แนวทางในการจัดกิจกรรมภาคปฏิบัติ

หัวข้อการเรียนรู้ :
การออกแบบโมเดลข้อมูลและการใช้ SwiftData (Data Modeling & SwiftData)

ระยะเวลาในการทำกิจกรรม: 360 นาที

วัตถุประสงค์ในการทำกิจกรรม :
เมื่อสิ้นสุดบทเรียน ผู้เรียนจะสามารถ:

1. **วิเคราะห์บริบทของปัญหาในโลกจริง และระบุองค์ประกอบของข้อมูลหลักได้อย่างถูกต้อง** โดยสามารถแยกแยะ Entity, Attribute และ Relationship จากสถานการณ์ที่กำหนด โดยไม่อ้างอิงโครงสร้าง UI หรือรายละเอียดของเทคโนโลยีที่ใช้
2. **ออกแบบ Conceptual Data Model ในระดับนามธรรมได้อย่างเป็นระบบ** โดยสร้างแบบจำลองข้อมูลที่สะท้อนโครงสร้างและความสัมพันธ์ของข้อมูลในเชิงแนวคิด โดยไม่ผูกติดกับภาษาโปรแกรม ฐานข้อมูล หรือ Framework ใด ๆ
3. **อธิบายความแตกต่างระหว่าง In-memory Model และ Persistent Model ได้อย่างถูกต้องในเชิงแนวคิด** โดยสามารถเชื่อมโยงความแตกต่างดังกล่าวกับแนวคิดเรื่อง lifecycle, identity และบทบาทของข้อมูลในระดับระบบ
4. **ยกระดับ Data Model จาก struct ไปสู่ SwiftData @Model ได้อย่างมีเหตุผลเชิงสถาปัตยกรรม** โดยสามารถอธิบายเหตุผลของการใช้ class แทน struct ในบริบทของ SwiftData และอธิบายบทบาทของ Persistent Domain Model ในการจัดการข้อมูลของแอปพลิเคชัน

ความคิดรวบยอด:

กิจกรรมการเรียนรู้ชุดนี้มุ่งพัฒนาความรู้และความเข้าใจของผู้เรียนเกี่ยวกับการออกแบบและพัฒนาแอปพลิเคชันบนระบบปฏิบัติการ iOS โดยยึด “ข้อมูล” เป็นศูนย์กลางของกระบวนการคิดและการพัฒนามากกว่าการเริ่มต้นจากส่วนติดต่อผู้ใช้หรือการเขียนโค้ดเชิงโต้ตอบเพียงอย่างเดียว แนวทางดังกล่าวตั้งอยู่บนกรอบแนวคิด Thinking in Data ซึ่งเน้นการทำความเข้าใจโครงสร้างของข้อมูล ความสัมพันธ์ของข้อมูล และบทบาทของข้อมูลในระดับสถาปัตยกรรม ก่อนนำไปสู่การพัฒนาเชิงเทคนิคด้วยภาษา Swift, SwiftUI และ SwiftData ภายใต้สถาปัตยกรรม MVVM

บทความสำหรับอ่านประกอบการทำกิจกรรม
https://ajthiti.gitbook.io/develop-in-swift/swiftdata/thinking-in-data

แนวทางในการจัดกิจกรรมการเรียนรู้

ขั้นนำเข้าสู่บทเรียน (Warm-up)

ขั้นนำเข้าสู่บทเรียนในหัวข้อนี้มีบทบาทสำคัญในการเปลี่ยนกรอบความคิดเดิมของผู้เรียนเกี่ยวกับการพัฒนาแอปพลิเคชันและนำพาผู้เรียนไปสู่มุมมองใหม่ที่ยึดข้อมูลเป็นศูนย์กลาง การดำเนินกิจกรรมในช่วงนี้จึงไม่ควรเริ่มจากการบรรยายทฤษฎีหรือเปิดโปรแกรมพัฒนาแอปทันที แต่ควรเริ่มจากการตั้งคำถามเชิงสถานการณ์ที่ใกล้ตัวและเปิดพื้นที่ให้ผู้เรียนได้แสดงกระบวนการคิดของตนเองอย่างเป็นธรรมชาติ

ผู้สอนสามารถเริ่มต้นด้วยการยกตัวอย่างสถานการณ์ง่าย ๆ เช่น **“หากเราต้องการพัฒนาแอปบันทึกรายจ่ายสำหรับนักศึกษา แอปนี้ควรเป็นอย่างไร”** คำถามนี้ตั้งใจให้ผู้เรียนตอบอย่างอิสระ โดยไม่กำหนดกรอบล่วงหน้า ผู้สอนควรจดคำตอบทั้งหมดไว้บนกระดานโดยไม่ประเมินหรือวิจารณ์ทันที โดยทั่วไปคำตอบที่ได้มักเกี่ยวข้อง

กับหน้าจอ ปุ่ม หรือองค์ประกอบของส่วนติดต่อผู้ใช้ เช่น หน้ารายการรายจ่าย ปุ่มเพิ่มข้อมูล หน้าสรุปยอด หรือหน้าจอเลือกหมวดหมู่

เมื่อคำตอบส่วนใหญ่สะท้อนถึงการคิดเชิงการออกแบบ UI ผู้สอนจึงค่อยๆ ตั้งคำถามต่อยอดในลักษณะที่เปลี่ยนทิศทางของการคิด เช่น **“ระบบนี้ต้องจัดการข้อมูลอะไรบ้าง”** หรือ **“ข้อมูลหนึ่งรายการต้องประกอบด้วยอะไรจึงจะสมบูรณ์”** คำถามเหล่านี้จะทำให้ผู้เรียนเริ่มเปลี่ยนจากการคิดถึงสิ่งที่มองเห็นบนหน้าจอไปสู่การคิดถึงสิ่งที่ระบบต้องรับผิดชอบในระดับโครงสร้าง ผู้เรียนจะเริ่มพูดถึงข้อมูลรายจ่าย จำนวนเงิน วันที่ หมวดหมู่ หรือความสัมพันธ์ระหว่างรายการต่างๆ ซึ่งเป็นจุดที่ผู้สอนสามารถชี้ให้เห็นอย่างชัดเจนว่า ขณะนี้ทั้งชั้นเรียนกำลังออกแบบ “โครงสร้างของระบบ” โดยยังไม่ได้กล่าวถึงเทคโนโลยีใดเลย **ในขั้นตอนนี้ ผู้สอนไม่ควรรีบสรุปเชิงทฤษฎี แต่ควรใช้วิธีสะท้อนความคิดกลับไปยังผู้เรียน เช่น การชี้ให้ผู้เรียนเห็นว่า ช่วงแรกทุกคนคิดถึงหน้าจอ แต่เมื่อเปลี่ยนคำถาม ทุกคนเริ่มพูดถึงข้อมูล** การสะท้อนเช่นนี้ช่วยให้ผู้เรียนตระหนักด้วยตนเองถึงความแตกต่างระหว่างการคิดแบบ UI-Driven กับ Data-Driven ซึ่งเป็นแก่นสำคัญของบทเรียน

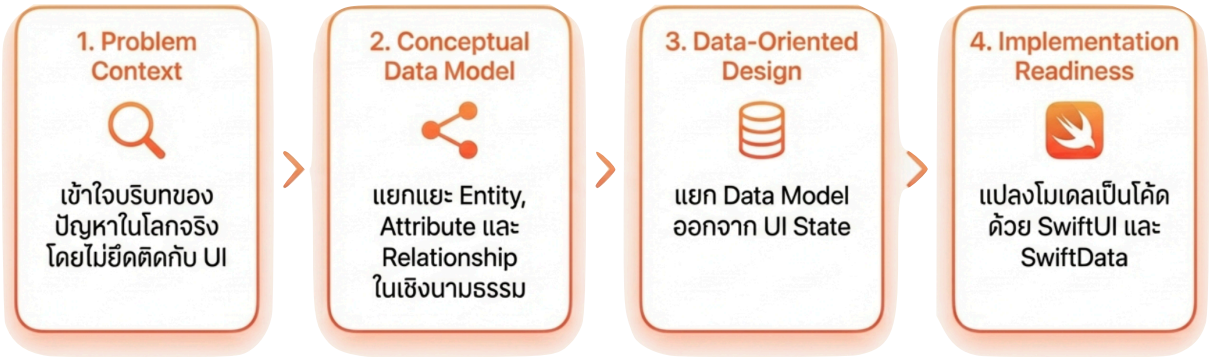
สิ่งสำคัญคือผู้สอนควรใช้กระดานหรือสื่อที่มองเห็นร่วมกัน มากกว่าการเปิดเครื่องมือพัฒนาโปรแกรมทันที เพราะจุดประสงค์ของช่วง Warm-up ไม่ใช่การเริ่มลงมือเขียนโค้ด แต่เป็นการเตรียมกรอบความคิด หากผู้เรียนสามารถมองเห็นว่า การออกแบบข้อมูลเกิดขึ้นได้ก่อนการเลือกเครื่องมือ พวกเขาจะพร้อมเปิดรับแนวคิด Thinking in Data ในขั้นถัดไป

ขั้นการเรียนรู้ (Learn)

กิจกรรมที่ 1: กรอบแนวคิด (Conceptual Framework) ของ Thinking in Data

ในขั้นการเรียนรู้ช่วงนี้ ผู้สอนมีบทบาทสำคัญในการวาง “โครงสร้างทางความคิด” ให้กับผู้เรียนอย่างเป็นระบบ กิจกรรมนี้ไม่ควรถูกมองว่าเป็นเพียงการบรรยายแนวคิดเชิงทฤษฎี แต่เป็นการค่อยๆ พาผู้เรียนปรับมุมมองจากการคิดแบบเน้นส่วนติดต่อผู้ใช้ ไปสู่การคิดเชิงโครงสร้างข้อมูล การอธิบายควรดำเนินอย่างต่อเนื่องโดยเชื่อมโยงกับสิ่งที่ผู้เรียนได้อภิปรายในช่วงนำเข้าสู่บทเรียน เพื่อให้เกิดความสอดคล้องทางความคิดและไม่ทำให้เนื้อหาถูกตัดขาดเป็นส่วน ๆ

ผู้สอนควรเริ่มต้นด้วยการสะท้อนบทสนทนาในช่วง Warm-up โดยชี้ให้เห็นว่า **เมื่อคำถามเปลี่ยนจาก “แอปควรมีหน้าจออะไร” ไปเป็น “ระบบต้องจัดการข้อมูลอะไร”** ผู้เรียนจะเริ่มเห็นโครงสร้างของระบบอย่างชัดเจนขึ้น จากจุดนี้จึงอธิบายว่า **แนวทางการคิดดังกล่าวคือหัวใจของกรอบแนวคิดที่เรียกว่า Thinking in Data** ซึ่งเป็นรูปแบบหนึ่งของการคิดเชิงคำนวณที่ให้ความสำคัญกับโครงสร้างและความสัมพันธ์ของข้อมูล ก่อนการพิจารณากระบวนการหรือเทคโนโลยีที่ใช้พัฒนา



การอธิบายควรเริ่มจากภาพรวมของกรอบแนวคิด โดยเน้นว่า Thinking in Data ประกอบด้วยกระบวนการที่มีลำดับความคิดชัดเจน ตั้งแต่การทำความเข้าใจบริบทของปัญหา ไปสู่การสร้างแบบจำลองข้อมูลเชิงแนวคิด จาก

นั้นจึงเชื่อมโยงแบบจำลองดังกล่าวเข้าสู่การออกแบบแอป และปิดท้ายด้วยความพร้อมในการพัฒนาเชิงเทคนิค การนำเสนอควรใช้ภาษาที่เรียบง่ายและส่งเสริมให้ผู้เรียนเห็นว่า **กระบวนการนี้เป็น "วิธีคิด" มากกว่าสูตรสำเร็จทางเทคนิค**

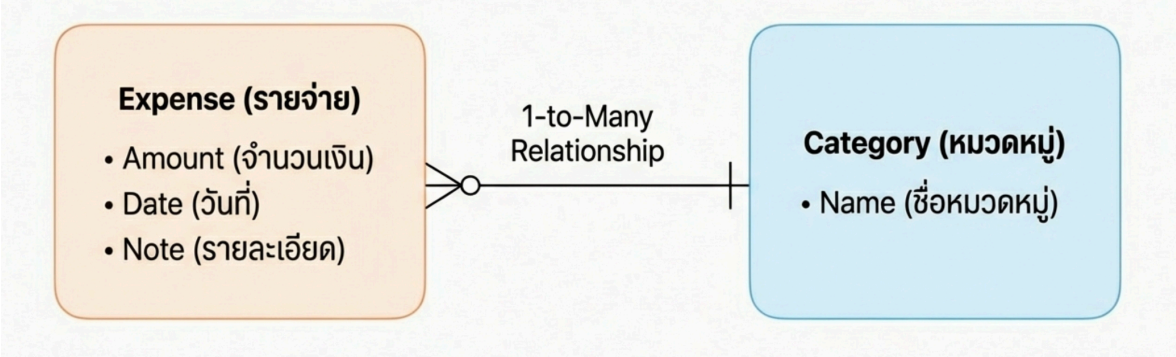
ขั้นตอนที่ 1: Problem Context

ในส่วนของ Problem Context ผู้สอนควรเน้นย้ำว่า **การเริ่มต้นที่ถูกต้องของระบบใด ๆ ไม่ได้อยู่ที่หน้าจอหรือโค้ด แต่คือการทำความเข้าใจโลกจริงที่ระบบต้องตอบสนอง** ผู้สอนควรตั้งคำถามว่า

- ผู้ใช้ต้องการทำอะไรในชีวิตจริง
- พฤติกรรมใดเกิดขึ้นซ้ำ ๆ
- ระบบต้องช่วยแก้ปัญหาอะไร

ตัวอย่างกรณีแอปบันทึกรายจ่ายสามารถนำมาใช้ประกอบการอธิบายได้อย่างชัดเจน โดยชี้ให้เห็นว่า สิ่งที่เกิดขึ้นจริงคือเหตุการณ์การใช้จ่าย ซึ่งมีจำนวนเงิน มีเวลา และมีบริบท ข้อมูลเหล่านี้เกิดขึ้นโดยไม่เกี่ยวข้องกับ UI การทำความเข้าใจบริบทเช่นนี้ช่วยให้ผู้เรียนมองเห็นว่าระบบมีหน้าที่จัดการ "เหตุการณ์" ไม่ใช่จัดการ "หน้าจอ"

ขั้นตอนที่ 2: Conceptual Data Model



เมื่อบริบทชัดเจนแล้ว ผู้สอนจึงพาผู้เรียนเข้าสู่ขั้น Conceptual Data Model โดยอธิบายว่า **ขั้นตอนนี้คือการแยกสิ่งที่ระบบต้องจัดการออกเป็นหน่วยข้อมูลเชิงนามธรรม ได้แก่ Entity, Attribute และ Relationship** การอธิบายควรใช้ตัวอย่างเดียวกันเพื่อความต่อเนื่อง เช่น Expense เป็น Entity หลัก ส่วน Category เป็นอีก Entity หนึ่งที่มีความสัมพันธ์กับ Expense ในลักษณะหนึ่งต่อหลาย ผู้สอนควรชี้ให้เห็นว่า การระบุองค์ประกอบเหล่านี้ไม่ได้เกิดจากการดูหน้าจอ แต่เกิดจากการวิเคราะห์ธรรมชาติของข้อมูล หากผู้เรียนเริ่มพูดถึงช่องกรอกข้อมูลหรือปุ่ม ผู้สอนควรค่อย ๆ นำกลับมาสู่คำถามว่า "ข้อมูลนี้มีตัวตนในโลกจริงหรือไม่"

ขั้นตอนที่ 3: Data-Oriented Application Design

ในขั้นตอนนี้ ผู้สอนควรชี้ให้ผู้เรียนเห็นถึงความจำเป็นของ **การแยกบทบาทของ Data Model (ข้อมูลถาวร) และ UI State (สถานะชั่วคราวของหน้าจอ)** ตัวอย่างเช่น จำนวนเงินที่ผู้ใช้กำลังพิมพ์อยู่ในช่องกรอกข้อมูลเป็นเพียง UI State ซึ่งมีอายุสั้นและเปลี่ยนแปลงได้ตลอดเวลา ขณะที่ Expense เป็น Data Model ที่ถูกบันทึกลงในระบบเพื่อเป็นข้อมูลถาวรของโดเมน การแยกสองสิ่งนี้ออกจากกันช่วยลดความซับซ้อนของโครงสร้าง และป้องกันไม่ให้ตรรกะของ UI เข้าไปปะปนกับข้อมูลหลัก ผู้สอนควรเน้นว่า การออกแบบที่ดีเริ่มจากการแยกความรับผิดชอบอย่างชัดเจน

ขั้นตอนที่ 4: Implementation Readiness
เมื่อโครงสร้างข้อมูลถูกออกแบบอย่างถูกต้องแล้ว **การเขียนโค้ดจะเป็นเพียงกระบวนการถ่ายทอดแนวคิดด้วยภาษาโปรแกรม** ตัวอย่างการสร้าง struct ของ Expense ใน Swift ควรถูกอธิบายว่าเป็นการแปลง Conceptual Model ให้เป็นโครงสร้างเชิงเทคนิค ไม่ใช่การเริ่มต้นคิดใหม่ทั้งหมด ผู้สอนควรตั้งคำถามสะท้อนให้ผู้เรียนเห็นว่า หากแบบจำลองข้อมูลชัดเจนตั้งแต่ต้นการพัฒนาในขั้นถัดไปจะมีความเป็นระบบและลดความสับสนอย่างไร

ตลอดการดำเนินกิจกรรม ผู้สอนควรใช้กระดานหรือสื่อที่ผู้เรียนสามารถมองเห็นร่วมกัน วาดโครงสร้าง Entity และ Relationship ให้ชัดเจน และเปิดโอกาสให้ผู้เรียนเสนอแนวคิดเพิ่มเติม การมีส่วนร่วมเช่นนี้ช่วยให้ผู้เรียนรู้สึกว่าตนเองเป็นส่วนหนึ่งของกระบวนการออกแบบ ไม่ใช่เพียงผู้รับสาร การหลีกเลี่ยงการเปิดเครื่องมือพัฒนาโปรแกรมในช่วงนี้จะช่วยให้ผู้เรียนโฟกัสที่แนวคิดมากกว่ารายละเอียดทางเทคนิค

กิจกรรมที่ 2: จาก Conceptual Model สู่ Swift Data Model

กิจกรรมในส่วนนี้มีจุดมุ่งหมายเพื่อพาผู้เรียนเปลี่ยนผ่านจากแบบจำลองข้อมูลเชิงแนวคิด (Conceptual Data Model) ที่ได้วิเคราะห์ไว้ในกิจกรรมก่อนหน้า ไปสู่การสร้างแบบจำลองข้อมูลในระดับภาษาโปรแกรมอย่างมีหลักการ ผู้สอนควรเน้นให้เห็นว่า **การเขียนโค้ด คือ การถ่ายทอดแนวคิดเชิงโครงสร้างที่ออกแบบไว้แล้วลงในภาษา Swift ซึ่งทำหน้าที่เป็น Modeling Language**

ผู้สอนอธิบายว่า เมื่อได้ Conceptual Model แล้ว ขั้นตอนถัดไปคือ **การเลือกภาษาโปรแกรมที่สามารถถ่ายทอดโครงสร้างข้อมูลได้อย่างชัดเจน** และ Swift เป็นภาษาที่ออกแบบมาโดยคำนึงถึงความปลอดภัยของข้อมูล (safety) และความชัดเจนของโครงสร้าง (clarity) ให้ผู้เรียนทบทวนว่า ในระดับแนวคิด เรามี Entity ชื่อ Expense และ Category พร้อม Attribute และ Relationship ที่ชัดเจนแล้ว คำถาม คือ “เราจะถ่ายทอดสิ่งนี้ลงในโค้ดอย่างไรโดยยังรักษาความหมายเชิงโดเมนไว้ครบถ้วน”
ผู้สอนอาจทำการสาธิตการสร้าง Data Model (หรือกำหนดให้ผู้เรียนทำตามไปพร้อมกัน) โดยการสร้าง Xcode Playground เพื่อให้ผู้เรียนเห็นกระบวนการอย่างเป็นขั้นตอน โดยเขียนโครงสร้าง Expense ตาม Conceptual Model ดังนี้

```
import Foundation

struct Expense: Identifiable {
    let id: UUID = UUID()
    let amount: Double
    let date: Date
    let note: String?
}

let expense1 = Expense(amount: 120, date: Date(), note: "Lunch")
let expense2 = expense1
```

ระหว่างพิมพ์โค้ด ผู้สอนอาจอธิบายอย่างช้าๆ เกี่ยวกับ

- **การใช้ struct ซึ่งสะท้อนถึง Value Semantics**
  ผู้สอนอธิบายว่า struct ใน Swift ใช้แนวคิดแบบ Value Semantics ซึ่งหมายความว่า ทุกครั้งที่มีการส่งต่อหรือกำหนดค่า (เช่น expense2 = expense1) จะเกิดการ “คัดลอกค่า” ไม่ใช่การอ้างอิงไปยังวัตถุเดียวกัน จากตัวอย่าง expense2 เป็นสำเนาของ expense1 หากมีการเปลี่ยนแปลง การเปลี่ยนแปลงจะไม่กระทบต้นฉบับ

- **การใช้ let เพื่อกำหนดให้ข้อมูลเป็น immutable**
  เมื่ออธิบายถึง property ภายใน struct ผู้สอนควรชี้ให้เห็นว่า ทุกตัวถูกประกาศด้วย let แทน var และตั้งคำถามว่า “เหตุใดเราจึงไม่อนุญาตให้เปลี่ยนแปลงค่าหลังสร้างเสร็จ” แนวคิดของ immutability มีความสำคัญอย่างยิ่งในเชิงสถาปัตยกรรม เพราะช่วยป้องกันการเปลี่ยนแปลงข้อมูลโดยไม่ตั้งใจ หากข้อมูลสามารถถูกแก้ไขได้จากหลายตำแหน่ง โอกาสเกิดข้อผิดพลาดจะเพิ่มขึ้นอย่างมาก ในบริบทของรายจ่ายหนึ่งรายการ เหตุการณ์ที่เกิดขึ้นในอดีตควรถูกบันทึกเป็นข้อเท็จจริง (fact) ไม่ควรถูกเปลี่ยนแปลงโดยพลการ หากต้องแก้ไข ควรสร้างอินสแตนซ์ใหม่แทนที่จะเปลี่ยนแปลงค่าภายใน นี่คือแนวคิดที่ช่วยให้ระบบมีความปลอดภัยและตรวจสอบได้ง่าย
- **การกำหนด Optional เพื่อสะท้อนความไม่แน่นอนในโลกจริง**
  ในการอธิบายข้อมูลแบบ Optional ผู้สอนไม่ควรอธิบายเพียงว่า “ใส่ ? เพราะเป็นตัวแปรที่อาจไม่มีค่า” แต่ควรขยายความในเชิงการออกแบบข้อมูลว่า Optional เป็นกลไกที่สะท้อนความเป็นจริงของโลกจริง ในกรณีรายจ่าย บางครั้งผู้ใช้อาจใส่รายละเอียดเพิ่มเติม เช่น “Lunch with team” แต่บางครั้งอาจไม่ใส่อะไรเลย การกำหนดให้ note เป็น String? แสดงให้เห็นว่า แบบจำลองข้อมูลยอมรับความไม่แน่นอน และไม่บังคับให้ทุกข้อมูลต้องสมบูรณ์ ดังนั้น Optional ไม่ได้มีไว้เพื่อหลีกเลี่ยง error เท่านั้น แต่เป็นการ “สื่อสารความหมายของข้อมูล” ในระดับ Data Model ว่าคุณลักษณะนี้ไม่จำเป็นต่อมีค่าเสมอไป การออกแบบเช่นนี้ช่วยให้แบบจำลองมีความยืดหยุ่นและสอดคล้องกับบริบทจริง
- **Identifiable ช่วยให้ SwiftUI ติดตามข้อมูลแต่ละรายการได้**
  ในการอธิบายถึง Identifiable และ let id: UUID = UUID ( ) ผู้สอนควรเชื่อมโยงกับการทำงานของ SwiftUI โดยตั้งคำถามว่า “ถ้าเรามีรายการรายจ่ายหลายรายการ SwiftUI จะรู้ได้อย่างไรว่ารายการใดคือรายการเดิม” SwiftUI ใช้กลไก identity เพื่อติดตามการเปลี่ยนแปลงของข้อมูลใน List หรือ ForEach หากไม่มี identifier ที่ชัดเจน SwiftUI จะไม่สามารถแยกแยะรายการเก่าและใหม่ได้อย่างถูกต้อง
  การกำหนด let id: UUID = UUID ( ) จึงทำให้รายจ่ายแต่ละรายการมีอัตลักษณ์เฉพาะตัว ผู้สอนควรเน้นว่า id ไม่ได้มีไว้เพื่อการแสดงผล แต่เป็นกลไกเชิงโครงสร้างที่ช่วยให้ระบบสามารถจัดการการเปลี่ยนแปลงของข้อมูลได้อย่างมีประสิทธิภาพ

ให้ผู้เรียนทดลองสร้าง Data Model ด้วย struct

```
import Foundation

struct Category: Identifiable, Hashable {
    let id: Int
    var name: String
}

let food = Category(id: 1, name: "Food")
let shopping = Category(id: 2, name: "Shopping")

struct Expense: Identifiable {
    let id: UUID = UUID()
    let amount: Double
    let date: Date
    let note: String?
    let category: Category

    init(amount: Double, date: Date, note: String? = nil, category: Category) {
        self.amount = amount
        self.date = date
        self.note = note
        self.category = category
    }
}
```

```
let expenses = [
    Expense(amount: 120, date: Date(), note: "Lunch", category: food),
    Expense(amount: 1000, date: Date(), category: shopping)
]

for item in expenses {
    print("Amount: \(item.amount)")
    print("Date: \(item.date)")
    print("Note: \(item.note ?? "None")")
    print("--------------------")
}
```

กิจกรรมที่ 3: การเปลี่ยนจาก struct สู่ SwiftData @Model

ในหัวข้อนี้เป็นช่วงเปลี่ยนผ่านที่สำคัญจาก Data Model ที่สร้างด้วย struct ไปสู่ SwiftData ผู้สอนควรให้ความสำคัญกับ “เหตุผลของการยกระดับ” มากกว่า การแนะนำเครื่องมือใหม่ทันที เป้าหมายของช่วงนี้คือ การทำให้ผู้เรียนตระหนักว่า **แม้ Model ที่ออกแบบด้วย struct จะมีความปลอดภัยและชัดเจนในเชิงโครงสร้าง แต่ก็ยังมีข้อจำกัดเชิงระบบที่ไม่เพียงพอสำหรับการใช้งานจริงในระดับแอปพลิเคชัน**

3.1. ความแตกต่างระหว่าง Value Semantics และ Reference Semantics

struct ในภาษา Swift ใช้แนวคิดแบบ Value Semantics กล่าวคือ เมื่อมีการกำหนดค่าหรือส่งต่อข้อมูล จะเกิดการคัดลอกค่า (copy) ทำให้แต่ละอินสแตนซ์มีสถานะเป็นอิสระจากกัน การเปลี่ยนแปลงค่าของอินสแตนซ์หนึ่งจะไม่ส่งผลกระทบต่ออินสแตนซ์อื่นโดยไม่ตั้งใจ

```
struct Score {
    var value: Int
}

var studentA = Score(value: 50)
var studentB = studentA

studentB.value = 80
print(studentA.value) // 50
print(studentB.value) // 80
```

ในทางตรงกันข้าม class ใช้แนวคิดแบบ Reference Semantics ซึ่งหมายความว่าอินสแตนซ์หลายตัวสามารถอ้างอิงไปยังข้อมูลชุดเดียวกันได้ การเปลี่ยนแปลงผ่านอ้างอิงหนึ่งจะส่งผลกระทบต่ออ้างอิงอื่นทั้งหมด แนวคิดนี้มีความสำคัญในบริบทที่ข้อมูลต้องมีอัตลักษณ์ (identity) และวงจรชีวิต (lifecycle) ที่ชัดเจน เช่น ข้อมูลที่ต้องถูกจัดเก็บถาวร ถูกแก้ไขซ้ำ และถูกใช้งานร่วมกันในหลายส่วนของระบบ

```
class Score {
    var value: Int
    init(value: Int) {
        self.value = value
    }
}

var studentA = Score(value: 50)
var studentB = studentA

studentB.value = 80
print(studentA.value) // 80
print(studentB.value) // 80
```

3.2. ความแตกต่างระหว่าง In-memory Model และ Persistent Model

ผู้สอนควรตั้งคำถามนำ เพื่อกระตุ้นการคิด เช่น “ถ้าเราปิดแอปแล้วเปิดใหม่ ข้อมูลจะยังอยู่หรือไม่?” และควรปล่อยให้ผู้เรียนตอบก่อน แล้วจึงให้ทดลองสร้างแอปอย่างง่ายๆ ชื่อ InMemoryDemo ดังนี้

```
import SwiftUI

struct Expense {
    let amount: Double
    let date: Date
}

struct ContentView: View {

    @State private var expenses: [Expense] = []

        var body: some View {
            VStack {
                Button("Add Expense") {
                    expenses.append(
                        Expense(amount: 120, date: .now)
                    )
                }

                List(expenses, id: \.date) { expense in
                    Text("\(expense.amount)")
                }
            }
        }
}
```

ทดลองรันโปรแกรมและกดปุ่ม “Add Expense” หลายครั้ง จะมีรายการปรากฏใน List
หยุดรันแอป (Stop) และ รันใหม่อีกครั้ง
ผลลัพธ์ : รายการทั้งหมดหายไป

ผู้สอนอธิบายว่า **Data Model ที่ใช้ struct จะเป็นเพียง In-memory Model** กล่าวคือ **ข้อมูลมีชีวิตอยู่เฉพาะในหน่วยความจำระหว่างที่แอปทำงาน เมื่อแอปหยุดทำงาน ข้อมูลจะสูญหายทั้งหมด**
จากนั้นควรขยายความถึงข้อจำกัดสำคัญของ In-memory Model ดังนี้

1. **ไม่มีความคงอยู่ (No Persistence) -** ข้อมูลไม่สามารถถูกจัดเก็บและเรียกกลับมาใช้งานได้เมื่อเปิดแอปใหม
2. **ไม่มีวงจรชีวิตในระดับระบบ (No Lifecycle Management) -** ไม่มีระบบติดตามโดยอัตโนมัติว่า object ถูกสร้างเมื่อใด ถูกแก้ไขเมื่อใด หรือถูกลบเมื่อใด ผู้พัฒนาต้องจัดการเอง
3. **ไม่มีการจัดการความสอดคล้องของข้อมูล (No Consistency Across Views) -** ในระบบที่มีหลาย View หรือหลาย ViewModel การจัดการข้อมูลแบบธรรมดาอาจทำให้เกิดปัญหาการอ้างอิงที่ไม่สอดคล้องกัน

ผู้สอนอธิบายต่อไปว่า แม้ struct จะเหมาะกับการสร้างแบบจำลองเชิงแนวคิด แต่เมื่อระบบมีความซับซ้อนมากขึ้น ข้อจำกัดเหล่านี้จะกลายเป็นอุปสรรคเชิงสถาปัตยกรรม เมื่อผู้เรียนเห็นข้อจำกัดของ In-memory Model แล้ว ผู้สอนจึงอธิบายว่า SwiftData ทำหน้าที่เป็นกลไกที่ยกระดับ Data Model ให้กลายเป็น Persistent Model ควรเน้นประโยคสำคัญว่า **SwiftData ไม่ได้เปลี่ยนโครงสร้างข้อมูล แต่เปลี่ยนบทบาทของข้อมูลในระบบ**

ให้ผู้เรียนลองสร้างโปรเจกต์ใหม่ โดยตั้งชื่อว่า PersistentModelDemo จากนั้นทดลองสร้างโมเดลข้อมูลด้วย Class และ SwiftData ดังนี้

```
import SwiftUI
import SwiftData
```

```
@Model
class Expense {
    var amount: Double
    var date: Date

    init(amount: Double, date: Date) {
        self.amount = amount
        self.date = date
    }
}

struct ContentView: View {

    @Environment(\.modelContext) private var context
    @Query private var expenses: [Expense]

    var body: some View {
        VStack {
            Button("Add Expense") {
                let expense = Expense(amount: 120, date: .now)
                context.insert(expense)
                try? context.save()
            }

            List(expenses, id: \.date) { expense in
                Text("\(expense.amount)")
            }
        }
    }
}
```

และแก้ไขคำสั่งที่ไฟล์ PersistentModelDemoApp.swift ดังนี้

```
import SwiftUI
import SwiftData

@main
struct PersistentModelDemoApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .modelContainer(for: [Expense.self])
    }
}
```

ทดลองรันโปรแกรมและกดปุ่ม “Add Expense” หลายครั้ง จะมีรายการปรากฏใน List
หยุดรันแอป (Stop) และ รันใหม่อีกครั้ง
ผลลัพธ์ : รายการทั้งหมดยังคงอยู่

เพื่อให้ผู้เรียนเกิดความเข้าใจเกี่ยวกับการใช้ SwiftData ผู้สอนควรอธิบายเพิ่มเติมดังนี้

- ในขั้นแรก ผู้สอนควรชี้ให้ผู้เรียนเห็นว่า คลาส Expense ไม่ใช่เพียงโครงสร้างข้อมูลทั่วไป แต่เป็นตัวแทนของข้อมูลที่ต้องถูกจัดเก็บถาวรในฐานข้อมูลของแอป การใช้ @Model ทำหน้าที่ประกาศให้ SwiftData ทราบว่าคลาสนี้เป็น entity ที่ต้องถูกติดตาม จัดการ และบันทึกลง persistent store เมื่อ SwiftData พบ @Model ระบบจะสร้าง schema สำหรับจัดเก็บข้อมูลโดยอัตโนมัติ เปลี่ยน property ภายในคลาสให้กลายเป็นฟิลด์ในฐานข้อมูล และจัดเตรียมกลไกสำหรับติดตามการเปลี่ยนแปลงของ object แต่ละตัว นอกจากนี้ SwiftData ยังสร้างอัตลักษณ์ของ object เพื่อให้สามารถอ้างอิงและซิงโครไนซ์ข้อมูลระหว่างหลาย view ได้อย่างถูกต้อง

จุดสำคัญที่ควรเน้นคือ SwiftData กำหนดให้โมเดลต้องเป็น class แทนที่จะเป็น struct เนื่องจากระบบต้องการ reference semantics เพื่อให้สามารถติดตาม identity ของ object และจัดการ transaction ได้อย่างมีประสิทธิภาพ

- หลังจากกำหนดโมเดลข้อมูลแล้ว ขั้นตอนถัดมา คือ การกำหนด container ของข้อมูลในไฟล์หลักของแอป คำสั่ง .modelContainer(for: [Expense.self]) มีบทบาทสำคัญในการเปิดใช้งานระบบจัดเก็บข้อมูลของ SwiftData ผู้สอนควรอธิบายว่า model container เปรียบเสมือนตัวแทนของฐานข้อมูลทั้งระบบ โดยมีหน้าที่สร้าง persistent store โหลด schema จากโมเดลที่กำหนด จัดการการย้ายโครงสร้างข้อมูล (migration) และสร้าง context สำหรับให้ view ต่าง ๆ ในแอปใช้ร่วมกัน การระบุ [Expense.self] เป็นการบอกให้ SwiftData ทราบว่าฐานข้อมูลนี้ต้องรองรับ Entity ใดบ้าง และจะใช้รายการนี้ในการสร้างโครงสร้างของฐานข้อมูลตั้งแต่เริ่มต้น
  การอธิบายส่วนนี้ควรทำให้ผู้เรียนเห็นว่า container เป็นศูนย์กลางของระบบ persistence ที่เชื่อมกับ lifecycle ของแอปทั้งหมด
- ภายใน ContentView การดึง modelContext ผ่าน @Environment(\.modelContext) เป็นขั้นตอนที่ทำให้ View สามารถเข้าถึงพื้นที่ทำงานสำหรับอ่านและเขียนข้อมูลได้ โดย context ทำหน้าที่เป็น workspace หรือ session สำหรับการทำงานกับข้อมูลในช่วงเวลาหนึ่ง โดย context จะเก็บ object ที่ถูกโหลดมา ติดตามการเปลี่ยนแปลง และจัดการ transaction ก่อนจะบันทึกลงฐานข้อมูลจริง การเรียก context.insert(expense) หมายถึง การเพิ่ม object ใหม่เข้าสู่พื้นที่ทำงานของ context ส่วนการเรียก context.save( ) คือการยืนยันและบันทึกการเปลี่ยนแปลงเหล่านั้นลง persistent store
- @Query ทำหน้าที่เป็นกลไกเชื่อมโยงระหว่างฐานข้อมูลกับส่วนติดต่อผู้ใช้โดยตรง กล่าวได้ว่าเป็นตัวแทนของ “คำร้องขอข้อมูลแบบประกาศ (declarative fetch request)” ที่ถูกผสานเข้ากับ lifecycle ของ SwiftUI เมื่อผู้พัฒนาประกาศ @Query ใน view ระบบจะทำการดึงข้อมูลจาก persistent store และผูกผลลัพธ์เข้ากับ view โดยอัตโนมัติ

  ลักษณะสำคัญของ @Query คือผลลัพธ์ที่ได้เป็นข้อมูลแบบมีชีวิต (live data) หมายความว่า เมื่อข้อมูลในฐานข้อมูลมีการเปลี่ยนแปลง ไม่ว่าจะเกิดจากการเพิ่ม แก้ไข หรือลบรายการ SwiftUI จะรีเฟรช view ที่ใช้ @Query ทันทีโดยไม่ต้องเขียนโค้ดสังเกตการเปลี่ยนแปลงเอง แนวคิดนี้สะท้อนหลัก reactive data flow ที่เป็นหัวใจของ SwiftUI

หลังจากทดลองทำกิจกรรมที่ 3.1 และ 3.2 แล้ว ผู้เรียนควรเห็นภาพชัดเจนว่า แม้หน้าตาของแอปจะคล้ายกัน แต่ “พฤติกรรมของข้อมูล” แตกต่างกันโดยสิ้นเชิง ในโปรเจกต์ **InMemoryDemo** ซึ่งใช้ struct และ @State ข้อมูลจะถูกเก็บไว้ในหน่วยความจำของแอประหว่างที่โปรแกรมกำลังทำงาน เมื่อหยุดรันแอป ข้อมูลในหน่วยความจำจะถูกล้าง ค่าในตัวแปรจึงหายไปทั้งหมด นี่คือสิ่งที่เรียกว่า **In-memory Model**

ในทางตรงกันข้าม โปรเจกต์ **PersistentModelDemo** ใช้ class ร่วมกับ @Model และ SwiftData ทำให้ข้อมูลไม่ได้อยู่เพียงใน RAM แต่ถูกจัดการผ่าน ModelContext และถูกบันทึกลงใน persistent store เมื่อปิดและเปิดแอปใหม่ ข้อมูลจึงยังคงอยู่ นี่คือ **Persistent Model**

ในส่วนนี้ ผู้สอนอธิบายต่อไปว่า การเปลี่ยนจาก struct ไปสู่ class ไม่ได้เกิดจากความสะดวกทางไวยากรณ์ แต่เกิดจากข้อกำหนดเชิงสถาปัตยกรรมของ SwiftData โดยสามารถอธิบายเหตุผลเชิงแนวคิด 3 ประการ ดังนี้:

1. **Identity -** ใน Persistent Model ข้อมูลหนึ่งรายการต้องมีตัวตนเดียวในระบบ การใช้ Reference Semantics ของ class ช่วยให้ทุกส่วนของแอปอ้างอิงไปยัง object เดียวกันได้ เมื่อมีการแก้ไขข้อมูลในที่หนึ่ง การเปลี่ยนแปลงจะสะท้อนในทุกที่ที่ใช้งานข้อมูลนั้น

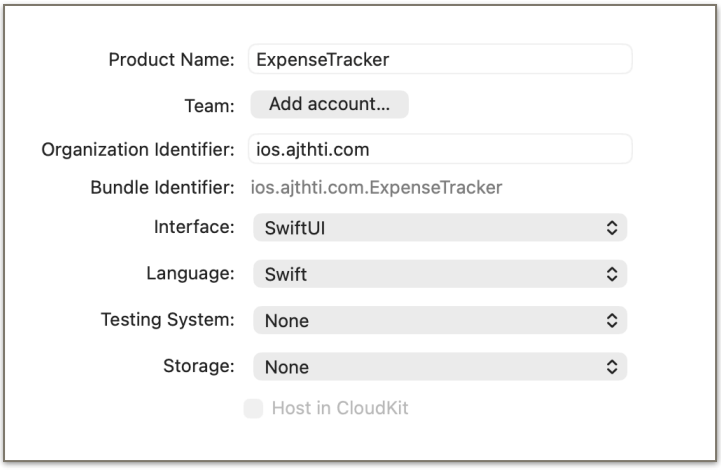
2. **Lifecycle Management -** SwiftData ต้องสามารถติดตามการสร้าง การแก้ไข และการลบข้อมูลได้ การทำเช่นนี้จำเป็นต้องอาศัย object ที่มีตัวตนต่อเนื่อง ไม่ใช่ค่าที่ถูกคัดลอกไปมาแบบ Value Semantics
3. **Consistency Across Views -** ในระบบที่มีหลาย View เมื่อข้อมูลถูกแก้ไขใน View หนึ่ง การเปลี่ยนแปลงควรถูกสะท้อนไปยัง View อื่นโดยอัตโนมัติ การใช้ class และ Reference Semantics ช่วยให้ข้อมูลมีความสอดคล้องกันทั่วทั้งระบบ

ผู้สอนควรเน้นย้ำในตอนท้ายว่า แม้การใช้ class จะเป็นข้อกำหนดของ SwiftData framework แต่ในเชิงแนวคิดแล้ว การตัดสินใจนี้สอดคล้องกับบทบาทใหม่ของข้อมูลในระดับระบบ นั่นคือ การยกระดับจากค่าชั่วคราว ไปสู่ข้อมูลเชิงโดเมนที่มีอัตลักษณ์และวงจรชีวิตชัดเจน

กิจกรรมที่ 4: การเชื่อมโยง “แนวคิด” เข้ากับการพัฒนาแอปด้วย Xcode

ก่อนสร้างโปรเจกต์ ผู้สอนควรแสดงโครงสร้างโฟลเดอร์ของโปรเจกต์ให้ผู้เรียนเห็นก่อน และอธิบายว่าโครงสร้างไฟล์สะท้อน “การแยกความรับผิดชอบ” ทางสถาปัตยกรรม

```
ExpenseTrackerApp/
│
├── ExpenseTrackerApp.swift          // จุดเริ่มต้นของแอป และกำหนด SwiftData modelContainer
│
├── Models/
│   ├── Expense.swift                // โมเดลรายจ่าย (Persistent Domain Model)
│   └── Category.swift               // โมเดลหมวดหมู่ และความสัมพันธ์แบบ One-to-Many
│
├── ViewModels/
│   └── ExpenseViewModel.swift       // ตัวกลางระหว่าง Model และ View จัดการ Create/Delete
│
└── Views/
    ├── ExpenseListView.swift        // แสดงรายการรายจ่ายทั้งหมดด้วย @Query
    ├── ExpenseRowView.swift         // แสดงข้อมูลรายจ่ายหนึ่งรายการ
    ├── AddExpenseView.swift          // หน้าสำหรับเพิ่มรายจ่ายใหม่
    └── CategoryPickerView.swift     // เลือกหมวดหมู่โดยใช้ @Binding และ @Query
```



ขั้นตอนที่ 1
เริ่มต้นสร้างโปรเจกต์ iOS App ใน Xcode โดยตั้งชื่อว่า **ExpenseTracker** และกำหนดค่าต่างๆ ดังนี้

- Interface: SwiftUI
- Language: Swift
- Storage: None (สามารถเพิ่ม SwiftData Framework ได้ภายหลัง)

ขั้นตอนที่ 2 สร้างโฟลเดอร์สำหรับจัดเก็บไฟล์ในโปรเจค ตามแนวคิด MVVM ดังนี้

- โฟลเดอร์ **Models** คือ Domain Layer
- โฟลเดอร์ **ViewModels** คือ Business Logic Layer
- โฟลเดอร์ **Views** คือ Presentation Layer

ขั้นตอนที่ 3 สร้างไฟล์โมเดลของประเภทรายจ่าย ตั้งชื่อว่า Category.swift ในโฟลเดอร์ Models และเขียนโค้ดดังนี้

```
import SwiftUI
import SwiftData

@Model
class Category {
    var name: String
    var expenses: [Expense] = []

    init(name: String) {
        self.name = name
    }
}
```

ขั้นตอนที่ 4 สร้างไฟล์โมเดลของรายจ่าย โดยตั้งชื่อว่า Expense.swift ในโฟลเดอร์ Models และเขียนโค้ดดังนี้

```
import SwiftUI
import SwiftData

@Model
class Expense {
    var amount: Double
    var date: Date
    var note: String?
    var category: Category?

    init(amount: Double, date: Date, note: String? = nil,
      category: Category? = nil) {
        self.amount = amount
        self.date = date
        self.note = note
        self.category = category
    }
}
```

ผู้สอนอธิบายว่า

- SwiftData จะทำหน้าที่เป็นตัวจัดการ persistence และ lifecycle ให้กับ Data Model ที่เราประกาศด้วย @Model และใช้ class เพื่อรองรับ reference semantics โดย SwiftData จัดการ identity ให้โดยอัตโนมัติ
- **Category** และ **Expense** เป็น Entity ซึ่งมีความสัมพันธ์กันแบบ One-to-Many โดย [Expense] ซึ่งอยู่ในไฟล์ Category.swift คือ collection ของ Expense ที่สัมพันธ์กับ Category นี้
- SwiftData จะสร้าง inverse relationship ระหว่าง Category และ Expense เมื่อ Expense ถูกกำหนดค่า `category` ระบบจะเพิ่ม Expense นั้นเข้าไปใน `expenses` ของ Category ให้อัตโนมัติ นี่คือความสามารถที่ช่วยรักษา **Consistency Across Objects**

ขั้นตอนที่ 5 เพิ่ม .modelContainer ในไฟล์ ExpenseTrackerApp.swift ดังนี้

```
import SwiftUI
import SwiftData

@main
struct ExpenseTrackerApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .modelContainer(for: [Expense.self, Category.self])
    }
}
```

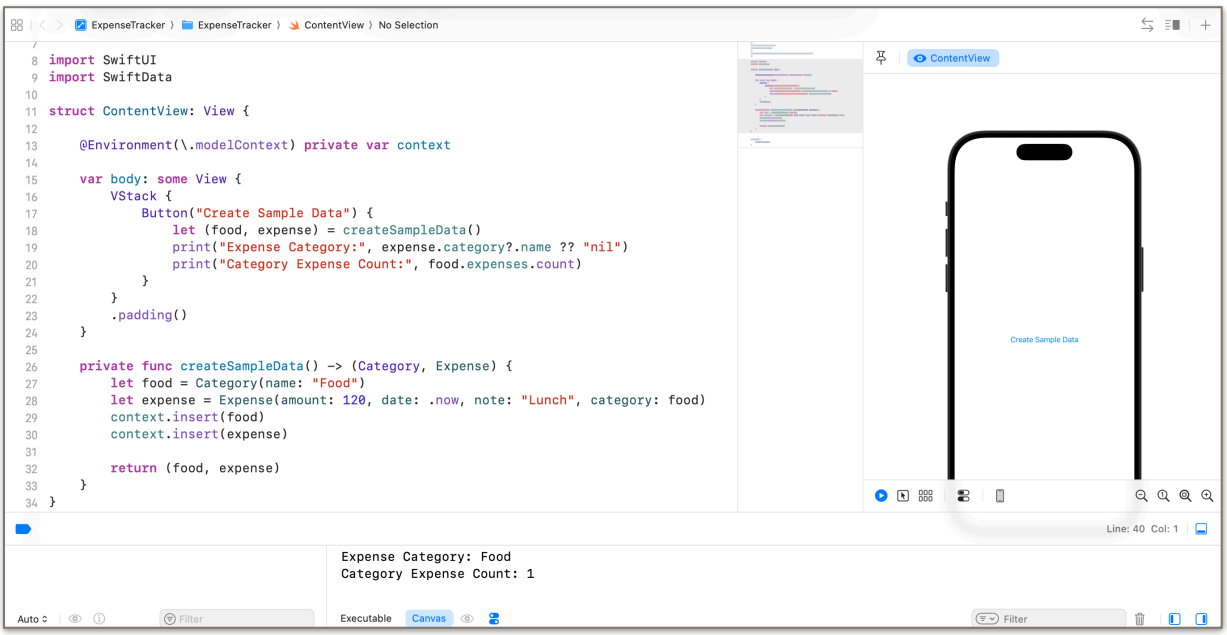
ในขณะสาธิต ควรเน้นยำให้ผู้เรียนเพิ่มคำสั่ง **import** SwiftData และอธิบายว่า `.modelContainer` ทำหน้าที่กำหนดพื้นที่จัดเก็บข้อมูลถาวร และประกาศกับระบบว่า Model ใดบ้าง จะถูกจัดการโดย SwiftData

เพื่อให้เห็นภาพในการเพิ่มข้อมูล ผู้สอนอาจสาธิตการทำงานของ SwiftData ในไฟล์ ContentView.swift ดังนี้

```
import SwiftUI
import SwiftData

struct ContentView: View {

    @Environment(\.modelContext) private var context

    var body: some View {

        VStack {

            Button("Create Sample Data") {
                let (food, expense) = createSampleData()
                print("Expense Category:", expense.category?.name ?? "nil")
                print("Category Expense Count:", food.expenses.count)
            }
        }
        .padding()
    }


    private func createSampleData() -> (Category, Expense) {
        let food = Category(name: "Food")
        let expense = Expense(amount: 120, date: .now, note: "Lunch", category: food)

        context.insert(food)
        context.insert(expense)

        return (food, expense)
    }
}
```

โดยผู้สอนอธิบายการใช้คำสั่ง @Environment(\.modelContext) private var context ใน ContentView.swift ว่า

- @Environment ใน SwiftUI เป็น Property Wrapper ที่ใช้สำหรับดึง “ค่าบริบทของระบบ (environment values)” เข้ามาใช้ใน View
- ModelContext คือ “ตัวกลาง” ที่เชื่อม View เข้ากับ ModelContainer



ปุ่ม `Button("Create Sample Data")` ทำหน้าที่ในการสร้างข้อมูลตัวอย่างผ่านการเรียกใช้เมธอด `createSampleData()` โดยจะคืนค่า `Category` และ `Expense` กลับไป และพิมพ์ผลลัพธ์ออกมา ดังนี้

```
Expense Category: Food
Category Expense Count: 1
```

ขั้นตอนที่ 6 สร้างไฟล์ ExpenseViewModel.swift ในโฟลเดอร์ ViewModels และเขียนโค้ดดังนี้

```
import SwiftData
import SwiftUI

@Observable
class ExpenseViewModel {

    private let context: ModelContext

    // UI State
    var isSaving = false
    var errorMessage: String?


    init(context: ModelContext) {
        self.context = context
    }

    // MARK: - Commands
    func addExpense(amount: Double, note: String?, category: Category?) {

        isSaving = true

        let expense = Expense(amount: amount,
                              date: .now,
                              note: note,
                              category: category)

        context.insert(expense)

        do {
            try context.save()
```

```
        } catch {
            errorMessage = "Failed to save expense"
        }

        isSaving = false
    }

    func deleteExpense(_ expense: Expense) {
        context.delete(expense)

        do {
            try context.save()
        } catch {
            errorMessage = "Failed to delete expense"
        }
    }

    // MARK: - Initial Data

    func setupInitialData(categories: [Category], expenses: [Expense]) {

        let defaultCategoryNames = ["Food",
                                    "Transport",
                                    "Shopping",
                                    "Bills",
                                    "Entertainment"]

        var foodCategory: Category?

        for name in defaultCategoryNames {
            if let existing = categories.first(where: { $0.name == name }) {
                if name == "Food" { foodCategory = existing }
            } else {
                let newCategory = Category(name: name)
                context.insert(newCategory)
                if name == "Food" { foodCategory = newCategory }
            }
        }

        if expenses.isEmpty, let foodCategory {
            let sample = Expense(
                amount: 120,
                date: .now,
                note: "Lunch",
                category: foodCategory
            )
            context.insert(sample)
        }

        try? context.save()
    }
}
```

ผู้สอนควรอธิบายว่า

- การใช้ @Observable ทำให้ ViewModel สามารถแจ้ง View ได้ หากค่า (state) ภายใน เช่น isSaving หรือ errorMessage มีการเปลี่ยนแปลง นี่คือ กลไกที่ทำให้ SwiftUI อัปเดต UI อัตโนมัติ
- การกำหนด ตัวแปร isSaving และ errorMessage เพื่อใช้สำหรับการตรวจสอบสถานะการทำงานของของระบบ โดยสามารถนำไปใช้ในการจัดการสถานะของ UI เช่น การปิดการโต้ตอบของผู้ใช้ การแสดง loading indicator หรือการแสดงข้อความ error บน AlertBox เป็นต้น

- คำสั่ง private let context: ModelContext ไม่ได้เป็นการสร้าง ModelContext ใหม่ แต่เป็นการประกาศตัวแปร context เพื่อใช้อ้างอิงถึง ModelContext ที่ถูกส่งเข้ามาจาก SwiftData ผ่าน View โดย ModelContext นี้ ใช้เป็นตัวกลางในการสร้าง ลบ และบันทึกข้อมูลใน ModelContainer ของแอป
- เมธอด addExpense, deleteExpense และ setupInitialData เป็นตัวแทนของการกระทำต่อข้อมูลในระบบ โดย ViewModel ทำหน้าที่เป็นชั้นที่ควบคุมการสร้าง ลบ และเตรียมข้อมูลเริ่มต้นผ่าน ModelContext การออกแบบลักษณะนี้ช่วยให้ View ไม่ต้องจัดการ persistence โดยตรง ทำให้โค้ดมีโครงสร้างชัดเจน รองรับการขยายระบบ และสอดคล้องกับสถาปัตยกรรม MVVM

ขั้นตอนที่ 7 สร้าง View ชื่อ ExpenseListView และ ExpenseRowView เพื่อแสดงรายการทั้งหมดจาก SwiftData ผ่าน @Query

คำสั่งในไฟล์ ExpenseListView.swift

```
import SwiftUI
import SwiftData

struct ExpenseListView: View {

    // MARK: - Environment
    @Environment(\.modelContext) private var context

    // MARK: - Data Queries
    @Query(sort: \Expense.date, order: .reverse)
    private var expenses: [Expense]
    @Query
    private var categories: [Category]

    // MARK: - UI State
    @State private var showingAddExpenseView = false

    var body: some View {
        NavigationStack {
            List {
                ForEach(expenses) { expenseItem in
                    ExpenseRowView(expense: expenseItem)
                }
                .onDelete { indexSet in
                    let viewModel = ExpenseViewModel(context: context)
                    indexSet.map { expenses[$0] }
                        .forEach { viewModel.deleteExpense($0) }
                }
            }
            .navigationTitle("Expenses")

            // MARK: - Toolbar
            .toolbar {
                ToolbarItem(placement: .topBarTrailing) {
                    Button {
                        showingAddExpenseView = true
                    } label: {
                        Image(systemName: "plus")
                    }
                }
            }

            // MARK: - Sheet
            .sheet(isPresented: $showingAddExpenseView) {
                AddExpenseView(viewModel: ExpenseViewModel(context: context))
            }
        }
```

```
        // MARK: - Initial Data Setup
        .task {
            let viewModel = ExpenseViewModel(context: context)
            viewModel.setupInitialData(categories: categories,
                                       expenses: expenses)
        }
    }
}
```

คำสั่งในไฟล์ ExpenseRowView.swift

```
import SwiftUI

struct ExpenseRowView: View {

    let expense: Expense

    var body: some View {
        VStack(alignment: .leading) {
            Text(expense.amount, format: .currency(code: "THB"))
                .font(.headline)

            Text(expense.category?.name ?? "No Category")
                .font(.caption)
                .foregroundStyle(.secondary)

            Text(expense.date.formatted(date: .abbreviated, time:.shortened))
                .font(.caption2)
        }
    }
}
```

เริ่มจากการอธิบายก่อนว่า ExpenseListView มีบทบาทเป็น “หน้าจอหลัก” ที่แสดงรายการรายจ่ายทั้งหมด โดยไม่ได้สร้างหรือจัดเก็บข้อมูลเอง แต่ดึงข้อมูลจาก SwiftData ผ่าน @Query และมีการใช้ .task ซึ่งจะทำงานเมื่อ ExpenseListView ปรากฏบนจอภาพ สำหรับสร้าง Sample Data ผ่านฟังกชัน setupInitialData ของ ViewModel

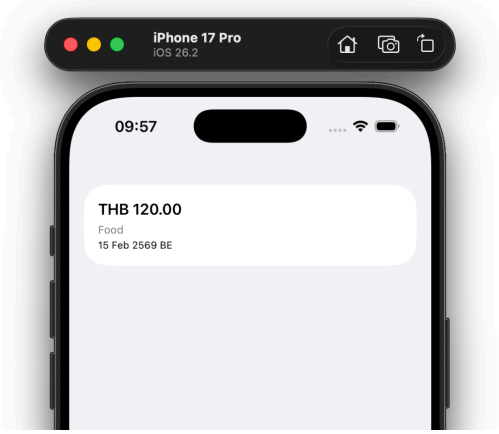
- คำสั่ง @Environment(\.modelContext) private var context เป็นการดึง ModelContext จากระบบ SwiftData ผ่าน Environment มาใช้ใน View นี้
- ส่วนของการ Qury ทำหน้าที่ในการดึงข้อมูลจาก SwiftData persistence store มาให้ View ใช้โดยอัตโนมัติ

  ผู้สอนอาจอธิบายเพิ่มเติมในรายละเอียดเกี่ยวกับการใช้ @Query ดังนี้
  @Query เป็น Property Wrapper ใน SwiftData ที่ทำหน้าที่ดึงข้อมูลจาก ModelContainer โดยอัตโนมัติ และผูกผลลัพธ์เข้ากับ SwiftUI View ในลักษณะ reactive data flow เมื่อข้อมูลใน persistent store มีการเปลี่ยนแปลง เช่น การเพิ่ม ลบ หรือแก้ไขข้อมูล SwiftUI จะทำการ re-render View ที่ใช้ @Query โดยอัตโนมัติ ในเชิงสถาปัตยกรรมจึงกล่าวได้ว่า @Query ทำหน้าที่เป็นกลไกเชื่อมโยงข้อมูลระหว่าง persistence layer และ View layer โดยทำให้ View สามารถเข้าถึงข้อมูลจาก SwiftData ในรูปแบบ collection ที่ bind กับ UI ได้โดยตรง

  โดยรูปแบบพื้นฐานของการใช้ @Query คือ @Query private var expenses: [Expense] และสามารถใช้ร่วมกับการ sort และ filter ได้ ดังนี้

```swift
@Query(sort: \Expense.date, order: .reverse)
private var expenses: [Expense]


@Query(filter: #Predicate<Expense> { $0.amount > 100 })
private var expenses: [Expense]


@Query(filter: #Predicate<Expense> { $0.amount > 0 },
         sort: \Expense.date,
        order: .reverse )
private var expenses: [Expense]
```

- ตัวแปร showingAddExpenseView ทำหน้าที่เก็บสถานะว่า หน้าจอสำหรับเพิ่มรายจ่ายควรถูกแสดงหรือไม่
- ในส่วนของการแสดงผล โครงสร้างเริ่มต้นด้วย NavigationStack ซึ่งทำหน้าที่กำหนดบริบทของการนำทางภายในหน้าจอ ทำให้ View นี้สามารถแสดงชื่อหน้าด้านบน และรองรับการนำทางไปยัง View อื่นในอนาคตได้ เช่น หน้ารายละเอียดหรือหน้าสร้างข้อมูลใหม่
- ภายใน NavigationStack มีการใช้ List เพื่อแสดงข้อมูลในรูปแบบรายการแนวตั้ง ซึ่งเป็นรูปแบบที่เหมาะสมกับข้อมูลประเภท collection อย่างรายจ่าย โดย List จะทำหน้าที่จัดการ layout การเลื่อนหน้าจอ การแบ่งแถว และพฤติกรรมพื้นฐานของรายการให้โดยอัตโนมัติ ทำให้ผู้พัฒนาไม่ต้องจัดการรายละเอียดเชิงกลไกของ UI ด้วยตนเอง
- การนำข้อมูลจาก SwiftData มาแสดงผลเกิดขึ้นผ่านคำสั่ง ForEach(expenses) ซึ่งรับข้อมูลจากตัวแปร expenses ที่ถูกดึงมาจาก persistence ด้วย @Query โดยคำสั่ง ForEach จะทำหน้าที่วนผ่านข้อมูลรายจ่ายทีละรายการ และส่งข้อมูลแต่ละหน่วยไปยัง ExpenseRowView เพื่อสร้าง UI สำหรับแถวนั้นโดยเฉพาะ
- ExpenseRowView มีบทบาทสำคัญในฐานะ View ย่อยที่ทำหน้าที่แสดงข้อมูลของรายจ่ายหนึ่งรายการโดยเฉพาะ การแยก Row View ออกเป็นคอมโพเนนต์เฉพาะช่วยให้โค้ดมีความชัดเจนและสามารถนำกลับมาใช้ซ้ำได้ นอกจากนี้ยังช่วยให้ View หลักไม่ต้องรับผิดชอบรายละเอียดของการจัดรูปแบบข้อมูลแต่ละรายการ ส่งผลให้โครงสร้างของหน้ารายการอ่านง่ายและดูแลรักษาได้สะดวกขึ้น



ผู้สอนควรเปลี่ยนมุมมองไปที่ ExpenseRowView ซึ่งเป็นหน่วยย่อยของ Presentation Layer เพื่ออธิบายถึงรูปแบบในการนำเสนอข้อมูลบนจอภาพ

การกำหนดรูปแบบด้วยคำสั่ง date: .abbreviated คือ การกำหนดรูปแบบวันที่แบบย่อ ตามการตั้งค่าภาษาของเครื่อง

- เมื่อข้อมูลใน SwiftData เปลี่ยนแปลง ไม่ว่าจะเป็นการเพิ่ม ลบ หรือแก้ไขรายการ ตัวแปร expenses จะถูกอัปเดตโดยอัตโนมัติผ่านกลไก reactive ของ SwiftData และ SwiftUI ทำให้ ForEach สร้าง UI ใหม่ตามข้อมูลล่าสุดทันทีโดยไม่ต้องเขียนโค้ดรีเฟรชหน้าจอเอง นี่คือจุดสำคัญที่ทำให้การแสดงข้อมูลผ่าน ExpenseRowView มีความสอดคล้องกับสถานะจริงของระบบอยู่เสมอ
- นอกจากนี้ List ยังรองรับการลบข้อมูลผ่านการปัด (swipe to delete) โดยคำสั่ง .onDelete จะได้รับตำแหน่งของรายการที่ถูกลบ จากนั้น View จะสร้าง ViewModel และเรียกใช้เมธอด deleteExpense เพื่อจัดการการลบ

ข้อมูลใน persistence อย่างเป็นระบบ การออกแบบนี้ทำให้ View ไม่ต้องจัดการฐานข้อมูลโดยตรง แต่ส่งต่อการกระทำไปยัง ViewModel ซึ่งเป็นไปตามแนวคิด MVVM ที่แยกตรรกะของข้อมูลออกจากการแสดงผล

- ส่วนของ toolbar และ sheet ทำหน้าที่เสริมให้หน้ารายการสามารถเปิดหน้าสำหรับเพิ่มข้อมูลใหม่ได้ โดยใช้ state เป็นตัวควบคุมการแสดงผลของหน้าต่างย่อย แสดงให้เห็นว่า UI ใน SwiftUI ถูกกำหนดโดยสถานะของระบบ ไม่ใช่การเรียกคำสั่งนำเสนอหน้าจอแบบลำดับขั้น
- ในตอนท้ายของโครงสร้าง View มีการใช้ตัวปรับแต่ง .task เพื่อกำหนดงานที่ควรถูกดำเนินการเมื่อ View ปรากฏขึ้นบนหน้าจอ คำสั่งนี้ทำหน้าที่เป็นจุดเริ่มต้นของกระบวนการเตรียมระบบให้พร้อมใช้งาน โดยเฉพาะในกรณีที่แอปเพิ่งถูกเปิดครั้งแรกหรือยังไม่มีข้อมูลในฐานข้อมูล
- ภายใน .task มีการสร้างอ็อบเจกต์ของ ExpenseViewModel โดยส่ง ModelContext เข้าไปผ่าน initializer การกระทำนี้สะท้อนแนวคิดของ Dependency Injection ซึ่งทำให้ ViewModel สามารถเข้าถึงระบบจัดเก็บข้อมูลของ SwiftData ได้โดยไม่ต้องสร้าง context เอง ส่งผลให้การจัดการข้อมูลยังคงอยู่ภายใต้โครงสร้างสถาปัตยกรรมที่ชัดเจน คือ View ทำหน้าที่สั่งงาน ส่วน ViewModel ทำหน้าที่จัดการข้อมูล
- เมื่อได้ ViewModel แล้ว View จะเรียกใช้เมธอด setupInitialData พร้อมส่งรายการ categories และ expenses ที่ดึงมาจาก SwiftData ผ่าน @Query เข้าไป เมธอดนี้มีหน้าที่ตรวจสอบว่าระบบมีข้อมูลพื้นฐานอยู่แล้วหรือไม่ หากยังไม่มี ก็จะสร้างหมวดหมู่เริ่มต้นและข้อมูลตัวอย่างขึ้นมา การออกแบบลักษณะนี้ช่วยให้แอปไม่เริ่มต้นจากหน้าว่าง

ขั้นตอนที่ 8 สร้าง View ชื่อ AddExpenseView และ CategoryPickerView เพื่อการเพิ่มข้อมูลรานการรายจ่ายใหม่

คำสั่งในไฟล์ AddExpenseView.swift

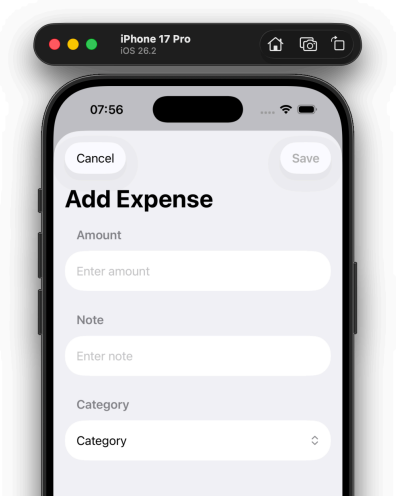
```
import SwiftUI

struct AddExpenseView: View {

    let viewModel: ExpenseViewModel

    @State private var amountText: String = ""
    @State private var note: String = ""
    @State private var selectedCategory: Category?

    @Environment(\.dismiss) private var dismiss

    var body: some View {
        NavigationStack {
            Form {
                Section("Amount") {
                    TextField("Enter amount", text: $amountText)
                        .keyboardType(.decimalPad)
                }
                Section("Note") {
                    TextField("Enter note",text: $note)
                }
                Section("Category") {
                    CategoryPickerView(
                        selectedCategory: $selectedCategory
                    )
                }
            }
            .navigationTitle("Add Expense")

            .toolbar {
                ToolbarItem(placement: .confirmationAction) {
                    Button("Save") {
                        saveExpense()
                    }
                    .disabled(!isValid || viewModel.isSaving)
                }
```

```
                ToolbarItem(placement: .cancellationAction) {
                    Button("Cancel") {
                        dismiss()
                    }
                }
            }
        }
    }
}

extension AddExpenseView {

    private var isValid: Bool {
        Double(amountText) != nil
    }

    private func saveExpense() {

        guard let amount = Double(amountText) else { return }

        viewModel.addExpense(
            amount: amount,
            note: note.isEmpty ? nil : note,
            category: selectedCategory
        )

        dismiss()
    }
}
```

คำสั่งในไฟล์ CategoryPickerView.swift

```
import SwiftUI
import SwiftData

struct CategoryPickerView: View {

    @Query private var categories: [Category]
    @Binding var selectedCategory: Category?

    var body: some View {
       Picker("Category", selection: $selectedCategory) {
            ForEach(categories) { category in
                Text(category.name)
                    .tag(category as Category?)
            }
        }
    }
}
```

- AddExpenseView ทำหน้าที่เป็นหน้าจอสำหรับสร้างข้อมูลรายจ่ายใหม่ในระบบ หน้าที่หลักของ View นี้ คือ รวบรวมข้อมูลจากผู้ใช้ แปลงข้อมูลให้อยู่ในรูปแบบที่เหมาะสม และส่งต่อไปยัง ViewModel เพื่อการเพิ่มข้อมูลรายการรายจ่ายใหม่
- ภายใน View มีการประกาศตัวแปร viewModel เพื่อใช้เป็นตัวกลางในการติดต่อกับชั้นข้อมูล การออกแบบลักษณะนี้ช่วยแยกความรับผิดชอบอย่างชัดเจน กล่าวคือ View ทำหน้าที่รวบรวมข้อมูลจากผู้ใช้ ส่วน ViewModel ทำหน้าที่สร้างและบันทึกข้อมูลในระบบ



- ตัวแปรที่ประกาศด้วย @State ได้แก่ amountText, note และ selectedCategory เป็นตัวแทนของ UI State ซึ่งมีชีวิตอยู่เฉพาะระหว่างการแสดงหน้าจอนี้เท่านั้น ข้อมูลเหล่านี้ยังไม่ใช่ Domain Model แต่เป็นเพียงค่าที่ผู้ใช้กำลังกรอกอยู่

- คำสั่ง @Environment(\.dismiss) private var dismiss คือ กลไกที่ใช้ดึง action สำหรับการปิด View ปัจจุบันจากบริบทของ SwiftUI โดยไม่ต้องพึ่งพาการส่ง closure จาก parent View การออกแบบลักษณะนี้ช่วยลด coupling ระหว่าง View ทำให้ View สามารถจัดการ lifecycle ของตัวเองได้อย่างอิสระ

- ภายใน body มีการใช้ NavigationStack เพื่อกำหนดบริบทของหน้าจอ และใช้ Form เพื่อจัดวางช่องกรอกข้อมูลในลักษณะที่เหมาะสมกับการป้อนข้อมูลของผู้ใช้ ฟอร์มถูกแบ่งเป็นส่วนย่อย ได้แก่ จำนวนเงิน หมายเหตุ และหมวดหมู่ โดยส่วนของหมวดหมู่ใช้ CategoryPickerView ซึ่งเป็น View ย่อยที่ดึงข้อมูลหมวดจาก SwiftData และส่งค่าที่เลือกกลับมาผ่าน Binding การออกแบบนี้ช่วยให้ View หลักไม่ต้องจัดการรายละเอียดของ Picker เอง
- CategoryPickerView เป็น View ย่อยที่ถูกออกแบบให้มีหน้าที่เฉพาะ คือ การแสดงรายการหมวดหมู่ให้ผู้ใช้เลือก การแยก Picker ออกมาเป็นคอมโพเนนต์เฉพาะช่วยลดความซับซ้อนของ AddExpenseView และทำให้โค้ดมีความเป็นโมดูลมากขึ้น
    - คำสั่ง @Query private var categories: [Category] ทำหน้าที่ดึงข้อมูลหมวดหมู่จาก SwiftData
    - คำสั่ง @Binding var selectedCategory: Category? แสดงให้เห็นว่า View นี้ไม่ได้เป็นเจ้าของค่าที่เลือก แต่ได้รับสิทธิ์ในการอ่านและแก้ไขค่าที่อยู่ใน View อื่น ค่าจริงของ selectedCategory นั้นถูกเก็บใน AddExpenseView (ผ่าน @State) ส่วน CategoryPickerView แค่ผูกตัวเองเข้ากับค่านั้น ดังนั้น ค่าที่ถูกผู้ใช้เลือกจาก Picker จะถูกส่งกลับไปยัง AddExpenseView
- extension ในภาษา Swift คือ กลไกที่ใช้ **ขยายความสามารถของชนิดข้อมูล (type)** โดยไม่ต้องแก้ไขนิยามเดิมของชนิดนั้น การแยก isValid และ saveExpense ไปไว้ใน extension ช่วยสร้างความชัดเจนระหว่าง “โครงสร้างของหน้าจอ” กับ “พฤติกรรมของหน้าจอ” กล่าวคือ ภายใน struct หลักจะเห็นเฉพาะการจัดวาง UI และการเชื่อมโยง state ขณะที่ extension ทำหน้าที่รวบรวมพฤติกรรมที่เกี่ยวข้องกับการตรวจสอบและการบันทึกข้อมูล การจัดวางเช่นนี้สอดคล้องกับหลัก Separation of Concern ซึ่งมุ่งแยกความรับผิดชอบของแต่ละส่วนออกจากกันอย่างชัดเจน
- เมื่อผู้ใช้กดปุ่ม Save ฟังก์ชัน saveExpense จะทำหน้าที่ตรวจสอบความถูกต้องของข้อมูล ผ่านคำสั่ง guard let amount = Double(amountText) else { return } จากนั้น จึงเรียกใช้ viewModel.addExpense(...) เพื่อสร้าง Domain Model ตัวจริง และบันทึกลง SwiftData ผ่าน ModelContext
- การควบคุมการแสดงสถานะของปุ่ม Save
    - ตัวแปร isValid เป็น computed property ที่ตรวจสอบว่า ค่าที่ผู้ใช้กรอกสามารถแปลงเป็นตัวเลขได้หรือไม่ หากข้อมูลไม่ถูกต้อง ปุ่มจะถูกปิดการใช้งานทันที นี่คือ การทำ validation ในระดับ UI เพื่อป้องกันข้อผิดพลาดก่อนส่งข้อมูลเข้าสู่ระบบ
    - viewModel.isSaving เป็นตัวแปรนี้สะท้อนสถานะการทำงานของระบบจาก ViewModel หากกำลังบันทึกข้อมูลอยู่ ปุ่ม Save จะถูกปิดการใช้งานชั่วคราว เพื่อป้องกันการกดซ้ำ ซึ่งอาจนำไปสู่การสร้างข้อมูลซ้ำหรือเกิดความไม่สอดคล้องของข้อมูล

- การเรียก dismiss( ) หลังจากบันทึกสำเร็จแสดงให้เห็นว่า View ทำหน้าที่เพียงควบคุมการแสดงผลของหน้าจอ ไม่ได้เกี่ยวข้องกับการจัดการข้อมูลในระดับ persistence

การพัฒนาแอปในโปรเจกต์นี้ไม่ได้มุ่งเน้นเพียงการสร้างหน้าจอหรือทำให้โค้ดทำงานได้เท่านั้น แต่เน้นการทำความเข้าใจโครงสร้างของข้อมูล การไหลของข้อมูล และการจัดวางความรับผิดชอบของแต่ละส่วนในระบบอย่างเป็นระบบ หัวใจสำคัญของการออกแบบแอปด้วย SwiftUI อยู่ที่การเข้าใจความสัมพันธ์ระหว่าง Data Flow, State และ Model โดย Model ทำหน้าที่แทนข้อมูลเชิงโดเมนที่มีความหมายในระยะยาว ขณะที่ State เป็นตัวแทนของสถานะปัจจุบันที่ควบคุมการแสดงผลของ UI ความเข้าใจความแตกต่างระหว่างสองสิ่งนี้ช่วยให้ผู้พัฒนาสามารถออกแบบระบบที่แยกความรับผิดชอบได้ชัดเจน ลดการผูกติดระหว่างข้อมูลกับหน้าจอ และทำให้โค้ดมีความยืดหยุ่นมากขึ้น

ขั้นสรุปบทเรียน (Conclusion)

การสรุปบทเรียนในหัวข้อ **Data Modeling & SwiftData** ไม่ควรเป็นเพียงการทบทวนคำสั่งหรือขั้นตอนทางเทคนิค หากควรมุ่งสะท้อน “กระบวนการคิด” ที่ผู้เรียนได้พัฒนาขึ้นตลอดกิจกรรม โดยเฉพาะการเปลี่ยนผ่านจากแนวคิดแบบ UI-Driven ไปสู่ Data-Driven ภายใต้กรอบ **Thinking in Data** ประเด็นสำคัญที่ควรเน้นย้ำคือ การออกแบบระบบที่ดีเริ่มต้นจากความเข้าใจข้อมูล ไม่ใช่เริ่มจากการออกแบบหน้าจอ

ผู้สอนสามารถเริ่มต้นการสรุปด้วยคำถามสะท้อน เช่น “ในช่วงต้นคาบ เราเริ่มต้นคิดถึงอะไรเป็นอันดับแรก — หน้าจอ หรือ ข้อมูล?” และ “เมื่อเราถูกถามว่า ‘ระบบต้องจัดการข้อมูลอะไร’ ความคิดของเราเปลี่ยนไปอย่างไร?” การย้อนกลับไปยังช่วง Warm-up ซึ่งผู้เรียนมักตอบในมุมมองของ UI ก่อนจะค่อย ๆ ปรับมุมมองไปสู่การวิเคราะห์ **Entity, Attribute และ Relationship** จะช่วยทำให้เห็นพัฒนาการทางความคิดอย่างชัดเจน การออกแบบโครงสร้างข้อมูลของแอปโดยการสร้างโมเดลของ Expense และ Category ไม่ได้เกิดจากการคาดเดาเชิงเทคนิค แต่เกิดจากการวิเคราะห์โลกจริง เช่น เหตุการณ์การใช้จ่าย จำนวนเงิน วันที่ และหมวดหมู่ ซึ่งสะท้อนกระบวนการสร้าง Conceptual Data Model อย่างเป็นระบบ

การทดลองสร้างโปรเจกต์ **InMemoryDemo** และ **PersistentModelDemo** ทำให้ผู้เรียนตระหนักอย่างชัดเจนว่า “พฤติกรรมของข้อมูล” ไม่ได้ขึ้นอยู่กับหน้าตาของแอปหรือโค้ด UI หากขึ้นอยู่กับบทบาทที่ข้อมูลถูกออกแบบให้มีในระบบ

- ในกรณีของ **In-memory Model** ข้อมูลมีชีวิตอยู่เพียงในหน่วยความจำ (RAM) ระหว่างที่แอปกำลังทำงาน ไม่มีการจัดเก็บถาวร ไม่มีระบบติดตามวงจรชีวิตของอ็อบเจกต์ และไม่มี identity ในระดับระบบ รูปแบบนี้เหมาะสำหรับ UI State หรือข้อมูลชั่วคราว แต่ไม่เหมาะสำหรับข้อมูลเชิงโดเมนที่ต้องคงอยู่ระยะยาว

- ในทางตรงกันข้าม การจัดเก็บข้อมูลแบบ **Persistent Model** โดยใช้ class ร่วมกับ @Model และ SwiftData แสดงให้เห็นว่า ข้อมูลไม่ได้เป็นเพียงค่าที่อยู่ในตัวแปรอีกต่อไป แต่ถูกยกระดับเป็น **Persistent Domain Object** ที่มี identity มี lifecycle และถูกจัดการผ่าน ModelContext ภายใต้กลไก persistence ของ SwiftData เพื่อรองรับการเก็บข้อมูลในระยะยาว

ในกระบวนการพัฒนา ผู้เรียนได้เห็นบทบาทของเครื่องมือสำคัญใน SwiftUI ได้แก่ @State ซึ่งใช้เก็บสถานะภายใน View, @Binding ซึ่งใช้เชื่อม State ระหว่าง View ต่าง ๆ และ @Observable ซึ่งช่วยให้ ViewModel สามารถแจ้งเตือน View เมื่อสถานะภายในเปลี่ยนแปลง กลไกเหล่านี้ทำงานร่วมกันเพื่อสร้างการไหลของข้อมูลแบบทิศทางเดียว (Unidirectional Data Flow) ซึ่งเป็นหลักการสำคัญของสถาปัตยกรรมสมัยใหม่

ขณะเดียวกัน การนำ SwiftData มาใช้ทำให้ Data Model ถูกยกระดับจากข้อมูลชั่วคราวในหน่วยความจำไปสู่ Persistent Domain Object ที่มีตัวตนในระบบจริง ผู้เรียนจึงได้เข้าใจว่า การเลือกใช้ class และ @Model ไม่ได้เป็นเพียงข้อกำหนดของ framework แต่เป็นการตัดสินใจเชิงแนวคิดที่เกี่ยวข้องกับ identity, lifecycle และความสอดคล้องของข้อมูลในระดับแอปพลิเคชัน

เมื่อเชื่อม SwiftUI กับ SwiftData ผ่าน @Query และ ModelContext ระบบจึงสามารถแสดงผลข้อมูลและตอบสนองต่อการเปลี่ยนแปลงได้โดยอัตโนมัติ โดย View ทำหน้าที่แสดงผล ViewModel ทำหน้าที่จัดการการกระทำต่อข้อมูล และ Model ทำหน้าที่แทนโครงสร้างของข้อมูลจริง การจัดวางบทบาทลักษณะนี้สะท้อนสถาปัตยกรรม MVVM อย่างชัดเจน และช่วยให้โค้ดมีความเป็นระเบียบ อ่านง่าย และสามารถขยายต่อในอนาคตได้อย่างมั่นคง

กล่าวโดยสรุป บทเรียนนี้ไม่ได้มุ่งให้ผู้เรียนเพียง “เขียนโค้ดได้” หากมุ่งให้ผู้เรียนเข้าใจว่าการออกแบบข้อมูลคือรากฐานของการออกแบบระบบ และการตัดสินใจเกี่ยวกับ Data Model คือการตัดสินใจเชิงสถาปัตยกรรมที่กำหนดศักยภาพของแอปในระยะยาว

คำถามเพื่อการสะท้อนผลการเรียนรู้จากการทำกิจกรรม

1. อธิบายความหมายของแนวคิด Thinking in Data และอธิบายว่ามุมมองของคุณต่อการเริ่มต้นพัฒนาแอปเปลี่ยนไปอย่างไรหลังทำกิจกรรมนี้
2. อธิบายความแตกต่างระหว่างการออกแบบแอปแบบ UI-Driven กับ Data-Driven พร้อมยกตัวอย่างจากกิจกรรม
3. อธิบายความแตกต่างระหว่าง Value Semantics และ Reference Semantics พร้อมยกตัวอย่างประกอบ
4. อธิบายความแตกต่างระหว่าง In-memory Model และ Persistent Model พร้อมยกตัวอย่างประกอบ
5. คำว่า identity ของข้อมูลในบริบทของ SwiftData หมายถึงอะไร และเหตุใดจึงสำคัญ
6. การแยกไฟล์เป็น Models, ViewModels และ Views ช่วยให้ระบบขยายต่อได้อย่างไร
7. จากกิจกรรมทั้งหมด คุณคิดว่าการออกแบบ Data Model มีผลต่อความยั่งยืนของระบบในระยะยาวอย่างไร

คำศัพท์สำคัญ

Object-Relational Mapping (ORM)
แนวคิดและเทคนิคในการเชื่อมโยงโลกของ **อ็อบเจกต์ในโปรแกรมเชิงวัตถุ (Object-Oriented Programming)** เข้ากับ **โครงสร้างข้อมูลแบบฐานข้อมูลเชิงสัมพันธ์ (Relational Database)** เพื่อให้นักพัฒนาสามารถทำงานกับข้อมูลในรูปแบบของอ็อบเจกต์ แทนการเขียนคำสั่ง SQL โดยตรง

Data Model
โครงสร้างเชิงนามธรรมที่ใช้แทนข้อมูลในโลกจริง โดยกำหนดชนิดข้อมูล (Entity), คุณลักษณะ (Attribute) และความสัมพันธ์ (Relationship) เพื่อให้ระบบสามารถจัดเก็บและจัดการข้อมูลได้อย่างมีความหมาย

Domain Model
โมเดลที่แทนแนวคิดหลักของระบบ ซึ่งสะท้อนถึงสิ่งที่มีอยู่จริงในระบบ ความสัมพันธ์ระหว่างสิ่งเหล่านั้น และความหมายของข้อมูลในเชิงธุรกิจ

Predicate
เงื่อนไขเชิงตรรกะที่ใช้ตรวจสอบว่า “ข้อมูลหนึ่งรายการตรงตามเงื่อนไขหรือไม่” โดยผลลัพธ์ของ predicate จะเป็นค่าแบบบูลีน คือ **จริง (true)** หรือ **เท็จ (false)** ใน SwiftData predicate ถูกใช้กับ @Query เพื่อระบุว่า ต้องการดึงข้อมูลแบบไหนจาก persistent store

In-memory Model
รูปแบบการจัดเก็บข้อมูลที่มีชีวิตอยู่เฉพาะในหน่วยความจำ (RAM) ระหว่างที่โปรแกรมทำงาน ไม่มีการจัดเก็บถาวร และไม่มีการจัดการวงจรชีวิตในระดับระบบ

Persistent Model
แบบจำลองข้อมูลที่ถูกจัดเก็บอย่างถาวรในระบบจัดเก็บข้อมูล (persistent store) มี identity ระยะยาว และมีการจัดการ lifecycle อย่างเป็นระบบ

Identity
คุณลักษณะที่ทำให้อ็อบเจกต์หนึ่งสามารถถูกระบุว่าเป็น “ตัวเดิม” แม้ค่าภายในจะเปลี่ยนแปลง เป็นแนวคิดสำคัญในระบบ persistence และ reference semantics

Value Semantics
รูปแบบการทำงานของข้อมูลที่เมื่อถูกกำหนดให้ตัวแปรใหม่ จะเกิดการคัดลอกค่า (copy) ทำให้แต่ละตัวแปรเป็นอิสระจากกัน (พบใน struct)

Reference Semantics
รูปแบบการทำงานที่ตัวแปรหลายตัวสามารถอ้างอิงอ็อบเจกต์เดียวกันได้ การเปลี่ยนแปลงผ่านตัวแปรหนึ่งจะสะท้อนในทุกตัวแปรที่อ้างอิง (พบใน class)

ModelContext
ตัวกลางที่ใช้จัดการการสร้าง บันทึก แก้ไข และลบข้อมูลใน SwiftData ทำหน้าที่คล้าย transaction layer ของระบบ persistence

Inverse Relationship
ความสัมพันธ์สองทิศทางที่ระบบจัดการความสอดคล้องของข้อมูลให้อัตโนมัติ เช่น เมื่อ Expense ถูกผูกกับ Category ระบบจะอัปเดตฝั่ง Category ให้สอดคล้องกัน

State
ข้อมูลที่สะท้อน “สถานะปัจจุบัน” ของ UI มักมีอายุสั้นและเปลี่ยนแปลงตามการโต้ตอบของผู้ใช้
Unidirectional Data Flow

แนวคิดที่ข้อมูลไหลในทิศทางเดียวจากแหล่งข้อมูล (Source of Truth) ไปยัง UI ลดความซับซ้อนและปัญหาความไม่สอดคล้องของข้อมูล

Separation of Concern
หลักการออกแบบซอฟต์แวร์ที่แยกความรับผิดชอบของแต่ละส่วนออกจากกันอย่างชัดเจน เช่น แยก Model, View และ ViewModel

Single Source of Truth
หลักการที่กำหนดให้ข้อมูลหนึ่งค่ามีเจ้าของเพียงจุดเดียวในระบบ และ View อื่นเข้าถึงผ่าน Binding หรือกลไกที่ควบคุมได้

@Model
Macro ใน SwiftData ที่ใช้ประกาศว่า class หนึ่งเป็น Persistent Domain Object ซึ่งจะถูกจัดการ identity และ lifecycle โดยระบบ

@Query
กลไกการดึงข้อมูลเชิงประกาศ (Declarative Data Access) ใน SwiftData ที่ทำให้ View สามารถเข้าถึงข้อมูลจาก persistent store ได้โดยอัตโนมัติ

@State
Property wrapper ใน SwiftUI สำหรับเก็บสถานะภายใน View ซึ่งมีชีวิตอยู่เท่ากับ lifecycle ของ View

@Binding
กลไกที่ใช้เชื่อมโยง State ระหว่าง View เพื่อรักษาหลัก Single Source of Truth และสนับสนุนการไหลของข้อมูลแบบทิศทางเดียว

รายละเอียดสำหรับการอ้างอิงเอกสารชุดนี้
- **ผู้เขียน** ธิติ ธีระเธียร
- **วันที่เผยแพร่** วันที่ 10 กุมภาพันธ์ 2569
- **เข้าถึงได้จาก** https://ajthiti.gitbook.io/develop-in-swift/swiftdata/thinking-in-data
- **เงื่อนไขในการใช้งาน** This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.