ในตัวอย่างนี้ เราได้กำหนดความสัมพันธ์แบบหลายต่อหลายระหว่างโมเดล **User** และ **Role** โดยใช้ตารางกลางที่ชื่อว่า **UserRole** ความสัมพันธ์นี้ช่วยให้ผู้ใช้หนึ่งคนสามารถมีหลายบทบาท และบทบาทหนึ่งสามารถถูกกำหนดให้กับผู้ใช้หลายคน

การกำหนดความสัมพันธ์ระหว่างโมเดลช่วยให้คุณสามารถจัดการกับข้อมูลในฐานข้อมูลที่มีความซับซ้อนได้อย่างมีประสิทธิภาพ

8.4 การเรียกค้นและการจัดการข้อมูลด้วย ORM: การดึงข้อมูลและการปรับปรุงข้อมูลผ่าน ORM

Sequelize ช่วยให้คุณสามารถเรียกค้นและจัดการข้อมูลในฐานข้อมูลได้อย่างง่ายดายผ่านการใช้งานเมธอดที่ถูกสร้างขึ้นสำหรับโมเดล คุณสามารถใช้ Sequelize ในการดึงข้อมูลที่ต้องการและจัดการกับข้อมูลเหล่านั้นโดยไม่ต้องเขียนคำสั่ง SQL ดิบ

8.4.1 การเรียกค้นข้อมูลใน Sequelize

การเรียกค้นข้อมูลใน Sequelize สามารถทำได้ผ่านเมธอดที่ถูกกำหนดไว้สำหรับโมเดล เช่น **findAll()**, **findByPk()**, **findOne()**, และ **findOrCreate()** ซึ่งช่วยให้การดึงข้อมูลจากฐานข้อมูลเป็นไปได้อย่างง่ายดาย

ตัวอย่างของการเรียกค้นข้อมูลใน Sequelize:

```
const { User } = require('./models');

// Fetching all users
User.findAll()
    .then(users => {
        console.log(users);
    })
    .catch(err => {
        console.error('Error fetching users:', err);
    });

// Fetching a user by ID
User.findByPk(1)
    .then(user => {
        if (!user) {
            console.log('User not found');
        } else {
            console.log(user);
        }
    })
    .catch(err => {
        console.error('Error fetching user:', err);
    });
```

Listing 8.8: Fetching Data in Sequelize

ในตัวอย่างนี้ เราได้ใช้เมธอด `findAll()` เพื่อดึงข้อมูลของผู้ใช้ทั้งหมดจากตาราง `users` และใช้เมธอด `findByPk()` เพื่อดึงข้อมูลของผู้ใช้ที่มี ID เท่ากับ 1

8.4.2 การปรับปรุงและลบข้อมูลใน Sequelize

นอกจากการเรียกค้นข้อมูลแล้ว คุณยังสามารถใช้ Sequelize ในการปรับปรุงและลบข้อมูลในฐานข้อมูลได้อย่างง่ายดาย

ตัวอย่างของการปรับปรุงข้อมูลใน Sequelize:

```
const { User } = require('./models');

// Updating a user by ID
User.update({ name: 'Updated Name' }, {
    where: {
        id: 1
    }
})
    .then(result => {
        console.log('Update successful:', result);
    })
    .catch(err => {
        console.error('Error updating user:', err);
    });
```

Listing 8.9: Updating Data in Sequelize

ในตัวอย่างนี้ เราได้ใช้เมธอด `update()` เพื่อปรับปรุงชื่อของผู้ใช้ที่มี ID เท่ากับ 1

ตัวอย่างของการลบข้อมูลใน Sequelize:

```
const { User } = require('./models');

// Deleting a user by ID
User.destroy({
    where: {
        id: 1
    }
})
    .then(result => {
        console.log('Delete successful:', result);
    })
    .catch(err => {
        console.error('Error deleting user:', err);
    });
```

Listing 8.10: Deleting Data in Sequelize

ในตัวอย่างนี้ เราได้ใช้เมธอด **destroy()** เพื่อลบผู้ใช้ที่มี ID เท่ากับ 1 ออกจากตาราง **users**

8.4.3 การใช้งาน Query ที่ซับซ้อนใน Sequelize

Sequelize ยังรองรับการใช้งาน query ที่ซับซ้อน เช่น การกรองข้อมูล, การจัดกลุ่ม, การจัดเรียงลำดับ, และการรวมข้อมูลจากหลายตาราง (joins) เพื่อให้การดึงข้อมูลเป็นไปตามความต้องการของแอปพลิเคชัน

ตัวอย่างของการใช้ Query ที่ซับซ้อนใน Sequelize:

```
const { User, Post } = require('./models');

// Fetching users with related posts
User.findAll({
    include: [{
        model: Post,
        where: { title: { [Sequelize.Op.like]: '%Sequelize%' } }
    }]
})
    .then(users => {
        console.log(users);
    })
    .catch(err => {
        console.error('Error fetching users with posts:', err);
    });
```

Listing 8.11: Using Complex Queries in Sequelize

ในตัวอย่างนี้ เราได้ใช้การรวมข้อมูล (join) ระหว่างตาราง **users** และ **posts** เพื่อดึงข้อมูลของผู้ใช้ที่มีโพสต์ที่ชื่อโพสต์มีคำว่า “Sequelize”

การใช้ ORM กับ Node.js โดยใช้ Sequelize ช่วยให้การทำงานกับฐานข้อมูลเป็นไปได้อย่างราบรื่นและมีประสิทธิภาพ คุณสามารถจัดการข้อมูลในฐานข้อมูลผ่านการใช้งานออบเจกต์ที่สอดคล้องกับตารางในฐานข้อมูล และใช้ Query ที่ซับซ้อนเพื่อดึงข้อมูลที่ต้องการได้อย่างมีประสิทธิภาพ

สรุปท้ายบท

บทนี้ได้เสนอแนวคิดของ Object-Relational Mapping (ORM) และวิธีการใช้ ORM ในการพัฒนาแอปพลิเคชัน Node.js โดยใช้ไลบรารี Sequelize คุณได้เรียนรู้เกี่ยวกับแนวคิดพื้นฐานของ ORM และข้อดีของการใช้ ORM ในการพัฒนาแอปพลิเคชัน

รวมถึงวิธีการตั้งค่า Sequelize และการกำหนดโมเดลและความสัมพันธ์ระหว่างตารางในฐานข้อมูล

นอกจากนี้ยังครอบคลุมถึงการเรียกค้นและการจัดการข้อมูลในฐานข้อมูลผ่าน ORM ซึ่งช่วยให้การพัฒนาแอปพลิเคชันที่ต้องทำงานร่วมกับฐานข้อมูลเป็นไปได้อย่างง่ายดายและมีประสิทธิภาพ

คำถามทบทวน:

1. Object-Relational Mapping (ORM) คืออะไร และมีข้อดีอย่างไรในการพัฒนาแอปพลิเคชัน?
2. คุณจะใช้ Sequelize ในการเชื่อมต่อกับฐานข้อมูลและสร้างโมเดลใน Node.js ได้อย่างไร?
3. การกำหนดความสัมพันธ์ระหว่างโมเดลใน Sequelize มีความสำคัญอย่างไร และมีวิธีการกำหนดอย่างไร?
4. คุณจะใช้ Sequelize ในการเรียกค้นข้อมูลและจัดการข้อมูลในฐานข้อมูลได้อย่างไร?

การอ่านเพิ่มเติม:

- *Sequelize Documentation* *Sequelize Documentation Team [14]*
- *Node.js Design Patterns* *Casanova and Mammino [8]*
- *SQL & NoSQL Databases* *Meier and Kaufmann [17]*

บทที่ 9

ฐานข้อมูล NoSQL และ Node.js

ในบทนี้ เราจะมาทำความเข้าใจเกี่ยวกับฐานข้อมูล NoSQL และวิธีการผสานการทำงานของฐานข้อมูลเหล่านี้กับ Node.js โดยจะครอบคลุมถึงการใช้ไลบรารี เช่น Mongoose สำหรับการจัดการกับข้อมูลใน MongoDB ซึ่งเป็นฐานข้อมูล NoSQL ที่นิยมใช้อย่างแพร่หลาย นอกจากนี้ยังมีการอธิบายถึงกรณีการใช้งาน NoSQL และวิธีการตัดสินใจเลือกใช้ NoSQL แทน SQL พร้อมกับแนวทางปฏิบัติที่ดีที่สุดในการใช้ฐานข้อมูลประเภทนี้

9.1 บทนำสู่ฐานข้อมูล NoSQL: ภาพรวมของประเภทฐานข้อมูล NoSQL

ฐานข้อมูล **NoSQL** (Not Only SQL) เป็นประเภทฐานข้อมูลที่ออกแบบมาเพื่อรองรับการทำงานกับข้อมูลขนาดใหญ่และโครงสร้างข้อมูลที่ไม่เป็นรูปแบบเดียวกันกับฐานข้อมูลเชิงสัมพันธ์ (SQL) ฐานข้อมูล NoSQL ถูกพัฒนาขึ้นมาเพื่อตอบสนองความต้องการของแอปพลิเคชันที่ต้องจัดการกับข้อมูลแบบกระจาย (distributed data) และข้อมูลที่มีโครงสร้างซับซ้อน

9.1.1 ประเภทของฐานข้อมูล NoSQL

ฐานข้อมูล NoSQL สามารถแบ่งออกเป็นหลายประเภทตามลักษณะการจัดเก็บข้อมูลและการใช้งาน ดังนี้:

- **Document-Oriented Databases**: ฐานข้อมูลประเภทนี้จัดเก็บข้อมูลในรูปแบบเอกสาร (documents) โดยทั่วไปจะใช้ JSON หรือ BSON เป็นรูปแบบในการจัดเก็บ ตัวอย่างเช่น **MongoDB** ซึ่งเป็นหนึ่งในฐานข้อมูล NoSQL ที่นิยมใช้มากที่สุด

- **Key-Value Stores**: ฐานข้อมูลประเภทนี้จัดเก็บข้อมูลในรูปแบบคู่ของคีย์ (key) และค่า (value) ซึ่งเหมาะสำหรับการจัดเก็บข้อมูลที่ต้องเข้าถึงได้รวดเร็ว ตัวอย่างเช่น **Redis** และ **DynamoDB**

- **Column-Oriented Databases**: ฐานข้อมูลประเภทนี้จัดเก็บข้อมูลในรูปแบบคอลัมน์ ซึ่งเหมาะสำหรับการประมวลผลข้อมูลขนาดใหญ่ในลักษณะแบบ analytics ตัวอย่างเช่น **Cassandra** และ **HBase**

- **Graph Databases**: ฐานข้อมูลประเภทนี้จัดเก็บข้อมูลในรูปแบบกราฟ (graph) ซึ่งเหมาะสำหรับการแสดงความสัมพันธ์ระหว่างข้อมูล เช่น โหนด (nodes) และขอบ (edges) ตัวอย่างเช่น **Neo4j** และ **ArangoDB**

9.1.2 การเลือกใช้ฐานข้อมูล NoSQL

การเลือกใช้ฐานข้อมูล NoSQL ขึ้นอยู่กับลักษณะและความต้องการของแอปพลิเคชันที่คุณกำลังพัฒนา ซึ่งการเลือกฐานข้อมูลที่เหมาะสมจะช่วยให้การจัดการข้อมูลเป็นไปได้อย่างมีประสิทธิภาพ ฐานข้อมูล NoSQL มีข้อดีหลายประการที่ทำให้เป็นทางเลือกที่น่าสนใจสำหรับแอปพลิเคชันสมัยใหม่ ดังนี้:

- **การรองรับข้อมูลที่ไม่เป็นโครงสร้าง**: ฐานข้อมูล NoSQL เหมาะสำหรับการจัดเก็บข้อมูลที่ไม่เป็นโครงสร้าง เช่น ข้อความ, รูปภาพ, และข้อมูลเชิงพื้นที่

- **การปรับขนาดได้ง่าย**: ฐานข้อมูล NoSQL ออกแบบมาเพื่อรองรับการเพิ่มหรือลดขนาดระบบได้ง่าย ทำให้สามารถจัดการกับข้อมูลขนาดใหญ่ได้อย่างมีประสิทธิภาพ
- **ความเร็วในการเข้าถึงข้อมูล**: ฐานข้อมูล NoSQL มักมีการเข้าถึงข้อมูลที่รวดเร็ว โดยเฉพาะเมื่อใช้ในแอปพลิเคชันที่ต้องการการเข้าถึงข้อมูลแบบเรียลไทม์

อย่างไรก็ตาม การเลือกใช้ฐานข้อมูล NoSQL ควรพิจารณาถึงความต้องการของแอปพลิเคชันอย่างรอบคอบ เนื่องจากฐานข้อมูล NoSQL บางประเภทอาจมีข้อจำกัดในการจัดการข้อมูลเชิงโครงสร้างและการประมวลผลข้อมูลที่ซับซ้อน

9.2 การผสานฐานข้อมูล NoSQL กับ Node.js: การใช้ไลบรารีเช่น Mongoose

ในการพัฒนาแอปพลิเคชันที่ต้องทำงานร่วมกับฐานข้อมูล NoSQL ใน Node.js นักพัฒนามักใช้ไลบรารีเพื่อช่วยในการจัดการกับข้อมูลในฐานข้อมูล หนึ่งในไลบรารีที่ได้รับความนิยมอย่างมากสำหรับการทำงานกับ MongoDB คือ **Mongoose**

9.2.1 การติดตั้ง Mongoose และการตั้งค่าโปรเจกต์

ก่อนที่จะเริ่มต้นการทำงานกับ MongoDB คุณจำเป็นต้องติดตั้ง Mongoose และตั้งค่าโปรเจกต์ Node.js ของคุณ

ขั้นตอนการติดตั้ง Mongoose:

1. สร้างโปรเจกต์ Node.js ใหม่และสร้างไฟล์ `package.json`:

```
mkdir my-mongodb-app
cd my-mongodb-app
npm init -y
```

2. ติดตั้ง Mongoose โดยใช้คำสั่ง `npm install`:

```
npm install mongoose
```

หลังจากที่ติดตั้ง Mongoose เรียบร้อยแล้ว คุณสามารถเริ่มต้นการพัฒนาแอปพลิเคชันของคุณได้

9.2.2 การตั้งค่า Mongoose

หลังจากที่คุณติดตั้ง Mongoose แล้ว ขั้นตอนถัดไปคือการตั้งค่า Mongoose เพื่อเชื่อมต่อกับ MongoDB คุณสามารถเชื่อมต่อกับฐานข้อมูล MongoDB ที่กำลังทำงานอยู่ได้ง่าย ๆ โดยใช้ Mongoose

ตัวอย่างของการตั้งค่า Mongoose:

```
const mongoose = require('mongoose');

// Connecting to the MongoDB database
mongoose.connect('mongodb://localhost:27017/mydatabase', {
    useNewUrlParser: true,
    useUnifiedTopology: true
})
    .then(() => console.log('Connected to MongoDB'))
    .catch(err => console.error('Could not connect to MongoDB...', err));
```

Listing 9.1: Setting Up Mongoose

ในตัวอย่างนี้ เราได้เชื่อมต่อกับฐานข้อมูล MongoDB ที่ทำงานอยู่บนเซิร์ฟเวอร์ `localhost` และพอร์ต `27017` โดยใช้ Mongoose เมื่อการเชื่อมต่อสำเร็จ ข้อความ `Connected to MongoDB` จะถูกพิมพ์ออกมาในคอนโซล

พารามิเตอร์ที่ใช้ในการตั้งค่า Mongoose:

- `useNewUrlParser`: ใช้เพื่อเปิดใช้งานการวิเคราะห์ URL แบบใหม่
- `useUnifiedTopology`: ใช้เพื่อเปิดใช้งานเครื่องมือการเชื่อมต่อใหม่ที่มีความเสถียรขึ้น

การตั้งค่า Mongoose ช่วยให้คุณสามารถเชื่อมต่อกับ MongoDB และเริ่มต้นการทำงานกับฐานข้อมูลได้อย่างง่ายดาย

9.3 การทำงานกับฐานข้อมูลที่เป็นเอกสาร: การสร้าง, อ่าน, อัปเดต, และลบเอกสารใน MongoDB

MongoDB เป็นฐานข้อมูล NoSQL ที่จัดเก็บข้อมูลในรูปแบบเอกสาร (documents) โดยเอกสารใน MongoDB มีโครงสร้างคล้ายกับ JSON ซึ่งทำให้ง่ายต่อการจัดการและการดึงข้อมูล

9.3.1 การสร้างโมเดลใน Mongoose

ใน Mongoose โมเดล (Model) ถูกใช้เพื่อแสดงถึงเอกสารในฐานข้อมูล โดยโมเดลถูกสร้างขึ้นจาก Schema ซึ่งกำหนดโครงสร้างและรูปแบบของข้อมูลในเอกสาร

ตัวอย่างของการสร้างโมเดลใน Mongoose:

```
const User = require('./models/User');

// Creating a new user document
const newUser = new User({
    name: 'Alice',
    email: 'alice@example.com',
    age: 25
});

newUser.save()
    .then(user => {
        console.log('User saved:', user);
    })
    .catch(err => {
        console.error('Error saving user:', err);
    });
```

Listing 9.2: Creating a New Document in Mongoose

ในตัวอย่างนี้ เราได้กำหนด `userSchema` เพื่อระบุโครงสร้างของเอกสาร `User` ซึ่งประกอบด้วยฟิลด์ `name`, `email`, และ `age` โดยมีการกำหนดคุณสมบัติต่าง ๆ เช่น `required`, `unique`, และ `min` หลังจากนั้น เราได้สร้างโมเดล `User` จาก Schema ที่กำหนดไว้

รายละเอียดของฟิลด์ใน Schema:

- `type`: กำหนดประเภทข้อมูลของฟิลด์ เช่น `String`, `Number`, `Date`
- `required`: กำหนดว่าฟิลด์นี้ต้องมีค่าและไม่สามารถเป็นค่าว่างได้
- `unique`: กำหนดว่าค่าของฟิลด์นี้ต้องไม่ซ้ำกันในฐานข้อมูล
- `min`: กำหนดค่าขั้นต่ำที่ฟิลด์สามารถมีได้ (ใช้สำหรับข้อมูลประเภทตัวเลข)

9.3.2 การสร้างเอกสารใหม่

หลังจากที่คุณสร้างโมเดลเรียบร้อยแล้ว คุณสามารถใช้โมเดลนี้ในการสร้างเอกสารใหม่และบันทึกลงในฐานข้อมูล MongoDB ได้

ตัวอย่างของการสร้างเอกสารใหม่ใน Mongoose:

```
const User = require('./models/User');

// Creating a new user document
const newUser = new User({
    name: 'Alice',
    email: 'alice@example.com',
    age: 25
});

newUser.save()
    .then(user => {
        console.log('User saved:', user);
    })
    .catch(err => {
        console.error('Error saving user:', err);
    });
```

Listing 9.3: Creating a New Document in Mongoose

ในตัวอย่างนี้ เราได้สร้างเอกสาร `User` ใหม่และบันทึกลงในฐานข้อมูลโดยใช้เมธอด `save()` เมื่อการบันทึกสำเร็จ ข้อมูลของผู้ใช้จะถูกพิมพ์ออกมาในคอนโซล

9.3.3 การอ่านเอกสาร

การอ่านข้อมูลใน MongoDB สามารถทำได้ผ่านเมธอดต่าง ๆ ที่ Mongoose จัดเตรียมไว้ เช่น `find()`, `findById()`, และ `findOne()` ซึ่งช่วยให้การดึงข้อมูลเป็นไปได้อย่างง่ายดาย

ตัวอย่างของการอ่านเอกสารใน Mongoose:

```
const User = require('./models/User');

// Fetching all users
User.find()
    .then(users => {
        console.log('All users:', users);
    })
    .catch(err => {
        console.error('Error fetching users:', err);
    });

// Fetching a user by ID
User.findById('60c72b2f4f1a2c001fbd35c5')
    .then(user => {
        if (!user) {
            console.log('User not found');
        } else {
            console.log('User found:', user);
        }
    })
    .catch(err => {
        console.error('Error fetching user:', err);
    });
```

Listing 9.4: Reading Documents in Mongoose

ในตัวอย่างนี้ เราได้ใช้เมธอด `find()` เพื่อดึงข้อมูลผู้ใช้ทั้งหมดจากฐานข้อมูล และใช้เมธอด `findById()` เพื่อดึงข้อมูลของผู้ใช้ที่มี ID ที่กำหนด

9.3.4 การอัปเดตเอกสาร

Mongoose ช่วยให้คุณสามารถอัปเดตข้อมูลในเอกสารได้อย่างง่ายดายผ่านเมธอด เช่น `updateOne()`, `updateMany()`, และ `findByIdAndUpdate()`

ตัวอย่างของการอัปเดตเอกสารใน Mongoose:

```
const User = require('./models/User');

// Updating a user's name by ID
User.findByIdAndUpdate('60c72b2f4f1a2c001fbd35c5', { name: 'Updated Alice'
    }, { new: true })
    .then(user => {
        console.log('Updated user:', user);
    })
    .catch(err => {
        console.error('Error updating user:', err);
    });
```

Listing 9.5: Updating Documents in Mongoose

ในตัวอย่างนี้ เราได้ใช้เมธอด `findByIdAndUpdate()` เพื่ออัปเดตชื่อของผู้ใช้ที่มี ID ที่กำหนด โดยผลลัพธ์ที่ได้จะเป็นข้อมูลของผู้ใช้ที่ถูกอัปเดตแล้ว

9.3.5 การลบเอกสาร

การลบเอกสารใน MongoDB สามารถทำได้ผ่านเมธอด เช่น `deleteOne()`, `deleteMany()`, และ `findByIdAndDelete()`

ตัวอย่างของการลบเอกสารใน Mongoose:

```
const User = require('./models/User');

//        ID
User.findByIdAndDelete('60c72b2f4f1a2c001fbd35c5')
    .then(result => {
        console.log('User deleted:', result);
    })
    .catch(err => {
        console.error('Error deleting user:', err);
    });
```

Listing 9.6: การลบเอกสารใน Mongoose

ในตัวอย่างนี้ เราได้ใช้เมธอด `findByIdAndDelete()` เพื่อลบผู้ใช้ที่มี ID ที่กำหนดออกจากฐานข้อมูล

การทำงานกับเอกสารใน MongoDB ผ่าน Mongoose ช่วยให้การจัดการข้อมูลเป็นไปได้อย่างง่ายดายและมีประสิทธิภาพ คุณสามารถใช้เมธอดต่าง ๆ ที่ Mongoose จัดเตรียมไว้เพื่อสร้าง, อ่าน, อัปเดต, และลบเอกสารในฐานข้อมูลได้อย่างรวดเร็ว

9.4 กรณีการใช้งาน NoSQL: เมื่อควรเลือก NoSQL แทน SQL และแนวทางปฏิบัติที่ดีที่สุด

NoSQL เป็นฐานข้อมูลที่มีความยืดหยุ่นสูงและสามารถรองรับข้อมูลที่มีโครงสร้างไม่แน่นอนและขนาดใหญ่ได้ อย่างไรก็ตาม การเลือกใช้ NoSQL แทน SQL ควรพิจารณาจากลักษณะของแอปพลิเคชันและความต้องการในการจัดการข้อมูล

9.4.1 เมื่อควรเลือก NoSQL แทน SQL

มีหลายกรณีที่การเลือกใช้ NoSQL เหมาะสมกว่าการใช้ SQL ดังนี้:

- **ข้อมูลที่ไม่เป็นโครงสร้าง (Unstructured Data)**: หากคุณต้องจัดการกับข้อมูลที่ไม่มีโครงสร้างที่ชัดเจน เช่น ข้อความ รูปภาพ วิดีโอ หรือข้อมูลที่เปลี่ยนแปลงโครงสร้างบ่อย การใช้ NoSQL จะทำให้การจัดการข้อมูลเหล่านี้เป็นไปได้ง่ายขึ้น
- **การรองรับการขยายระบบแนวนอน (Horizontal Scaling)**: หากแอปพลิเคชันของคุณต้องรองรับการขยายระบบแบบแนวนอน (เช่น การเพิ่มเซิร์ฟเวอร์เพื่อรองรับการทำงานมากขึ้น) NoSQL มักจะมีความยืดหยุ่นในการขยายระบบได้ดีกว่า SQL
- **การจัดการข้อมูลแบบเรียลไทม์ (Real-time Data Management)**: หากแอปพลิเคชันของคุณต้องการการเข้าถึงข้อมูลแบบเรียลไทม์ เช่น การวิเคราะห์ข้อมูลสด (live analytics) หรือการจัดการกับการแจ้งเตือนแบบเรียลไทม์ NoSQL สามารถตอบสนองความต้องการนี้ได้อย่างมีประสิทธิภาพ
- **ความต้องการความยืดหยุ่นในการจัดเก็บข้อมูล (Flexible Schema)**: หากคุณต้องการโครงสร้างฐานข้อมูลที่ยืดหยุ่นและสามารถปรับเปลี่ยนได้ง่าย NoSQL จะช่วยให้คุณสามารถเพิ่มหรือลดฟิลด์ในเอกสารโดยไม่ต้องปรับโครงสร้างฐานข้อมูลทั้งหมด

9.4.2 แนวทางปฏิบัติที่ดีที่สุดในการใช้ NoSQL

แม้ว่า NoSQL จะมีข้อดีหลายประการ แต่การใช้ NoSQL ก็ต้องอาศัยการวางแผนและการออกแบบที่ดีเพื่อให้การจัดการข้อมูลเป็นไปอย่างมีประสิทธิภาพ นี่คือแนวทางปฏิบัติที่ดีที่สุดในการใช้ NoSQL:

- **การเลือกประเภทของฐานข้อมูลที่เหมาะสม**: ควรเลือกประเภทของฐานข้อมูล NoSQL ที่เหมาะสมกับลักษณะข้อมูลและความต้องการของแอปพลิเคชัน เช่น การใช้ Document-Oriented Databases สำหรับการจัดการข้อมูลที่เป็นเอกสาร หรือการใช้ Key-Value Stores สำหรับการจัดการข้อมูลที่ต้องเข้าถึงได้รวดเร็ว

- **การออกแบบโครงสร้างข้อมูลอย่างรอบคอบ**: แม้ว่า NoSQL จะมีความยืดหยุ่นสูง แต่การออกแบบโครงสร้างข้อมูลที่ดีตั้งแต่ต้นจะช่วยให้การจัดการข้อมูลและการดึงข้อมูลเป็นไปได้อย่างมีประสิทธิภาพ
- **การพิจารณาความสม่ำเสมอของข้อมูล (Consistency)**: ควรพิจารณาถึงความสม่ำเสมอของข้อมูลในระบบที่ใช้ NoSQL เนื่องจาก NoSQL มักใช้แนวคิดของ CAP Theorem ซึ่งอาจทำให้เกิดการแลกเปลี่ยนระหว่าง Consistency, Availability, และ Partition Tolerance
- **การพิจารณาประสิทธิภาพการทำงาน (Performance)**: ควรทดสอบและตรวจสอบประสิทธิภาพของฐานข้อมูล NoSQL ในการจัดการกับโหลดการทำงานที่คาดหวัง เพื่อให้แน่ใจว่าฐานข้อมูลสามารถรองรับการทำงานได้อย่างเหมาะสม

การใช้ NoSQL อย่างมีประสิทธิภาพจะช่วยให้แอปพลิเคชันของคุณสามารถจัดการกับข้อมูลขนาดใหญ่และข้อมูลที่ไม่เป็นโครงสร้างได้อย่างมีประสิทธิภาพ และสามารถรองรับการขยายระบบได้อย่างยืดหยุ่น

9.5 สรุปท้ายบท

บทนี้ได้เสนอแนวคิดพื้นฐานเกี่ยวกับฐานข้อมูล NoSQL และวิธีการผสานการทำงานของฐานข้อมูลเหล่านี้กับ Node.js โดยใช้ไลบรารี Mongoose สำหรับการจัดการข้อมูลใน MongoDB ซึ่งเป็นหนึ่งในฐานข้อมูล NoSQL ที่ได้รับความนิยมสูงสุด คุณได้เรียนรู้เกี่ยวกับการสร้าง, อ่าน, อัปเดต, และลบเอกสารใน MongoDB ผ่านการใช้ Mongoose รวมถึงแนวทางปฏิบัติที่ดีที่สุดในการใช้ NoSQL ในการพัฒนาแอปพลิเคชัน

การเข้าใจและใช้เครื่องมือ NoSQL อย่างถูกต้องเป็นสิ่งสำคัญสำหรับการพัฒนาแอปพลิเคชันที่ต้องจัดการกับข้อมูลขนาดใหญ่และข้อมูลที่ไม่เป็นโครงสร้าง โดยมีประสิทธิภาพในการจัดการและความยืดหยุ่นในการขยายระบบ

คำถามทบทวน:

1. NoSQL คืออะไร และมีประเภทอะไรบ้างที่ใช้ในการพัฒนาแอปพลิเคชัน?
2. คุณจะใช้ Mongoose ในการเชื่อมต่อกับ MongoDB และจัดการข้อมูลใน Node.js ได้อย่างไร?
3. การสร้างและการจัดการเอกสารใน MongoDB ผ่าน Mongoose ทำได้อย่างไร?
4. เมื่อใดที่ควรเลือกใช้ NoSQL แทน SQL และมีแนวทางปฏิบัติที่ดีที่สุดอย่างไรในการใช้ NoSQL?

การอ่านเพิ่มเติม:

- *MongoDB Documentation* *MongoDB Documentation Team [18]*
- *Mongoose Documentation* *Mongoose.js Documentation Team [19]*
- *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence* *Sadalage and Fowler [20]*

บทที่ 10

บทนำสู่การพัฒนา Front-End

การพัฒนา Front-End เป็นส่วนสำคัญในการสร้างประสบการณ์การใช้งานที่ดีสำหรับผู้ใช้ที่เข้าถึงแอปพลิเคชันผ่านเว็บเบราว์เซอร์ โดย Front-End คือส่วนที่ผู้ใช้มองเห็นและโต้ตอบกับแอปพลิเคชัน ซึ่งการพัฒนา Front-End จะเกี่ยวข้องกับการใช้ HTML, CSS, และ JavaScript นอกจากนี้ยังมีการใช้เครื่องมือและไลบรารีต่าง ๆ เพื่อช่วยในการพัฒนา Front-End อย่างมีประสิทธิภาพ ในบทนี้เราจะสำรวจบทบาทของ HTML, CSS, และ JavaScript ในการพัฒนาเว็บ การตั้งค่าสภาพแวดล้อมสำหรับการพัฒนา Front-End การแนะนำเฟรมเวิร์กยอดนิยม และหลักการออกแบบพื้นฐานที่ช่วยสร้างประสบการณ์การใช้งานที่ดี

10.1 ความเข้าใจเกี่ยวกับ Front-End: บทบาทของ HTML, CSS, และ JavaScript ในการพัฒนาเว็บ

ในการพัฒนาเว็บแอปพลิเคชัน HTML, CSS, และ JavaScript เป็นสามเสาหลักที่ช่วยให้การสร้างหน้าเว็บเป็นไปได้อย่างราบรื่นและมีประสิทธิภาพ

10.1.1 HTML: โครงสร้างของหน้าเว็บ

HTML (HyperText Markup Language) เป็นภาษามาร์กอัปที่ใช้ในการสร้างโครงสร้างและเนื้อหาของหน้าเว็บ HTML ทำหน้าที่เป็นกระดูกสันหลังของหน้าเว็บ โดยกำหนดว่าองค์ประกอบต่าง ๆ บนหน้าเว็บจะถูกจัดวางอย่างไร และมีเนื้อหาอะไรบ้าง

ตัวอย่างของ HTML:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Web Page</title>
</head>
<body>
    <header>
        <h1>Welcome to My Web Page</h1>
    </header>
    <nav>
        <ul>
            <li><a href="#home">Home</a></li>
            <li><a href="#about">About</a></li>
            <li><a href="#contact">Contact</a></li>
        </ul>
    </nav>
    <main>
        <section id="home">
            <h2>Home Section</h2>
            <p>This is the home section of the web page.</p>
        </section>
        <section id="about">
            <h2>About Section</h2>
            <p>This is the about section of the web page.</p>
        </section>
        <section id="contact">
            <h2>Contact Section</h2>
            <p>This is the contact section of the web page.</p>
        </section>
    </main>
    <footer>
        <p>&copy; 2024 My Web Page</p>
    </footer>
</body>
</html>
```

Listing 10.1: โครงสร้าง HTML พื้นฐาน

ในตัวอย่างนี้ HTML ถูกใช้เพื่อสร้างโครงสร้างของหน้าเว็บที่มีส่วนประกอบต่าง ๆ เช่น header, nav, main, และ footer HTML ช่วยกำหนดลำดับการแสดงผลขององค์ประกอบต่าง ๆ บนหน้าเว็บ ทำให้ผู้ใช้สามารถนำทางและโต้ตอบกับเนื้อหาบนหน้าเว็บได้อย่างมีประสิทธิภาพ

10.1.2 CSS: การจัดรูปแบบและการนำเสนอ

CSS (Cascading Style Sheets) เป็นภาษาที่ใช้ในการจัดรูปแบบและการนำเสนอเนื้อหาของหน้าเว็บ CSS ช่วยให้คุณสามารถปรับแต่งลักษณะการแสดงผลของ HTML ได้ เช่น สี, ขนาด, การจัดวาง, และการเคลื่อนไหว (animations) CSS ทำให้หน้าเว็บของคุณมีความสวยงามและน่าสนใจยิ่งขึ้น

ตัวอย่างของ CSS:

```css
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
}

header {
    background-color: #4CAF50;
    color: white;
    text-align: center;
    padding: 1em;
}

nav ul {
    list-style-type: none;
    margin: 0;
    padding: 0;
    background-color: #333;
    overflow: hidden;
}

nav ul li {
    float: left;
}

nav ul li a {
    display: block;
    color: white;
    text-align: center;
    padding: 14px 16px;
    text-decoration: none;
}

nav ul li a:hover {
    background-color: #111;
}

main {
    padding: 1em;
}

footer {
    background-color: #333;
    color: white;
    text-align: center;
    padding: 1em;
    position: fixed;
    bottom: 0;
    width: 100%;
}
```

Listing 10.2: การจัดรูปแบบด้วย CSS

ในตัวอย่างนี้ CSS ถูกใช้เพื่อปรับแต่งลักษณะการแสดงผลของ HTML ที่ถูกสร้างขึ้นในตัวอย่างก่อนหน้า CSS ช่วยให้หน้าเว็บของคุณมีความสวยงามและเป็นระเบียบมากขึ้น โดยการกำหนดสไตล์สำหรับองค์ประกอบต่าง ๆ เช่น header, nav, main, และ footer

10.1.3 JavaScript: ความสามารถในการโต้ตอบและฟังก์ชัน

JavaScript เป็นภาษาการเขียนโปรแกรมที่ใช้ในการเพิ่มความสามารถในการโต้ตอบและฟังก์ชันต่าง ๆ ให้กับหน้าเว็บ JavaScript ช่วยให้คุณสามารถทำงานกับ DOM (Document Object Model) เพื่อสร้างการเปลี่ยนแปลงในหน้าเว็บโดยไม่ต้องโหลดหน้าเว็บใหม่ทั้งหมด นอกจากนี้ JavaScript ยังใช้ในการจัดการกับเหตุการณ์ต่าง ๆ (events) เช่น การคลิก, การเลื่อน, และการป้อนข้อมูล

ตัวอย่างของ JavaScript:

```
document.addEventListener('DOMContentLoaded', () => {
    const navLinks = document.querySelectorAll('nav ul li a');
    navLinks.forEach(link => {
        link.addEventListener('click', event => {
            event.preventDefault();
            const sectionId = link.getAttribute('href').substring(1);
            document.getElementById(sectionId).scrollIntoView({ behavior:
                'smooth' });
        });
    });
});
```

Listing 10.3: การเพิ่มความสามารถในการโต้ตอบด้วย JavaScript

ในตัวอย่างนี้ JavaScript ถูกใช้เพื่อเพิ่มฟังก์ชันการเลื่อนหน้าเว็บอย่างราบรื่นเมื่อผู้ใช้คลิกที่ลิงก์ในเมนูนำทาง JavaScript ช่วยให้คุณสามารถเพิ่มความสามารถในการโต้ตอบกับผู้ใช้และสร้างประสบการณ์การใช้งานที่ดีขึ้น

10.2 การตั้งค่าสภาพแวดล้อม Front-End: เครื่องมือและไลบรารี (เช่น npm, webpack)

การตั้งค่าสภาพแวดล้อมในการพัฒนา Front-End เป็นขั้นตอนสำคัญในการพัฒนาเว็บแอปพลิเคชันที่มีประสิทธิภาพ การใช้เครื่องมือและไลบรารีที่เหมาะสมจะช่วยให้คุณสามารถจัดการกับกระบวนการพัฒนาได้อย่างมีประสิทธิภาพมากขึ้น

10.2.1 npm: การจัดการแพ็กเกจ

npm (Node Package Manager) เป็นเครื่องมือสำหรับการจัดการแพ็กเกจใน Node.js ที่ช่วยให้คุณสามารถติดตั้ง, อัปเดต, และจัดการไลบรารีและเครื่องมือที่ใช้ในการพัฒนา Front-End ได้อย่างง่ายดาย npm เป็นแพ็กเกจเมเนเจอร์ที่ได้รับความนิยมอย่างมากในวงการพัฒนาเว็บ

การใช้งาน npm:

- การสร้างไฟล์ `package.json` สำหรับโปรเจกต์:

```
npm init -y
```

Listing 10.4: การสร้างไฟล์ `package.json`

- การติดตั้งแพ็กเกจ Front-End เช่น React:

```
npm install react react-dom
```

Listing 10.5: การติดตั้งแพ็กเกจ Front-End

- การจัดการสคริปต์สำหรับการพัฒนา:

```
{
  "scripts": {
    "start": "webpack serve --open",
    "build": "webpack --mode production"
  }
}
```

Listing 10.6: การจัดการสคริปต์ใน `package.json`

ในตัวอย่างนี้ npm ถูกใช้ในการติดตั้งแพ็กเกจที่จำเป็นสำหรับการพัฒนา Front-End และการจัดการสคริปต์ที่ใช้ในการพัฒนาและการผลิต npm ช่วยให้คุณสามารถจัดการกับไลบรารีและเครื่องมือที่ใช้ในการพัฒนาได้อย่างมีประสิทธิภาพ

10.2.2 Webpack: การบันเดิลโมดูล

Webpack เป็นเครื่องมือที่ใช้ในการบันเดิลโมดูล JavaScript และไฟล์อื่น ๆ เช่น CSS, รูปภาพ, และฟอนต์ เข้าเป็นไฟล์เดียวหรือหลายไฟล์เพื่อให้สามารถนำไปใช้ในโปรเจกต์ได้ง่ายขึ้น Webpack ช่วยให้การจัดการกับโค้ด Front-End เป็นไปได้อย่างมีประสิทธิภาพและมีโครงสร้างที่ดีขึ้น

การตั้งค่า Webpack:

- การติดตั้ง Webpack และ Webpack CLI:

```
npm install --save-dev webpack webpack-cli
```

Listing 10.7: การติดตั้ง Webpack และ Webpack CLI

- การสร้างไฟล์ `webpack.config.js` สำหรับการตั้งค่า Webpack:

```
const path = require('path');

module.exports = {
    entry: './src/index.js',
    output: {
        filename: 'bundle.js',
        path: path.resolve(__dirname, 'dist')
    },
    module: {
        rules: [
            {
                test: /\.css$/,
                use: ['style-loader', 'css-loader']
            },
            {
                test: /\.(png|svg|jpg|jpeg|gif)$/,
                type: 'asset/resource'
            }
        ]
    }
};
```

Listing 10.8: ไฟล์ `webpack.config.js`

- การใช้ Webpack ในการบันเดิลโค้ด:

```
npm run build
```

Listing 10.9: การใช้ Webpack ในการบันเดิลโค้ด

ในตัวอย่างนี้ Webpack ถูกใช้ในการบันเดิลไฟล์ JavaScript และ CSS เข้าเป็นไฟล์เดียวที่สามารถนำไปใช้ในโปรเจกต์ได้ Webpack ช่วยให้การจัดการกับโค้ด Front-End เป็นไปได้อย่างมีประสิทธิภาพและเป็นระเบียบมากขึ้น

10.2.3 การใช้เครื่องมืออื่น ๆ ในการพัฒนา Front-End

นอกจาก npm และ Webpack แล้ว ยังมีเครื่องมืออื่น ๆ ที่สามารถช่วยในการพัฒนา Front-End ได้ เช่น:

- **Babel**: เครื่องมือที่ใช้ในการแปลงโค้ด JavaScript ที่เขียนด้วย ES6+ ให้สามารถทำงานได้บนเบราว์เซอร์ที่ไม่รองรับฟีเจอร์ใหม่ ๆ
- **ESLint**: เครื่องมือสำหรับตรวจสอบและแก้ไขข้อผิดพลาดในโค้ด JavaScript เพื่อให้โค้ดมีคุณภาพและเป็นมาตรฐาน
- **Prettier**: เครื่องมือสำหรับจัดรูปแบบโค้ดให้อ่านง่ายและเป็นมาตรฐานเดียวกัน

การใช้เครื่องมือเหล่านี้ร่วมกันจะช่วยให้การพัฒนา Front-End เป็นไปได้อย่างราบรื่นและมีประสิทธิภาพมากขึ้น

10.3 บทนำสู่เฟรมเวิร์ก Front-End: ภาพรวมของเฟรมเวิร์กยอดนิยม เช่น React, Angular, และ Vue.js

เฟรมเวิร์ก Front-End เป็นเครื่องมือที่ช่วยให้การพัฒนาเว็บแอปพลิเคชันเป็นไปได้อย่างมีประสิทธิภาพมากขึ้น โดยมีการจัดการกับองค์ประกอบต่าง ๆ ของหน้าเว็บอย่างเป็นระบบ และช่วยให้การทำงานร่วมกันระหว่างทีมพัฒนาเป็นไปได้อย่างราบรื่น

10.3.1 React: การสร้าง UI ที่มีประสิทธิภาพ

React เป็นเฟรมเวิร์ก JavaScript ที่ถูกพัฒนาโดย Facebook เพื่อช่วยในการสร้างส่วนติดต่อผู้ใช้ (UI) ที่มีประสิทธิภาพและสามารถจัดการกับการเปลี่ยนแปลงของข้อมูลได้อย่างรวดเร็ว React ใช้แนวคิดของ **component-based architecture** ซึ่งหมายความว่าคุณสามารถสร้าง UI โดยการรวมองค์ประกอบเล็ก ๆ ที่เรียกว่า **components** เข้าด้วยกัน

ตัวอย่างของการใช้งาน React:

```
import React from 'react';
import ReactDOM from 'react-dom';

function App() {
    return (
        <div>
            <h1>Hello, React!</h1>
            <p>This is a simple React component.</p>
        </div>
    );
}

ReactDOM.render(<App />, document.getElementById('root'));
```

Listing 10.10: ตัวอย่างของการใช้งาน React

ในตัวอย่างนี้ React ถูกใช้ในการสร้าง UI ง่าย ๆ ที่ประกอบด้วยหัวข้อและข้อความ React ช่วยให้คุณสามารถสร้าง UI ที่ซับซ้อนโดยการรวม components เล็ก ๆ หลาย ๆ ตัวเข้าด้วยกัน

10.3.2 Angular: การพัฒนาเว็บแอปพลิเคชันที่มีความซับซ้อน

Angular เป็นเฟรมเวิร์ก JavaScript ที่ถูกพัฒนาโดย Google เพื่อช่วยในการพัฒนาเว็บแอปพลิเคชันที่มีความซับซ้อน Angular ใช้แนวคิดของ **two-way data binding** ซึ่งช่วยให้ข้อมูลใน UI และข้อมูลในโมเดลเชื่อมโยงกันได้อย่างราบรื่น นอกจากนี้ Angular ยังมีเครื่องมือและฟีเจอร์ต่าง ๆ ที่ช่วยในการจัดการกับแอปพลิเคชันขนาดใหญ่ เช่น **dependency injection** และ **routing**

ตัวอย่างของการใช้งาน Angular:

```
import { Component } from '@angular/core';

Component({
  selector: 'app-root',
  template: `
    <h1>Hello, Angular!</h1>
    <p>This is a simple Angular component.</p>
  `,
  styles: [`
    h1 {
      color: #4CAF50;
    }
  `]
})
export class AppComponent {
  title = 'my-angular-app';
}
```

Listing 10.11: ตัวอย่างของการใช้งาน Angular

ในตัวอย่างนี้ Angular ถูกใช้ในการสร้าง UI ง่าย ๆ ที่ประกอบด้วยหัวข้อและข้อความ Angular ช่วยให้การพัฒนาแอปพลิเคชันที่มีความซับซ้อนเป็นไปได้อย่างมีประสิทธิภาพมากขึ้น

10.3.3 Vue.js: เฟรมเวิร์กที่ง่ายต่อการเรียนรู้และใช้งาน

Vue.js เป็นเฟรมเวิร์ก JavaScript ที่ออกแบบมาให้ใช้งานง่ายและยืดหยุ่น Vue.js เหมาะสำหรับนักพัฒนาที่ต้องการเริ่มต้นการพัฒนา Front-End ด้วยเฟรมเวิร์กที่ไม่ซับซ้อน Vue.js ใช้แนวคิดของ **reactive data binding** ซึ่งช่วยให้ข้อมูลใน UI และข้อมูลในโมเดลเชื่อมโยงกันได้อย่างง่ายดาย

ตัวอย่างของการใช้งาน Vue.js:

```
<div id="app">
  <h1>{{ message }}</h1>
  <p>This is a simple Vue.js component.</p>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
  new Vue({
    el: '#app',
    data: {
      message: 'Hello, Vue.js!'
    }
  });
</script>
```

Listing 10.12: ตัวอย่างของการใช้งาน Vue.js

ในตัวอย่างนี้ Vue.js ถูกใช้ในการสร้าง UI ง่าย ๆ ที่ประกอบด้วยหัวข้อและข้อความ Vue.js ช่วยให้การพัฒนา UI เป็นไปได้อย่างรวดเร็วและง่ายดาย โดยใช้แนวคิดของ reactive data binding

10.3.4 การเลือกเฟรมเวิร์กที่เหมาะสม

การเลือกเฟรมเวิร์ก Front-End ที่เหมาะสมขึ้นอยู่กับลักษณะและความต้องการของโปรเจกต์ของคุณ ต่อไปนี้เป็นแนวทางในการเลือกเฟรมเวิร์กที่เหมาะสม:

- **React**: เหมาะสำหรับโปรเจกต์ที่ต้องการความยืดหยุ่นและต้องการสร้าง UI ที่มีการเปลี่ยนแปลงของข้อมูลบ่อยครั้ง React มีชุมชนผู้ใช้ที่กว้างขวางและเครื่องมือที่ช่วยในการพัฒนาอย่างมากมาย
- **Angular**: เหมาะสำหรับโปรเจกต์ที่มีความซับซ้อนและต้องการเครื่องมือที่มีประสิทธิภาพในการจัดการกับแอปพลิเคชันขนาดใหญ่ Angular มีโครงสร้างที่เป็นระบบและมีเครื่องมือที่ช่วยในการพัฒนาอย่างครบถ้วน
- **Vue.js**: เหมาะสำหรับนักพัฒนาที่ต้องการเริ่มต้นการพัฒนา Front-End ด้วยเฟรมเวิร์กที่ง่ายต่อการเรียนรู้และใช้งาน Vue.js มีความยืดหยุ่นสูงและสามารถนำไปใช้ในโปรเจกต์ที่มีขนาดเล็กถึงขนาดกลางได้อย่างมีประสิทธิภาพ

การเลือกเฟรมเวิร์กที่เหมาะสมจะช่วยให้การพัฒนา Front-End ของคุณเป็นไปได้อย่างราบรื่นและมีประสิทธิภาพมากขึ้น

10.4 หลักการออกแบบพื้นฐานของ Front-End: การออกแบบที่ตอบสนองและการพิจารณาประสบการณ์ผู้ใช้

การออกแบบ Front-End ที่ดีไม่เพียงแต่ต้องมีความสวยงาม แต่ยังต้องคำนึงถึงการตอบสนองต่ออุปกรณ์ต่าง ๆ และการมอบประสบการณ์การใช้งานที่ดีให้กับผู้ใช้ การใช้หลักการออกแบบพื้นฐานเหล่านี้จะช่วยให้คุณสามารถสร้างเว็บแอปพลิเคชันที่มีประสิทธิภาพและน่าใช้งาน

10.4.1 การออกแบบที่ตอบสนอง (Responsive Design)

การออกแบบที่ตอบสนอง (Responsive Design) เป็นแนวคิดที่เกี่ยวข้องกับการทำให้หน้าเว็บสามารถปรับตัวได้ตามขนาดหน้าจอและอุปกรณ์ที่ผู้ใช้ใช้งาน การออกแบบที่ตอบสนองช่วยให้ผู้ใช้สามารถเข้าถึงเนื้อหาและฟังก์ชันต่าง ๆ ของเว็บแอปพลิเคชันได้อย่างเหมาะสมไม่ว่าจะใช้หน้าจอขนาดใด

แนวทางในการออกแบบที่ตอบสนอง:

- **การใช้ Grid Layout**: การใช้ Grid Layout ช่วยให้การจัดวางองค์ประกอบบนหน้าเว็บเป็นไปได้อย่างยืดหยุ่นและสามารถปรับตัวได้ตามขนาดหน้าจอ
- **การใช้ Flexbox**: Flexbox เป็นเครื่องมือที่ช่วยในการจัดการการจัดวางองค์ประกอบบนหน้าเว็บให้ตอบสนองต่อขนาดหน้าจอและการเรียงลำดับขององค์ประกอบ
- **การใช้ Media Queries**: Media Queries ช่วยให้คุณสามารถปรับแต่งสไตล์ของหน้าเว็บให้ตอบสนองต่อขนาดหน้าจอที่แตกต่างกันได้ เช่น การเปลี่ยนขนาดของฟอนต์, การซ่อนองค์ประกอบบางอย่าง, หรือการจัดเรียงองค์ประกอบใหม่

ตัวอย่างของ Media Queries:

```
@media (max-width: 600px) {
  header {
    font-size: 1.2em;
  }

  nav ul {
    flex-direction: column;
  }
}
```

Listing 10.13: ตัวอย่างของ Media Queries

ในตัวอย่างนี้ Media Queries ถูกใช้ในการปรับแต่งสไตล์ของหน้าเว็บเมื่อขนาดหน้าจอมีความกว้างไม่เกิน 600px การออกแบบที่ตอบสนองช่วยให้หน้าเว็บของคุณสามารถใช้งานได้อย่างเหมาะสมบนอุปกรณ์ต่าง ๆ

10.4.2 การพิจารณาประสบการณ์ผู้ใช้ (User Experience)

ประสบการณ์ผู้ใช้ (User Experience หรือ UX) เป็นปัจจัยสำคัญที่ส่งผลต่อความพึงพอใจของผู้ใช้ที่เข้ามาใช้งานเว็บแอปพลิเคชัน การออกแบบ UX ที่ดีช่วยให้ผู้ใช้สามารถใช้งานเว็บแอปพลิเคชันได้อย่างง่ายดายและเพลิดเพลิน

แนวทางในการออกแบบ UX ที่ดี:

- **ความเรียบง่าย**: การออกแบบ UX ที่ดีควรมีความเรียบง่ายและใช้งานง่าย ผู้ใช้ควรสามารถเข้าใจและใช้งานฟังก์ชันต่าง ๆ ได้โดยไม่ต้องมีการอธิบายมาก
- **การตอบสนองที่รวดเร็ว**: UX ที่ดีควรตอบสนองต่อการกระทำของผู้ใช้อย่างรวดเร็ว เช่น เมื่อผู้ใช้คลิกที่ปุ่ม ควรมีการตอบสนองทันที ไม่ว่าจะเป็นการเปลี่ยนแปลงสี, การโหลดข้อมูล, หรือการแสดงผลของหน้าต่างใหม่
- **การออกแบบที่ใช้งานได้กับผู้ใช้หลากหลาย**: UX ที่ดีควรคำนึงถึงผู้ใช้ที่มีความหลากหลาย ไม่ว่าจะเป็นการใช้งานบนอุปกรณ์ที่แตกต่างกัน, การเข้าถึงข้อมูลด้วยอินเทอร์เน็ตที่ช้า, หรือผู้ใช้ที่มีความบกพร่องทางการมองเห็น

ตัวอย่างของการพิจารณา UX:

```
<button onclick="showLoading()">Submit</button>

<script>
function showLoading() {
    const button = document.querySelector('button');
    button.innerHTML = 'Loading...';
    setTimeout(() => {
        button.innerHTML = 'Submit';
    }, 2000);
}
</script>
```

Listing 10.14: การเพิ่ม UX ที่ดีผ่านการตอบสนองต่อผู้ใช้

ในตัวอย่างนี้ เราได้เพิ่มการตอบสนองทันทีเมื่อผู้ใช้คลิกที่ปุ่ม โดยแสดงข้อความ `Loading...` เพื่อแจ้งให้ผู้ใช้ทราบว่ากำลังมีการดำเนินการอยู่ การตอบสนองที่รวดเร็วและชัดเจนช่วยให้ผู้ใช้รู้สึกมั่นใจในการใช้งานเว็บแอปพลิเคชัน

10.4.3 การใช้สีและการออกแบบที่เหมาะสม

การเลือกสีและการออกแบบที่เหมาะสมเป็นปัจจัยสำคัญที่ช่วยให้เว็บแอปพลิเคชันของคุณดูน่าสนใจและดึงดูดผู้ใช้ สีมีผลต่อความรู้สึกและอารมณ์ของผู้ใช้ ดังนั้นการเลือกใช้สีที่เหมาะสมสามารถช่วยเพิ่มประสบการณ์การใช้งานที่ดีขึ้นได้

แนวทางในการเลือกสี:

- **สีที่ตรงกันข้าม**: การใช้สีที่ตรงกันข้ามช่วยให้ข้อความหรือองค์ประกอบที่สำคัญโดดเด่นและดึงดูดสายตาผู้ใช้
- **การใช้สีที่มีความสอดคล้อง**: การเลือกสีที่มีความสอดคล้องช่วยให้หน้าเว็บดูเป็นระเบียบและมีความสมดุล
- **การใช้สีที่มีความหมาย**: การใช้สีที่มีความหมายช่วยให้ผู้ใช้เข้าใจการทำงานของฟังก์ชันต่าง ๆ ได้ง่ายขึ้น เช่น การใช้สีเขียวสำหรับการยืนยัน และสีแดงสำหรับการเตือนหรือข้อผิดพลาด

ตัวอย่างของการใช้สี:

```
button {
    background-color: #4CAF50; /* Green color */
    color: white;
    padding: 10px 20px;
    border: none;
    cursor: pointer;
}

button:hover {
    background-color: #45a049; /* Darker green when hovered */
}
```

Listing 10.15: Using Color to Enhance Meaning in UX

ในตัวอย่างนี้ สีเขียวถูกใช้เพื่อแสดงถึงการยืนยันและการทำงานที่สำเร็จ การเลือกใช้สีที่เหมาะสมช่วยให้ UX ของเว็บแอปพลิเคชันของคุณดูน่าสนใจและใช้งานได้อย่างมีประสิทธิภาพ

10.5 สรุปท้ายบท

บทนี้ได้นำเสนอแนวคิดพื้นฐานเกี่ยวกับการพัฒนา Front-End ซึ่งเป็นส่วนที่ผู้ใช้มองเห็นและโต้ตอบกับแอปพลิเคชันบนเว็บ เริ่มต้นจากการทำความเข้าใจบทบาทของ HTML, CSS, และ JavaScript ในการพัฒนาเว็บ และการตั้งค่าสภาพแวดล้อม Front-End ที่มีประสิทธิภาพด้วยเครื่องมือและไลบรารีเช่น npm และ Webpack

นอกจากนี้ยังมีการแนะนำเฟรมเวิร์กยอดนิยมสำหรับการพัฒนา Front-End เช่น React, Angular, และ Vue.js ซึ่งช่วยให้การพัฒนา UI เป็นไปได้อย่างราบรื่นและมีประสิทธิภาพ สุดท้ายนี้ บทนี้ได้ครอบคลุมถึงหลักการออกแบบพื้นฐานที่ช่วยให้เว็บแอปพลิเคชันของคุณตอบสนองต่ออุปกรณ์ต่าง ๆ และมอบประสบการณ์การใช้งานที่ดีให้กับผู้ใช้

คำถามทบทวน:

1. HTML, CSS, และ JavaScript มีบทบาทอย่างไรในการพัฒนา Front-End ของเว็บแอปพลิเคชัน?
2. คุณจะตั้งค่าสภาพแวดล้อม Front-End ด้วยเครื่องมือเช่น npm และ Webpack ได้อย่างไร?
3. เฟรมเวิร์ก Front-End ใดบ้างที่ได้รับความนิยม และคุณจะเลือกเฟรมเวิร์กใดในโปรเจกต์ของคุณ?
4. หลักการออกแบบที่ตอบสนองและการพิจารณาประสบการณ์ผู้ใช้มีความสำคัญอย่างไรในการพัฒนา Front-End?

การอ่านเพิ่มเติม:

- *HTML & CSS: Design and Build Websites* Duckett *[21]*
- *JavaScript: The Good Parts* Crockford *[4]*
- *Learning Web Design* Robbins *[22]*

บทที่ 11

เจาะลึก React: Exploring the Ecosystem

บทนำ

บทที่ 11 นี้จะเป็นการลงลึกในรายละเอียดของการพัฒนาเว็บแอปพลิเคชันฝั่งลูกข่าย (Front-End) โดยใช้ **React** ซึ่งเป็นหนึ่งในไลบรารี JavaScript ที่ได้รับความนิยมสูงสุดในการสร้างส่วนติดต่อผู้ใช้ (User Interface) เราจะสำรวจระบบนิเวศ (Ecosystem) ของ React อย่างละเอียด ตั้งแต่การจัดการสถานะของแอปพลิเคชันด้วย **Redux** ไปจนถึงการสร้าง Component ที่นำกลับมาใช้ซ้ำได้ (Reusable Components) และการจัดการการนำทางและการ Routing ในแอปพลิเคชันแบบหน้าเดียว (Single-Page Applications: SPAs)

วัตถุประสงค์การเรียนรู้:

1. ทำความเข้าใจระบบนิเวศของ React และวิธีการใช้เครื่องมือเสริมเพื่อเพิ่มประสิทธิภาพในการพัฒนา
2. เรียนรู้การจัดการสถานะของแอปพลิเคชันด้วย Redux
3. เข้าใจแนวคิดของ Component-Based Architecture และวิธีการสร้าง Component ที่นำกลับมาใช้ซ้ำได้
4. เรียนรู้วิธีการจัดการการนำทางและ Routing ใน SPAs ด้วย React Router

11.1 Deep Dive into React Ecosystem

React เป็นมากกว่าแค่ไลบรารีในการสร้าง UI; มันมีระบบนิเวศที่แข็งแกร่งซึ่งช่วยให้การพัฒนาเว็บแอปพลิเคชันมีความยืดหยุ่นและทรงพลังมากขึ้น โดยเครื่องมือเสริมหลายตัวช่วยเพิ่มขีดความสามารถในการพัฒนา เช่น Redux สำหรับการจัดการสถานะ, React Router สำหรับการจัดการ Routing, และอื่นๆ

11.1.1 การใช้ Create React App เพื่อเริ่มต้นโปรเจกต์

Create React App เป็นเครื่องมือที่ช่วยในการตั้งค่าโปรเจกต์ React อย่างรวดเร็ว โดยมีการตั้งค่าพื้นฐานทั้งหมดที่จำเป็นสำหรับการพัฒนาแอปพลิเคชัน React

การสร้างโปรเจกต์ด้วย Create React App:

```
npx create-react-app my-app
cd my-app
npm start
```

Listing 11.1: Create React App Example

คำสั่งนี้จะสร้างโปรเจกต์ React ใหม่และรันแอปพลิเคชันในโหมดพัฒนา

11.1.2 การสำรวจระบบนิเวศของ React

ระบบนิเวศของ React ประกอบด้วยเครื่องมือและไลบรารีหลายตัวที่ช่วยในการพัฒนาและปรับปรุงแอปพลิเคชัน React เช่น:

- **React Router** สำหรับการจัดการการนำทางและ Routing
- **Redux** สำหรับการจัดการสถานะของแอปพลิเคชัน
- **Styled Components** สำหรับการจัดการสไตล์ใน React Component
- **React Testing Library** สำหรับการทดสอบ Component ใน React

ตัวอย่างการติดตั้งไลบรารีที่สำคัญ:

```
npm install react-router-dom redux react-redux styled-components
```

Listing 11.2: Library Installation Example

การใช้ไลบรารีเหล่านี้ช่วยให้การพัฒนาแอปพลิเคชัน React มีประสิทธิภาพและยืดหยุ่นมากขึ้น

11.2 State Management: Managing Application State with Redux

การจัดการสถานะในแอปพลิเคชันที่ซับซ้อนอาจเป็นความท้าทายที่สำคัญ Redux เป็นไลบรารียอดนิยมที่ช่วยจัดการสถานะของแอปพลิเคชัน React ได้อย่างมีประสิทธิภาพ โดยทำให้การจัดการสถานะเป็นไปอย่างมีระบบและสามารถคาดการณ์ได้

11.2.1 การติดตั้งและตั้งค่า Redux

การเริ่มต้นใช้งาน Redux ในโปรเจกต์ React จำเป็นต้องติดตั้งไลบรารี Redux และ React-Redux

การติดตั้ง Redux และ React-Redux:

```
npm install redux react-redux
```

Listing 11.3: Redux Installation Example

การตั้งค่า Redux ในโปรเจกต์ React:

```
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import rootReducer from './reducers';
import App from './App';

const store = createStore(rootReducer);

ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root')
);
```

Listing 11.4: Redux Setup Example

ในตัวอย่างนี้ เราได้สร้าง Redux Store และใช้ `Provider` เพื่อเชื่อมต่อ React กับ Redux Store

11.2.2 การสร้าง Actions และ Reducers

Actions และ Reducers เป็นส่วนประกอบหลักใน Redux ที่ใช้จัดการสถานะของแอปพลิเคชัน

ตัวอย่างการสร้าง Action:

```
export const increment = () => {
    return {
        type: 'INCREMENT'
    };
};
```

Listing 11.5: Action Example

ตัวอย่างการสร้าง Reducer:

```
const counter = (state = 0, action) => {
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        default:
            return state;
    }
};

export default counter;
```

Listing 11.6: Reducer Example

ในตัวอย่างนี้ เราได้สร้าง Action **`increment`** และ Reducer **`counter`** เพื่อจัดการการเพิ่มค่าของตัวนับในแอปพลิเคชัน

11.2.3 การเชื่อมต่อ Component กับ Redux Store

หลังจากสร้าง Actions และ Reducers แล้ว เราสามารถเชื่อมต่อ Component กับ Redux Store เพื่อเข้าถึงและปรับปรุงสถานะได้

ตัวอย่างการเชื่อมต่อ Component กับ Redux Store:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment } from './actions';

function Counter() {
    const count = useSelector(state => state.counter);
    const dispatch = useDispatch();

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() =>
               dispatch(increment())}>Increment</button>
        </div>
    );
}

export default Counter;
```

Listing 11.7: Connecting Component to Redux Store Example

ในตัวอย่างนี้ เราได้ใช้ **useSelector** เพื่อเข้าถึงสถานะใน Redux Store และ **useDispatch** เพื่อส่ง Action **increment** ไปยัง Store

11.3 Component-Based Architecture: Building Reusable Components and Modular Front-End Applications

Component-Based Architecture เป็นแนวคิดที่สำคัญใน React ซึ่งช่วยให้การพัฒนาแอปพลิเคชันมีความเป็นโมดูลและสามารถนำกลับมาใช้ซ้ำได้

11.3.1 การสร้าง Reusable Components

การสร้าง Component ที่นำกลับมาใช้ซ้ำได้เป็นหนึ่งในข้อดีของการใช้ React ในการพัฒนาแอปพลิเคชัน

ตัวอย่างการสร้าง Reusable Button Component:

```
import React from 'react';

function Button({ label, onClick }) {
    return <button onClick={onClick}>{label}</button>;
}

export default Button;
```

Listing 11.8: Reusable Button Component Example

ในตัวอย่างนี้ **Button** Component สามารถนำไปใช้ซ้ำในหลายๆ ส่วนของแอปพลิเคชันได้โดยการส่ง **label** และ **onClick** เป็น Props

11.3.2 การจัดการ Component Hierarchy

การจัดการโครงสร้าง Component ที่มีลำดับชั้นซับซ้อนเป็นเรื่องสำคัญในการพัฒนาแอปพลิเคชันที่มีขนาดใหญ่

ตัวอย่างการจัดการ Component Hierarchy:

```
import React from 'react';
import Header from './Header';
import Footer from './Footer';
import MainContent from './MainContent';

function App() {
    return (
        <div>
            <Header />
            <MainContent />
            <Footer />
        </div>
    );
}

export default App;
```

Listing 11.9: Component Hierarchy Example

ในตัวอย่างนี้ เราได้สร้างโครงสร้างของแอปพลิเคชันที่ประกอบด้วย `Header`, `MainContent`, และ `Footer` Component

11.3.3 การใช้ Higher-Order Components (HOCs)

Higher-Order Components เป็นเทคนิคในการสร้าง Component ที่สามารถนำฟังก์ชันและคุณสมบัติเพิ่มเติมไปใช้กับ Component อื่นๆ ได้

ตัวอย่างการสร้าง HOC:

```
import React from 'react';

function withLogging(WrappedComponent) {
    return function WrappedWithLogging(props) {
        console.log('Component rendered with props:', props);
        return <WrappedComponent {...props} />;
    };
}

export default withLogging;
```

Listing 11.10: Higher-Order Component Example

ในตัวอย่างนี้ `withLogging` HOC จะเพิ่มฟังก์ชันการบันทึกข้อความลงใน Component ที่ถูกห่อหุ้ม (Wrapped)

11.4 Front-End Routing: Managing Navigation and Routing in Single-Page Applications (SPAs)

การจัดการการนำทางและ Routing เป็นส่วนสำคัญในการพัฒนา SPAs ซึ่งช่วยให้ผู้ใช้สามารถนำทางระหว่างหน้าต่างๆ ในแอปพลิเคชันได้อย่างราบรื่น

11.4.1 การติดตั้งและตั้งค่า React Router

React Router เป็นไลบรารียอดนิยมสำหรับการจัดการ Routing ใน React SPAs

การติดตั้ง React Router:

```
npm install react-router-dom
```

Listing 11.11: React Router Installation Example

การตั้งค่า React Router:

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Home from './Home';
import About from './About';

function App() {
    return (
        <Router>
            <Switch>
                <Route path="/about" component={About} />
                <Route path="/" component={Home} />
            </Switch>
        </Router>
    );
}

export default App;
```

Listing 11.12: React Router Setup Example

ในตัวอย่างนี้ เราได้ตั้งค่า React Router และกำหนดเส้นทางสำหรับ `/about` และ `/`

11.4.2 การจัดการ Dynamic Routing

Dynamic Routing ช่วยให้แอปพลิเคชันสามารถสร้างเส้นทางที่เปลี่ยนแปลงตามข้อมูลที่ถูกส่งเข้ามาได้

ตัวอย่างการใช้ Dynamic Routing:

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import UserProfile from './UserProfile';

function App() {
    return (
        <Router>
            <Switch>
                <Route path="/user/:id" component={UserProfile} />
            </Switch>
        </Router>
    );
}

export default App;
```

Listing 11.13: Dynamic Routing Example

ในตัวอย่างนี้ `UserProfile` Component จะถูกเรนเดอร์เมื่อเส้นทาง `/user/:id` ถูกเรียกใช้ และ `id` จะถูกส่งเป็น Props ไปยัง Component

11.4.3 การใช้ NavLink สำหรับการนำทาง

NavLink เป็น Component ที่ช่วยในการสร้างลิงก์นำทางภายในแอปพลิเคชันและสามารถกำหนดคลาสสำหรับลิงก์ที่ถูกเลือกได้

ตัวอย่างการใช้ NavLink:

```
import React from 'react';
import { NavLink } from 'react-router-dom';

function Navigation() {
    return (
        <nav>
            <NavLink exact to="/" activeClassName="active">Home</NavLink>
            <NavLink to="/about" activeClassName="active">About</NavLink>
        </nav>
    );
}

export default Navigation;
```

Listing 11.14: NavLink Example

ในตัวอย่างนี้ `NavLink` จะเพิ่มคลาส `active` ให้กับลิงก์ที่ถูกเลือกอยู่ในปัจจุบัน

สรุปท้ายบท

บทนี้ได้เจาะลึกเกี่ยวกับการพัฒนาแอปพลิเคชัน Front-End ด้วย React โดยครอบคลุมตั้งแต่การสำรวจระบบนิเวศของ React การจัดการสถานะของแอปพลิเคชันด้วย Redux การสร้าง Component ที่นำกลับมาใช้ซ้ำได้ใน Component-Based

Architecture และการจัดการการนำทางและ Routing ใน SPAs ด้วย React Router ความรู้เหล่านี้จะช่วยให้คุณสามารถพัฒนาแอปพลิเคชัน React ที่ซับซ้อนและยืดหยุ่นได้อย่างมีประสิทธิภาพ

คำถามทบทวน:

1. ระบบนิเวศของ React ประกอบด้วยเครื่องมือเสริมใดบ้างที่ช่วยเพิ่มประสิทธิภาพในการพัฒนา?
2. Redux มีบทบาทอย่างไรในการจัดการสถานะของแอปพลิเคชัน React?
3. Component-Based Architecture คืออะไร และมีประโยชน์อย่างไรในการพัฒนาเว็บแอปพลิเคชัน?
4. คุณจะจัดการการนำทางและ Routing ใน SPAs ด้วย React Router ได้อย่างไร?

การอ่านเพิ่มเติม:

- *React in Depth* Thomas *[23]*
- *Redux in Action* Garreau *[24]*
- *React Router Ready* Anwar *[25]*

บทที่ 12

การพัฒนาและปรับใช้แอปพลิเคชันแบบ Full-Stack

ในบทนี้ เราจะสำรวจขั้นตอนที่สำคัญในการพัฒนาและปรับใช้แอปพลิเคชันแบบ Full-Stack โดยครอบคลุมตั้งแต่การเชื่อมต่อ Front-End เข้ากับ Back-End, การใช้ซอฟต์แวร์คอนเทนเนอร์เช่น Docker, การปรับใช้แอปพลิเคชันไปยังแพลตฟอร์มคลาวด์เช่น AWS, Azure, หรือ Heroku และการปฏิบัติที่ดีที่สุดในการพัฒนา Full-Stack รวมถึงโครงการสุดท้ายที่รวมเอาความรู้ทั้งหมดที่คุณได้เรียนรู้ตลอดหลักสูตร

12.1 การเชื่อมต่อ Front-End เข้ากับ Back-End

การพัฒนาแอปพลิเคชันแบบ Full-Stack ไม่ได้จำกัดเพียงแค่การพัฒนา Front-End หรือ Back-End เท่านั้น แต่ยังรวมถึงการเชื่อมต่อทั้งสองส่วนเข้าด้วยกันอย่างราบรื่น ในขั้นตอนนี้ คุณจะได้เรียนรู้วิธีการเชื่อมต่อ Front-End ของคุณกับ Back-End ที่พัฒนาโดยใช้ Node.js

12.1.1 การสร้าง Node.js APIs

ก่อนที่จะเชื่อมต่อ Front-End เข้ากับ Back-End คุณจำเป็นต้องสร้าง API ที่จะใช้ในการสื่อสารระหว่างสองส่วนนี้ Node.js เป็นเครื่องมือที่ยอดเยี่ยมสำหรับการสร้าง APIs โดยใช้เฟรมเวิร์กเช่น Express.js คุณสามารถสร้างเส้นทาง API เพื่อจัดการกับคำร้องขอจาก Front-End ได้อย่างง่ายดาย

ตัวอย่างของการสร้าง API ด้วย Node.js:

```
const express = require('express');
const app = express();
app.use(express.json());

let todos = [
    { id: 1, task: 'Learn JavaScript', completed: false },
    { id: 2, task: 'Learn Node.js', completed: false }
];

// GET: Retrieve all todo items
app.get('/api/todos', (req, res) => {
    res.send(todos);
});

// POST: Add a new todo item
app.post('/api/todos', (req, res) => {
    const newTodo = {
        id: todos.length + 1,
        task: req.body.task,
        completed: false
    };
    todos.push(newTodo);
    res.status(201).send(newTodo);
});

// PUT: Update the completion status of a todo item
app.put('/api/todos/:id', (req, res) => {
    const todo = todos.find(t => t.id === parseInt(req.params.id));
    if (!todo) return res.status(404).send('Todo not found');
    todo.completed = req.body.completed;
    res.send(todo);
});

// DELETE: Delete a todo item
app.delete('/api/todos/:id', (req, res) => {
    const todoIndex = todos.findIndex(t => t.id ===
       parseInt(req.params.id));
    if (todoIndex === -1) return res.status(404).send('Todo not found');
    const deletedTodo = todos.splice(todoIndex, 1);
    res.send(deletedTodo);
});

const port = process.env.PORT || 3000;
app.listen(port, () => console.log(`Server running on port ${port}`));
```

Listing 12.1: Creating an API with Node.js

ในตัวอย่างนี้ เราได้สร้าง API สำหรับการจัดการรายการงาน (todos) ซึ่งประกอบด้วยเส้นทางสำหรับการดึง, เพิ่ม, อัปเดต, และลบรายการงานเหล่านี้ API เหล่านี้จะเป็นตัวกลางที่ช่วยให้ Front-End สามารถสื่อสารกับ Back-End ได้

12.1.2 การเชื่อมต่อ Front-End กับ Node.js APIs

หลังจากที่คุณสร้าง API สำหรับ Back-End ของคุณแล้ว ขั้นตอนถัดไปคือการเชื่อมต่อ Front-End เข้ากับ APIs เหล่านี้ คุณสามารถใช้ JavaScript (หรือไลบรารีเช่น Axios หรือ Fetch API) เพื่อส่งคำร้องขอไปยัง APIs และแสดงผลข้อมูลที่ได้รับบนหน้าเว็บ

ตัวอย่างของการเชื่อมต่อ Front-End กับ Node.js APIs:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Todo App</title>
    <script>
        async function fetchTodos() {
            const response = await fetch('/api/todos');
            const todos = await response.json();
            const todoList = document.getElementById('todo-list');
            todoList.innerHTML = '';
            todos.forEach(todo => {
                const li = document.createElement('li');
                li.textContent = todo.task + (todo.completed ? '
                    (Completed)' : '');
                todoList.appendChild(li);
            });
        }

        async function addTodo() {
            const taskInput = document.getElementById('task-input');
            const newTask = taskInput.value;
            const response = await fetch('/api/todos', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({ task: newTask })
            });
            const todo = await response.json();
            fetchTodos();
            taskInput.value = '';
        }

        window.onload = fetchTodos;
    </script>
</head>
<body>
    <h1>Todo App</h1>
    <input type="text" id="task-input" placeholder="Add a new task">
    <button onclick="addTodo()">Add Task</button>
    <ul id="todo-list"></ul>
</body>
</html>
```

Listing 12.2: การเชื่อมต่อ Front-End กับ Node.js APIs

ในตัวอย่างนี้ เราได้ใช้ Fetch API ใน JavaScript เพื่อดึงข้อมูลจาก API ที่สร้างไว้ใน Back-End จากนั้นแสดงผลรายการงานบนหน้าเว็บ นอกจากนี้เรายังสามารถเพิ่มรายการงานใหม่ผ่านอินพุตฟอร์มบนหน้าเว็บ ซึ่งข้อมูลจะถูกส่งไปยัง API ใน Back-End เพื่อบันทึกลงในฐานข้อมูล

การเชื่อมต่อ Front-End กับ Back-End ผ่าน APIs เป็นขั้นตอนสำคัญในการพัฒนาแอปพลิเคชันแบบ Full-Stack ที่สามารถสื่อสารและแลกเปลี่ยนข้อมูลได้อย่างราบรื่น

12.2 บทนำสู่ซอฟต์แวร์คอนเทนเนอร์: การใช้ Docker ในการคอนเทนเนอร์แอปพลิเคชัน

ในยุคของการพัฒนาแอปพลิเคชันแบบ Full-Stack การใช้ซอฟต์แวร์คอนเทนเนอร์เช่น Docker เป็นสิ่งที่สำคัญอย่างยิ่ง Docker ช่วยให้คุณสามารถคอนเทนเนอร์แอปพลิเคชันของคุณทั้งหมด รวมถึงโค้ด, ไลบรารี, และการตั้งค่าทั้งหมดในไฟล์เดียว ซึ่งสามารถปรับใช้ได้ง่ายและรวดเร็วบนเซิร์ฟเวอร์ใด ๆ ที่รองรับ Docker

12.2.1 Docker คืออะไร?

Docker เป็นแพลตฟอร์มซอฟต์แวร์ที่ใช้สำหรับการสร้าง, ทดสอบ, และปรับใช้แอปพลิเคชันในสภาพแวดล้อมที่คอนเทนเนอร์ (containerized environment) Docker ช่วยให้คุณสามารถบรรจุแอปพลิเคชันและส่วนประกอบต่าง ๆ ไว้ในคอนเทนเนอร์ที่สามารถย้ายไปยังระบบอื่นได้ง่าย ๆ โดยไม่ต้องกังวลเกี่ยวกับความเข้ากันได้ของระบบปฏิบัติการหรือการตั้งค่าของระบบ

12.2.2 การสร้าง Dockerfile

ในการใช้ Docker คุณจำเป็นต้องสร้างไฟล์ที่เรียกว่า `Dockerfile` ซึ่งระบุขั้นตอนในการสร้างคอนเทนเนอร์สำหรับแอปพลิเคชันของคุณ `Dockerfile` จะประกอบด้วยคำสั่งต่าง ๆ ที่บอก Docker ว่าต้องทำอะไรบ้าง เช่น การติดตั้งแพ็กเกจ, การคัดลอกไฟล์, และการตั้งค่าเซิร์ฟเวอร์

ตัวอย่างของ Dockerfile:

```
# Use Node.js as the base image
FROM node:14

# Set the working directory for the application
WORKDIR /usr/src/app

# Copy the package.json and package-lock.json files
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the application files to the container
COPY . .

# Expose the port the server will listen on
EXPOSE 3000

# Command to run the application
CMD ["node", "index.js"]
```

Listing 12.3: Creating a Dockerfile for a Node.js Application

ในตัวอย่างนี้ `Dockerfile` ถูกใช้ในการสร้างคอนเทนเนอร์สำหรับแอปพลิเคชัน Node.js โดยเริ่มต้นจากการใช้ Node.js เป็นฐาน (base image) จากนั้นคัดลอกไฟล์ที่จำเป็นและติดตั้ง dependencies ก่อนที่จะรันแอปพลิเคชันในคอนเทนเนอร์

12.2.3 การสร้างและรัน Docker Container

หลังจากที่คุณสร้าง `Dockerfile` เรียบร้อยแล้ว คุณสามารถสร้างและรันคอนเทนเนอร์สำหรับแอปพลิเคชันของคุณได้ด้วยคำสั่ง Docker

ขั้นตอนในการสร้างและรัน Docker Container:

1. การสร้าง Docker Image:

```
    docker build -t my-node-app .
```

2. การรัน Docker Container:

```
    docker run -p 3000:3000 my-node-app
```

ในตัวอย่างนี้ `docker build` ถูกใช้ในการสร้าง Docker Image จาก `Dockerfile` ที่คุณสร้างไว้ และ `docker run` ถูกใช้ในการรันคอนเทนเนอร์ที่สร้างขึ้นโดยเชื่อมต่อพอร์ต 3000 ของคอนเทนเนอร์กับพอร์ต 3000 ของเครื่องโฮสต์

การใช้ Docker ช่วยให้การปรับใช้แอปพลิเคชันเป็นไปได้อย่างง่ายดายและมีประสิทธิภาพ เนื่องจากคุณสามารถสร้างคอนเทนเนอร์ที่รวมเอาทุกส่วนประกอบของแอปพลิเคชันไว้ในไฟล์เดียว

12.3 การปรับใช้แอปพลิเคชันในคลาวด์

การปรับใช้แอปพลิเคชันไปยังแพลตฟอร์มคลาวด์เป็นขั้นตอนสำคัญในการทำให้แอปพลิเคชันของคุณสามารถเข้าถึงได้จากทุกที่และสามารถขยายขนาดได้ตามความต้องการ แพลตฟอร์มคลาวด์เช่น AWS, Azure, และ Heroku เป็นทางเลือกที่ยอดเยี่ยมสำหรับการปรับใช้แอปพลิเคชันแบบ Full-Stack

12.3.1 การปรับใช้แอปพลิเคชันใน AWS

AWS (Amazon Web Services) เป็นหนึ่งในแพลตฟอร์มคลาวด์ที่ได้รับความนิยมมากที่สุดในการปรับใช้แอปพลิเคชัน AWS มีบริการมากมายที่ช่วยให้การปรับใช้แอปพลิเคชันเป็นไปได้อย่างราบรื่น เช่น EC2 (Elastic Compute Cloud), S3 (Simple Storage Service), และ RDS (Relational Database Service)

การปรับใช้แอปพลิเคชันใน AWS EC2:

1. การสร้าง EC2 Instance:

    - เข้าสู่ AWS Management Console และเลือกบริการ EC2
    - เลือกสร้าง Instance ใหม่และเลือก AMI (Amazon Machine Image) ที่คุณต้องการใช้ เช่น Amazon Linux 2
    - กำหนดขนาดของ Instance และการตั้งค่าเครือข่าย
    - เลือกหรือสร้าง Key Pair สำหรับการเข้าถึง Instance ผ่าน SSH

2. การเชื่อมต่อกับ EC2 Instance:

```
    ssh -i "my-key.pem"
        ec2-user@ec2-xx-xx-xx-xx.compute-1.amazonaws.com
```

3. การติดตั้ง Node.js และการปรับใช้แอปพลิเคชัน:

```
    sudo yum update -y
    sudo yum install -y nodejs npm
    git clone https://github.com/username/my-node-app.git
    cd my-node-app
    npm install
    npm start
```

ในตัวอย่างนี้ คุณได้สร้าง EC2 Instance ใน AWS และปรับใช้แอปพลิเคชัน Node.js โดยการติดตั้ง Node.js และโคลนโปรเจกต์จาก GitHub

12.3.2 การปรับใช้แอปพลิเคชันใน Azure

Microsoft Azure เป็นแพลตฟอร์มคลาวด์ที่ได้รับความนิยมอีกแห่งหนึ่ง ซึ่งมีบริการที่หลากหลายสำหรับการปรับใช้และจัดการแอปพลิเคชันแบบ Full-Stack

การปรับใช้แอปพลิเคชันใน Azure App Service:

1. การสร้าง Azure App Service:
   - เข้าสู่ Azure Portal และเลือกสร้าง App Service
   - ระบุชื่อแอปพลิเคชันและเลือก Runtime Stack เช่น Node.js
   - กำหนดแผนการใช้งานและการตั้งค่าอื่น ๆ
2. การปรับใช้แอปพลิเคชันไปยัง Azure App Service:

```
az webapp up --name my-node-app --resource-group myResourceGroup
    --plan myAppServicePlan
```

3. การตรวจสอบสถานะการปรับใช้:
   - เข้าสู่ Azure Portal และตรวจสอบสถานะของ App Service ที่คุณสร้าง
   - หากการปรับใช้สำเร็จ คุณสามารถเข้าถึงแอปพลิเคชันผ่าน URL ที่ Azure ให้มา

Azure ช่วยให้การปรับใช้แอปพลิเคชันเป็นไปได้อย่างง่ายดาย และมีเครื่องมือที่ช่วยในการจัดการและตรวจสอบแอปพลิเคชันอย่างครบถ้วน

12.3.3 การปรับใช้แอปพลิเคชันใน Heroku

Heroku เป็นแพลตฟอร์มคลาวด์ที่เน้นความเรียบง่ายและการปรับใช้ที่รวดเร็ว Heroku ช่วยให้นักพัฒนาสามารถปรับใช้แอปพลิเคชันได้ง่าย ๆ โดยไม่ต้องกังวลเกี่ยวกับการตั้งค่าเซิร์ฟเวอร์

การปรับใช้แอปพลิเคชันใน Heroku:

1. การติดตั้ง Heroku CLI:

```
curl https://cli-assets.heroku.com/install.sh | sh
```

2. การล็อกอินเข้าสู่ Heroku:

```
heroku login
```

3. การสร้างแอปพลิเคชันใน Heroku:

```
heroku create my-node-app
```

4. การปรับใช้แอปพลิเคชันไปยัง Heroku:

```
git push heroku main
```

5. การเปิดแอปพลิเคชัน:

```
heroku open
```

ในตัวอย่างนี้ Heroku ถูกใช้ในการปรับใช้แอปพลิเคชัน Node.js โดยใช้คำสั่ง `git push` เพื่อปรับใช้โค้ดไปยังเซิร์ฟเวอร์ของ Heroku Heroku ช่วยให้นักพัฒนาสามารถปรับใช้แอปพลิเคชันได้อย่างรวดเร็วและง่ายดาย

12.4 แนวทางปฏิบัติที่ดีที่สุดและโครงการสุดท้าย ในการพัฒนา Full-Stack

การพัฒนาแอปพลิเคชันแบบ Full-Stack มีความท้าทายหลายประการ แต่การใช้แนวทางปฏิบัติที่ดีที่สุดจะช่วยให้การพัฒนาและการปรับใช้เป็นไปได้อย่างราบรื่นและมีประสิทธิภาพ

12.4.1 แนวทางปฏิบัติที่ดีที่สุดในการพัฒนา Full-Stack

- **การใช้เวอร์ชันคอนโทรล (Version Control)**: การใช้ Git หรือระบบเวอร์ชันคอนโทรลอื่น ๆ ช่วยให้คุณสามารถติดตามการเปลี่ยนแปลงของโค้ดและทำงานร่วมกับทีมพัฒนาได้อย่างมีประสิทธิภาพ
- **การทดสอบอัตโนมัติ (Automated Testing)**: การทดสอบอัตโนมัติช่วยให้คุณสามารถตรวจสอบความถูกต้องของโค้ดและลดความผิดพลาดในการพัฒนาได้
- **การใช้งาน CI/CD (Continuous Integration/Continuous Deployment)**: การตั้งค่า CI/CD ช่วยให้การปรับใช้แอปพลิเคชันเป็นไปได้อย่างรวดเร็วและปลอดภัย
- **การรักษาความปลอดภัย (Security)**: การรักษาความปลอดภัยของแอปพลิเคชันเป็นสิ่งสำคัญ ควรมีการตรวจสอบช่องโหว่และการป้องกันการโจมตีอย่างต่อเนื่อง
- **การเพิ่มประสิทธิภาพ (Performance Optimization)**: การเพิ่มประสิทธิภาพของแอปพลิเคชันช่วยให้ผู้ใช้ได้รับประสบการณ์ที่ดีขึ้น ควรตรวจสอบและปรับปรุงประสิทธิภาพอย่างสม่ำเสมอ

12.4.2 โครงการสุดท้าย: การพัฒนาแอปพลิเคชันแบบ Full-Stack

ในโครงการสุดท้ายนี้ คุณจะได้ใช้ความรู้ทั้งหมดที่คุณได้เรียนรู้ในหลักสูตรนี้ในการพัฒนาแอปพลิเคชันแบบ Full-Stack ที่สมบูรณ์ โครงการนี้จะประกอบด้วยการพัฒนา Front-End ที่เชื่อมต่อกับ Back-End ผ่าน APIs, การใช้ Docker ในการคอนเทนเนอร์แอปพลิเคชัน, และการปรับใช้แอปพลิเคชันไปยังแพลตฟอร์มคลาวด์ที่คุณเลือก

ขั้นตอนในการดำเนินโครงการสุดท้าย:

1. **การวางแผนโครงการ**: กำหนดวัตถุประสงค์ของแอปพลิเคชันและออกแบบสถาปัตยกรรม Full-Stack ที่เหมาะสม
2. **การพัฒนา Front-End และ Back-End**: พัฒนา UI สำหรับผู้ใช้และสร้าง APIs ที่จำเป็นสำหรับการสื่อสารระหว่าง Front-End และ Back-End
3. **การคอนเทนเนอร์แอปพลิเคชันด้วย Docker**: สร้าง `Dockerfile` และใช้ Docker ในการคอนเทนเนอร์แอปพลิเคชันทั้งหมด
4. **การปรับใช้ไปยังคลาวด์**: เลือกแพลตฟอร์มคลาวด์ที่คุณต้องการใช้ เช่น AWS, Azure, หรือ Heroku และปรับใช้แอปพลิเคชันไปยังคลาวด์
5. **การทดสอบและการปรับปรุง**: ทดสอบแอปพลิเคชันอย่างละเอียดและปรับปรุงตามผลการทดสอบเพื่อให้แอปพลิเคชันทำงานได้อย่างมีประสิทธิภาพ

ผลลัพธ์ที่คาดหวังจากโครงการสุดท้าย:

- การพัฒนาแอปพลิเคชันแบบ Full-Stack ที่สามารถใช้งานได้จริงและพร้อมสำหรับการปรับใช้
- ความเข้าใจที่ลึกซึ้งในการพัฒนาและการปรับใช้แอปพลิเคชันในสภาพแวดล้อมคลาวด์
- การฝึกฝนการใช้ Docker และแพลตฟอร์มคลาวด์ในการปรับใช้แอปพลิเคชัน

สรุปท้ายบท

บทนี้ได้เสนอแนวทางในการพัฒนาและปรับใช้แอปพลิเคชันแบบ Full-Stack โดยเริ่มต้นจากการเชื่อมต่อ Front-End เข้ากับ Back-End ผ่าน APIs, การใช้ซอฟต์แวร์คอนเทนเนอร์เช่น Docker ในการคอนเทนเนอร์แอปพลิเคชัน, และการปรับใช้แอปพลิเคชันไปยังแพลตฟอร์มคลาวด์เช่น AWS, Azure, หรือ Heroku นอกจากนี้ยังได้ครอบคลุมถึงแนวทางปฏิบัติที่ดีที่สุดในการพัฒนา Full-Stack และโครงการสุดท้ายที่เป็นการรวมความรู้ทั้งหมดที่คุณได้เรียนรู้ในหลักสูตรนี้

คำถามทบทวน:

1. การเชื่อมต่อ Front-End เข้ากับ Back-End ผ่าน APIs มีขั้นตอนอย่างไรบ้าง?
2. Docker คืออะไร และคุณจะใช้ Docker ในการคอนเทนเนอร์แอปพลิเคชันได้อย่างไร?
3. คุณจะปรับใช้แอปพลิเคชันไปยังแพลตฟอร์มคลาวด์เช่น AWS, Azure, หรือ Heroku ได้อย่างไร?
4. แนวทางปฏิบัติที่ดีที่สุดในการพัฒนาแอปพลิเคชันแบบ Full-Stack คืออะไร?

การอ่านเพิ่มเติม:

- *Docker in Action, Second Edition* Nickoloff and Kuenzli [26]
- *Node.js Design Patterns* Casanova and Mammino [8]
- *JavaScript from Frontend to Backend* Sarrion [27]

บรรณานุกรม

[1] David Flanagan. *JavaScript: The Definitive Guide*. 7th. Sebastopol, CA: O'Reilly Media, 2016. ISBN: 9781491952016. URL: https://www.oreilly.com/library/view/javascript-the-definitive/9781491952016/.

[2] Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. 3rd. San Francisco, CA: No Starch Press, 2018. ISBN: 9781593279509. URL: https://eloquentjavascript.net/.

[3] MDN contributors. *JavaScript | MDN*. Accessed: 2023-08-14. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[4] Douglas Crockford. *JavaScript: The Good Parts*. Accessed: 2024-08-17. O'Reilly Media, 2008. ISBN: 978-0596517748. URL: https://www.google.co.th/books/edition/JavaScript_The_Good_Parts/PXa2bby0oQ0C?hl=en&gbpv=0.

[5] Kyle Simpson. *You Don't Know JS*. Accessed: 2024-08-17. 2015. URL: https://github.com/getify/You-Dont-Know-JS.

[6] Node.js Documentation Team. *Node.js API Documentation*. Accessed: 2024-08-17. 2024. URL: https://nodejs.org/docs/latest/api/.

[7] Express.js Documentation Team. *Express.js Documentation*. Accessed: 2024-08-17. 2024. URL: https://expressjs.com/.

[8] Mario Casanova and Luciano Mammino. *Node.js Design Patterns*. Third Edition. Accessed: 2024-08-17. O'Reilly Media, 2020. ISBN: 978-1839214110. URL: https://www.oreilly.com/library/view/nodejs-design-patterns/9781839214110/.

[9] David Flanagan. *JavaScript: The Definitive Guide*. 6th Edition. Accessed: 2024-08-17. O'Reilly Media, 2011. ISBN: 978-0596805524. URL: https://www.amazon.com/JavaScript-Definitive-Guide-Activate-Guides/dp/0596805527.

[10] Trevor Burnham. *Async JavaScript*. Accessed: 2024-08-17. Pragmatic Bookshelf, 2012. ISBN: 978-1937785079. URL: https://www.goodreads.com/book/show/13570848-async-javascript.

[11] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. Accessed: 2024-08-17. O'Reilly Media, 2013. ISBN: 978-1449359713. URL: https://www.oreilly.com/library/view/restful-web-apis/9781449359713/.

[12] Jon Duckett. *JavaScript and jQuery: Interactive Front-End Web Development*. Accessed: 2024-08-17. Wiley, 2014. ISBN: 978-1118531648. URL: https://www.oreilly.com/library/view/javascript-and-jquery/9781118531648/.

[13] Caio Ribeiro Pereira. *Building APIs with Node.js*. Accessed: 2024-08-17. Leanpub, 2016. ISBN: 978-1535488006. URL: https://www.google.co.th/books/edition/Building_APIs_with_Node_js/_i2yDQAAQBAJ?hl=en&gbpv=0.

[14] Sequelize Documentation Team. *Sequelize Documentation*. Accessed: 2024-08-17. 2024. URL: https://sequelize.org/.

[15] Knex.js Documentation Team. *Knex.js Documentation*. Accessed: 2024-08-17. 2024. URL: https://knexjs.org/.

[16] Mark O'Donovan. *SQL for Everyone*. Accessed: 2024-08-17. Mark O'Donovan, 2014. ISBN: 978-1494357324. URL: https://www.google.co.th/books/edition/SQL_for_Everyone/WfzxCQAAQBAJ?hl=en&gbpv=0.

[17] Andreas Meier and Michael Kaufmann. *SQL & NoSQL Databases*. Accessed: 2024-08-17. Springer Vieweg, 2019. ISBN: 978-3658253507. DOI: 10.1007/978-3-658-25351-6. URL: https://www.google.co.th/books/edition/SQL_NoSQL_Databases/XOCgDwAAQBAJ?hl=en&gbpv=0.

[18] MongoDB Documentation Team. *MongoDB Documentation*. Accessed: 2024-08-17. 2024. URL: https://www.mongodb.com/docs/.

[19] Mongoose.js Documentation Team. *Mongoose.js Documentation*. Accessed: 2024-08-17. 2024. URL: https://mongoosejs.com/.

[20] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Accessed: 2024-08-17. Addison-Wesley, 2012. ISBN: 978-0321826626. URL: https://www.google.co.th/books/edition/NoSQL_Distilled/tYhsAQAAQBAJ?hl=en.

[21] Jon Duckett. *HTML and CSS: Design and Build Websites*. Accessed: 2024-08-17. Wiley, 2011. ISBN: 978-1118008188. URL: https://www.google.co.th/books/edition/HTML_and_CSS/aGjaBTbT0o0C?hl=en&gbpv=0.

[22] Jennifer Robbins. *Learning Web Design*. 5th Edition. Accessed: 2024-08-17. O'Reilly Media, 2018. ISBN: 978-1491960206. URL: https://www.google.co.th/books/edition/Learning_Web_Design/UMFeDwAAQBAJ?hl=en&gbpv=0.

[23] Mark Tielens Thomas. *React in Depth*. Accessed: 2024-08-17. Packt Publishing, 2019. ISBN: 978-1788832821. URL: https://www.google.co.th/books/edition/React_in_Depth/C9cVEQAAQBAJ?hl=en.

[24] Marc Garreau. *Redux in Action*. Accessed: 2024-08-17. Manning Publications, 2021. ISBN: 978-1617296855. URL: https://www.google.co.th/books/edition/Redux_in_Action/CTgzEAAAQBAJ?hl=en&gbpv=0.

[25] Haseeb Anwar. *React Router Ready*. Accessed: 2024-08-17. Independently Published, 2023. ISBN: 979-8395928697. URL: https://www.google.co.th/books/edition/React_Router_Ready/jh8X0AEACAAJ?hl=en.

[26] Jeffrey Nickoloff and Stephen Kuenzli. *Docker in Action, Second Edition*. Accessed: 2024-08-17. Manning Publications, 2019. ISBN: 978-1617294769. URL: https://www.google.co.th/books/edition/Docker_in_Action_Second_Edition/qzozEAAAQBAJ?hl=en&gbpv=0.

[27] Eric Sarrion. *JavaScript from Frontend to Backend*. Accessed: 2024-08-17. O'Reilly Media, 2022. ISBN: 978-1098119511. URL: https://www.google.co.th/books/edition/JavaScript_from_Frontend_to_Backend/qOV5EAAAQBAJ?hl=en&gbpv=0.

อภิธานศัพท์

Angular แพลตฟอร์มและเฟรมเวิร์กสำหรับการสร้างแอปพลิเคชันไคลเอ็นต์แบบหน้าเดียวโดยใช้ HTML และ TypeScript ซึ่งนำโดยทีม Angular ของ Google.

API Application Programming Interface ชุด ของ กิจวัตร โปรโตคอล และ เครื่อง มือ สำหรับ การ สร้าง ซอฟต์แวร์ และ แอปพลิเคชัน.

Arrow Function ไวยากรณ์ที่กระชับสำหรับการเขียนฟังก์ชันที่แนะนำใน ECMAScript 2015 (ES6) โดยมีค่าของ `this` ถูกผูกไว้อย่างเป็นรูปธรรม.

Async/Await คุณสมบัติไวยากรณ์ใน JavaScript สำหรับการทำงานกับพรอมิส ซึ่งช่วยให้สามารถเขียนโค้ดแบบอะซิงโครนัสในลักษณะเชิงซิงโครนัส.

Asynchronous Programming รูปแบบการเขียนโปรแกรมที่อนุญาตให้หน่วยงานทำงานแยกต่างหากจากเธรดหลักของแอปพลิเคชันและแจ้งเตือนเธรดหลักเมื่อการทำงานเสร็จสิ้น มักจะทำได้โดยใช้คอลแบ็ก, พรอมิส และ async/await ใน JavaScript.

Babel คอมไพเลอร์ JavaScript ที่ใช้หลักในการแปลงโค้ด ECMAScript 2015+ ให้เป็นเวอร์ชัน JavaScript ที่เข้ากันได้ย้อนหลังสำหรับเบราว์เซอร์หรือสภาพแวดล้อมที่เก่ากว่า.

Callback ฟังก์ชันที่ส่งผ่านไปยังฟังก์ชันอื่นเป็นอาร์กิวเมนต์ซึ่งจะถูกเรียกใช้ภายในฟังก์ชันภายนอกเพื่อดำเนินการบางอย่าง.

Continuous Integration แนวทางการพัฒนาที่ต้องการให้นักพัฒนารวมโค้ดเข้ากับที่เก็บข้อมูลที่ใช้ร่วมกันหลายครั้งต่อวัน.

CRUD ตัวย่อที่ย่อมาจาก Create, Read, Update และ Delete หมายถึงการดำเนินการพื้นฐานทั้งสี่ในการจัดการข้อมูลในฐานข้อมูล.

CSS Cascading Style Sheets ภาษาสไตล์ชีตที่ใช้สำหรับอธิบายการนำเสนอเอกสารที่เขียนด้วย HTML หรือ XML.

Docker ชุดผลิตภัณฑ์แพลตฟอร์มเป็นบริการที่ใช้การจำลองระดับระบบปฏิบัติการเพื่อส่งมอบซอฟต์แวร์ในแพ็กเกจที่เรียกว่าคอนเทนเนอร์.

ECMAScript สคริปต์ภาษามาตรฐานที่ JavaScript อิงจาก ซึ่งดูแลโดยสมาคมผู้ผลิตคอมพิวเตอร์ยุโรป (ECMA).

ESLint เครื่องมือวิเคราะห์โค้ดแบบคงที่สำหรับการระบุรูปแบบปัญหาที่พบในโค้ด JavaScript.

Event Loop โครงสร้างการเขียนโปรแกรมที่รอและแจกจ่ายเหตุการณ์หรือข้อความในโปรแกรม โดยหลักแล้วใช้ในการเขียนโปรแกรมแบบอะซิงโครนัสเพื่อจัดการคอลแบ็กและการดำเนินการอื่นๆ ใน JavaScript และ Node.js.

EventEmitter คลาสใน Node.js ที่อำนวยความสะดวกในการสื่อสารระหว่างวัตถุ โดยอนุญาตให้คุณผูกฟังก์ชันกับเหตุการณ์และเรียกใช้ฟังก์ชันเหล่านี้เมื่อเหตุการณ์ถูกกระจายออกมา.

Express.js เฟรมเวิร์กเว็บแอปพลิเคชัน Node.js ที่มีขนาดเล็กและยืดหยุ่น ซึ่งให้ชุดฟีเจอร์ที่แข็งแกร่งสำหรับการสร้างเว็บและแอปพลิเคชันมือถือ.

HTML Hypertext Markup Language ภาษามาตรฐานสำหรับเอกสารที่ออกแบบมาให้แสดงในเว็บเบราว์เซอร์.

HTTP Hypertext Transfer Protocol ซึ่งเป็นพื้นฐานของการสื่อสารข้อมูลสำหรับเว็บทั่วโลก.

JavaScript ภาษาโปรแกรมระดับสูงแบบไดนามิกที่เป็นหนึ่งในเทคโนโลยีหลักของเว็บ นอกเหนือจาก HTML และ CSS.

JSON JavaScript Object Notation รูปแบบการแลกเปลี่ยนข้อมูลที่มีน้ำหนักเบาซึ่งมนุษย์อ่านและเขียนได้ง่าย และเครื่องจักรสามารถแยกวิเคราะห์และสร้างได้ง่าย.

Middleware ซอฟต์แวร์ที่ทำหน้าที่เป็นสะพานเชื่อมระหว่างระบบปฏิบัติการหรือฐานข้อมูลและแอปพลิเคชัน โดยเฉพาะอย่างยิ่งในเครือข่าย ใน Express.js ฟังก์ชัน Middleware สามารถแก้ไขออบเจ็กต์ request และ response และสิ้นสุดวงจร request-response ได้.

Mocha เฟรมเวิร์กการทดสอบ JavaScript ที่มีคุณสมบัติครบถ้วนซึ่งทำงานบน Node.js และในเบราว์เซอร์ ทำให้การทดสอบแบบอะซิงโครนัสเป็นเรื่องง่าย.

MongoDB ฐานข้อมูล NoSQL ที่ได้รับความนิยมเนื่องจากความยืดหยุ่นและความสามารถในการปรับขนาด มักใช้กับ Node.js เพื่อจัดเก็บและเรียกข้อมูลในรูปแบบเอกสาร.

Mongoose ไลบรารี ODM (Object Data Modeling) สำหรับ MongoDB และ Node.js ที่ให้โซลูชันตามโครงร่างเพื่อจำลองข้อมูลแอปพลิเคชัน.

Node.js สภาพแวดล้อมรันไทม์ JavaScript แบบโอเพนซอร์ส ที่ทำงานข้ามแพลตฟอร์มและทำงานบนเอนจิน V8 เพื่อดำเนินการโค้ด JavaScript นอกเบราว์เซอร์.

NPM Node Package Manager ผู้จัดการแพ็กเกจสำหรับ JavaScript และเป็นผู้จัดการแพ็กเกจเริ่มต้นสำหรับ Node.js.

ORM เทคนิคการเขียนโปรแกรมสำหรับการแปลงข้อมูลระหว่างระบบที่ไม่สามารถใช้งานร่วมกันได้โดยใช้ภาษาการเขียนโปรแกรมเชิงวัตถุ เช่น Sequelize และ Mongoose.

Prettier ตัวจัดรูปแบบโค้ดที่มีความคิดเห็นรองรับภาษาหลายภาษาและผสานรวมกับโปรแกรมแก้ไขส่วนใหญ่.

Promise ตัวแทนของค่าที่ไม่จำเป็นต้องทราบเมื่อสร้าง Promise โดยให้คุณเชื่อมโยงตัวจัดการกับค่าความสำเร็จในที่สุดหรือเหตุผลของความล้มเหลวของการดำเนินการแบบอะซิงโครนัส.

Prototype วัตถุที่วัตถุอื่นๆ สืบทอดคุณสมบัติและเมธอดใน JavaScript สามารถเข้าถึงและแก้ไข Prototype ของวัตถุได้ผ่านเมธอด `Object.getPrototypeOf` และ `Object.setPrototypeOf` หรือคุณสมบัติ `__proto__`.

React ไลบรารี JavaScript สำหรับการสร้างส่วนต่อประสานผู้ใช้ ซึ่งดูแลโดย Facebook และชุมชนนักพัฒนาและบริษัทที่เป็นอิสระ.

Redux คอนเทนเนอร์สถานะที่คาดเดาได้สำหรับแอป JavaScript มักใช้กับ React ในการจัดการสถานะของแอปพลิเคชัน.

RESTful API API ที่ปฏิบัติตามหลักการของ Representational State Transfer (REST) ซึ่งเป็นชุดข้อจำกัดทางสถาปัตยกรรมสำหรับการออกแบบแอปพลิเคชันเครือข่าย.

Routing กระบวนการกำหนดปลายทางในเว็บเซิร์ฟเวอร์ที่ตอบสนองต่อคำขอ HTTP เฉพาะ ใน Express.js Routing หมายถึงวิธีการที่ปลายทางของแอปพลิเคชันตอบสนองต่อคำขอของไคลเอ็นต์.

Sequelize ORM แบบพรอมิสสำหรับ Node.js สำหรับฐานข้อมูล SQL รวมถึงรองรับ MySQL, PostgreSQL, SQLite และ Microsoft SQL Server.

V8 เอนจิน JavaScript และ WebAssembly แบบโอเพนซอร์สประสิทธิภาพสูงของ Google ที่เขียนด้วย C++ และใช้ในทั้ง Google Chrome และ Node.js เพื่อดำเนินการโค้ด JavaScript.

Vue.js เฟรมเวิร์ก JavaScript แบบโปรเกรสซีฟที่ใช้สำหรับสร้างส่วนต่อประสานผู้ใช้และแอปพลิเคชันหน้าเดียว.

Webpack โปรแกรมรวบรวมโมดูล JavaScript ที่นิยมใช้ในการรวมไฟล์ JavaScript เพื่อการใช้งานในเบราว์เซอร์.