Week4 -layered

เสนอ

อาจารย์ ธนิต เกตุแก้ว

จัดทำโดย

นายศุภโชค แสงจันทร์ 6754321006-6

เอกสารฉบับนี้เป็นส่วนหนึ่งของรายวิชา

ENGSE207 Software Architecture

ภาคเรียนที่ 2 ปีการศึกษา 2568

มหาวิทยาลัยเทคโนโลยีราชมงคลล้านนา เชียงใหม่

การวิเคราะห์เปรียบเทียบ Monolithic vs Layered

ข้อมูล	Monolithic (Week 3)	Layered (Week 4)
จำนวนไฟล์ JS หลัก	1-3 ไฟล์ (เช่น app.js)	4-6 ไฟล์ (Controller, Service, Model, Router)
จำนวนบรรทัดทั้งหมด	200-300 บรรทัด	250-350 บรรทัด (รวมทุก layer)
จำนวน layers	1 layer (รวมทุกฟังก์ชันในไฟล์เดียว)	3-4 layers (Presentation, Business Logic, Data Access)
ความซับซ้อนโดยรวม	ต่ำ-ปานกลาง แต่ยากต่อการขยาย	ปานกลาง-สูง แต่ชัดเจนต่อการบำรุงรักษา

จำนวนไฟล์และบรรทัดโค้ด: Layered vs Monolithic

- **จำนวนไฟล์**: **Layered มักมีจำนวนไฟล์มากกว่า Monolithic**
    - เพราะ Layered แยกโค้ดออกเป็นแต่ละชั้น เช่น **Controller, Service, Model, Router**
    - Monolithic รวมทุกฟังก์ชันในไฟล์หลักเพียงไฟล์เดียวหรือสองไฟล์
- **จำนวนบรรทัดโค้ด**: **Layered อาจมากกว่าหรือน้อยกว่าขึ้นกับความซ้ำซ้อน**
    - บางครั้ง Layered มีโค้ด boilerplate (เช่น การเชื่อมต่อแต่ละ layer, import/export) ทำให้บรรทัดรวมสูงขึ้น
    - แต่ Layered ลดการซ้ำซ้อนของ logic ภายในแต่ละ layer ทำให้โค้ดสะอาดกว่า

สรุป: โดยรวม **Layered มักมีไฟล์และบรรทัดโค้ดมากกว่า Monolithic** เพราะโค้ดถูกแยกชัดเจนเป็นหลาย laye

ความซับซ้อนที่เพิ่มขึ้นคุ้มค่าหรือไม่?

คุ้มค่าแน่นอนในโปรเจกต์ขนาดกลางถึงใหญ่ เพราะ:

1. **การบำรุงรักษาง่ายขึ้น**
    - แก้ไขหรือเพิ่มฟีเจอร์ใน Layer ใด ๆ ไม่กระทบ Layer อื่นมาก
    - ลดโอกาสเกิด bug ที่กระทบหลายส่วน
2. **สามารถทดสอบแยกส่วนได้ (Unit Testing)**
    - Layered ออกแบบให้แต่ละ layer รับผิดชอบงานเฉพาะ ทำให้เขียน test ได้ง่าย
3. **รองรับการขยายและทีมหลายคน**
    - ทีมสามารถทำงานบน layer ต่าง ๆ พร้อมกันโดยไม่ชนกัน

ข้อเสีย:

- ต้องจัดการหลายไฟล์และหลาย layer ⟶ เริ่มต้นอาจซับซ้อนกว่า Monolithic
- มี boilerplate เพิ่มขึ้น

สรุป:

- สำหรับโปรเจกต์เล็ก ๆ Monolithic อาจพอเพียง
- สำหรับโปรเจกต์ขนาดกลางขึ้นไป **Layered เพิ่มความซับซ้อนเล็กน้อย แต่คุ้มค่าเพราะความชัดเจนและสามารถขยายได้ง่าย**

วิเคราะห์จุดแข็งของ Layered Architecture

Quality Attribute	Monolithic	Layered	อธิบายเหตุผล / ตัวอย่างโค้ด
Maintainability (ความง่ายในการดูแล)	3	5	Layered แยกฟังก์ชันออกเป็น **Controller, Service, Model** ทำให้แก้ไขส่วนใดก็ได้โดยไม่กระทบส่วนอื่น เช่น ถ้าแก้ logic การคำนวณใน Service ไม่กระทบ Controller หรือ Model ใน Monolithic ทุกอย่างอยู่ใน app.js การแก้ไขฟังก์ชันหนึ่งอาจทำให้โค้ดส่วนอื่นเสีย
Testability (ความง่ายในการทดสอบ)	2	5	Layered ทำให้สามารถทดสอบ Unit ของแต่ละ Layer ได้ เช่น เขียน Unit Test สำหรับ Service โดยไม่ต้องเรียก Controller หรือ Database ใน Monolithic การแยก test ยาก เพราะทุกอย่างผสมกันอยู่
Modifiability (ความง่ายในการแก้ไข)	3	5	การเพิ่มฟีเจอร์ใหม่ เช่น เพิ่มฟังก์ชันค้นหาข้อมูล สามารถเพิ่มเฉพาะ Layer ที่เกี่ยวข้อง (Service + Controller) โดยไม่กระทบ Layer อื่น Monolithic ต้องแก้ไขไฟล์เดียวที่รวมหลาย logic ⟶ เสี่ยงเกิด bug
Reusability (การนำกลับมาใช้ใหม่)	2	5	Layered ช่วยให้ Service หรือ Model สามารถนำไปใช้ซ้ำในฟีเจอร์อื่นได้ เช่น UserService.getUserById() ถูกเรียกจากหลาย Controller ได้ง่าย Monolithic ฟังก์ชันมักเขียนเฉพาะที่ ใช้ซ้ำยาก
Team Collaboration (การทำงานเป็นทีม)	2	5	Layered แยกหน้าที่ชัดเจน ทีมหนึ่งทำ Controller อีกทีมทำ Service/Model ได้พร้อมกัน ลด conflict ใน Git Monolithic ทีมต้องแก้ไฟล์เดียวกัน ⟶ conflict บ่อย

Performance (ประสิทธิภาพ)	5	3	Monolithic อาจโหลดเร็วกว่า เพราะไม่มีการแยก Layer แต่ Layered มี overhead ในการเรียก function ข้าม Layer เล็กน้อย เช่น Controller เรียก Service ⟶ function call เพิ่มขึ้น แต่โดยทั่วไป overhead น้อยมากในเว็บแอปขนาดปกติ
Simplicity (ความเรียบง่าย)	5	3	Monolithic เรียบง่าย เห็นทุก logic ในไฟล์เดียว Layered ต้องเข้าใจ flow ของหลาย Layer ก่อน ทำให้ซับซ้อนสำหรับผู้เริ่มต้น แต่เป็นความซับซ้อนที่มีโครงสร้างและเข้าใจง่ายเมื่อทำงานต่อเนื่อง

สถานการณ์จริง

<u>สถานการณ์ที่ 1: ต้องการเพิ่มฟีเจอร์ "assign task to user"</u>

ใน Monolithic จะต้องแก้ไขอย่างไร?

โค้ดอยู่ในไฟล์เดียว (app.js หรือไฟล์หลัก)

การเพิ่มฟีเจอร์นี้ต้องทำหลายอย่างในไฟล์เดียว เช่น:

1. เพิ่ม UI/Endpoint ในส่วน Controller
2. เขียน logic การเชื่อมโยง Task กับ User ในฟังก์ชันเดียวกับที่มีอยู่แล้ว
3. แก้ไข Data Handling / Database Query ในฟังก์ชันเดียว

ข้อเสีย:

- การแก้ไขหลายจุดในไฟล์เดียว ⟶ เสี่ยงทำให้โค้ดเดิมเสีย
- หากระบบใหญ่ขึ้น ⟶ การค้นหาจุดที่ต้องแก้ไขทำได้ยาก

ใน Layered จะต้องแก้ไขอย่างไร?

แยกหน้าที่เป็น Layer: **Controller, Service, Model**

การเพิ่มฟีเจอร์ต้องแก้ไข **เฉพาะ Layer ที่เกี่ยวข้อง**:

1. **Controller:** เพิ่ม endpoint หรือฟังก์ชันรับ request
2. **Service:** เขียน logic "assign task to user"
3. **Model:** หากต้องแก้โครงสร้างข้อมูล (เช่น Task model)

ข้อดี:

- แต่ละ Layer แก้ไขแยกส่วน ⟶ ลดผลกระทบต่อ Layer อื่น
- สามารถเขียน Unit Test แยก Layer ได้

แบบไหนง่ายกว่า? เพราะอะไร?

ประเด็น	Monolithic	Layered	ผลสรุป
การแก้ไขไฟล์	แก้ไขไฟล์เดียว	แก้ไขหลายไฟล์ตาม Layer	Monolithic เริ่มต้นง่าย แต่ Layered แยกความรับผิดชอบชัดเจน
ความเสี่ยงกระทบส่วนอื่น	สูง (แก้ไขฟังก์ชันเดียวอาจกระทบฟีเจอร์อื่น)	ต่ำ (แก้ไขเฉพาะ Layer)	Layered ปลอดภัยกว่า
การทดสอบ	ยาก ต้องเรียกทั้งฟังก์ชัน	ง่าย สามารถ Unit Test Layer	Layered ดีต่อการทดสอบ
การขยายในอนาคต	ยาก	ง่าย	Layered เหมาะสำหรับฟีเจอร์เพิ่มขึ้น

<u>สถานการณ์ที่ 2: มีบั๊กที่ validation logic (ตรวจสอบ title)</u>

ใน Monolithic จะต้องหาบั๊กและแก้ไขที่ไหน?

Monolithic

- **โค้ดทุกอย่างรวมอยู่ในไฟล์เดียว (app.js)**
- บั๊ก validation อาจอยู่ในฟังก์ชันสร้าง/แก้ไข task

- การแก้ไขบั๊กต้องทำใน **ฟังก์ชันเดียวที่มีทุก logic**
- **ข้อเสีย:**
    - ต้องอ่านโค้ดทั้งหมดของฟังก์ชัน เพราะ logic validation, DB, และ UI อยู่รวมกัน
    - ถ้าฟังก์ชันใหญ่ ⟶ ใช้เวลาหา bug นาน

ใน Layered จะต้องหาบั๊กและแก้ไขที่ไหน?

- แยกเป็น **Controller, Service, Model**

- Validation logic มักอยู่ที่ **Service Layer** หรือ **Model Layer**
- การหาบั๊กง่ายกว่าเพราะ **โค้ด validation ถูกแยกไว้ในไฟล์/ฟังก์ชันเฉพาะ**
- สามารถ **unit test validation แยกจาก Layer อื่น** ได้

แบบไหนง่ายกว่า? เพราะอะไร?

ประเด็น	Monolithic	Layered	ผลสรุป
การหาบั๊ก	ต้องดูฟังก์ชันใหญ่รวมทุก logic	ดูเฉพาะ Service/Validation Layer	Layered ง่ายกว่า
การแก้ไข	เสี่ยงกระทบฟีเจอร์อื่น	แก้ไขเฉพาะ Layer ที่เกี่ยวข้อง	Layered ปลอดภัยกว่า
การทดสอบ	ต้องทดสอบฟังก์ชันรวมทั้ง Controller/DB	Unit Test เฉพาะ validation ได้	Layered ง่ายกว่า

<u>สถานการณ์ที่ 3: ต้องการเปลี่ยนจาก SQLite เป็น PostgreSQL</u>

ใน Monolithic จะต้องแก้ไขกี่ที่?

- **โค้ดทุกอย่างรวมอยู่ในไฟล์เดียว**
- Database logic มัก **กระจายอยู่หลายจุด**
- การเปลี่ยนเป็น PostgreSQL ต้องแก้ไข **ทุกจุดที่เรียก SQLite API** และอาจต้องแก้ query syntax

  ข้อเสีย:
  - จำนวนจุดแก้ไขขึ้นกับขนาดโปรเจกต์ ⟶ เสี่ยงเกิด bug
  - ต้องแก้ทั้ง connection, query, transaction
  -

ใน Layered จะต้องแก้ไขกี่ที่?

- แยกเป็น Model/Repository Layer (หรือ Data Access Layer)
- Database logic ถูก รวมอยู่ใน Model Layer
- การเปลี่ยน DB แค่ แก้ไขที่ Layer เดียว (Model/Repository Layer)
- Service/Controller ไม่ต้องแก้ไขอะไร

แบบไหนง่ายกว่า? เพราะอะไร?

ประเด็น	Monolithic	Layered	ผลสรุป
จำนวนจุดแก้ไข	หลายจุด (ทุก function ที่ใช้ DB)	1 Layer (Model/Repository)	Layered ง่ายกว่า
ความเสี่ยงกระทบส่วนอื่น	สูง (อาจกระทบ logic อื่นที่ใช้ DB)	ต่ำ (Controller/Service ไม่ต้องแก้)	Layered ปลอดภัยกว่า
การทดสอบ	ต้องทดสอบทั้งโปรเจกต์	Test แค่ Model Layer	Layered ง่ายกว่า

<u>คำถาม 4: Trade-offs (5 คะแนน)</u>

วิเคราะห์ Trade-offs: Complexity vs Maintainability

Trade-off: Complexity vs Maintainability

Observation:

- **Layered Architecture** ทำให้โค้ด **ซับซ้อนขึ้น** เพราะต้องแยกหลาย Layer (Controller, Service, Model, Validation ฯลฯ)
- แต่ **Maintainability ดีขึ้น** อย่างชัดเจน เพราะโค้ดแต่ละ Layer รับผิดชอบงานเฉพาะ ทำให้แก้ไข เพิ่มฟีเจอร์ หรือทดสอบแยกส่วนได้ง่าย

ความคิดเห็น:

- **คุ้มค่าหรือไม่: คุ้มค่า**
  - เหมาะกับ **โปรเจกต์ขนาดกลางถึงใหญ่**
  - ทีมหลายคนสามารถทำงานพร้อมกันโดยไม่ชนกัน
  - การแก้ไขบั๊กหรือเปลี่ยนเทคโนโลยี (เช่น Database, Validation Logic) ทำได้ง่ายและปลอดภัย
- **กรณีที่คุ้มค่า:**
  - แอปพลิเคชันมีหลายฟีเจอร์
  - ต้องทำงานร่วมกันหลายคน
  - ต้องดูแลและขยายโปรเจกต์ต่อเนื่อง
- **กรณีที่ไม่คุ้มค่า:**
  - แอปเล็ก มีฟีเจอร์ไม่กี่อย่าง
  - ทีมเล็กหรือทำคนเดียว
  - การเพิ่ม Layer อาจทำให้โค้ด **ดูซับซ้อนเกินความจำเป็น** และเริ่มต้นช้ากว่า Monolithic

สรุป:

- Layered เพิ่ม **Complexity เล็กน้อย** แต่แลกมาด้วย **Maintainability, Testability, Modifiability, Reusability** ที่ดีขึ้น
- สำหรับโปรเจกต์ใหญ่และยาวนาน trade-off นี้ **คุ้มค่า**
- สำหรับโปรเจกต์เล็กและชั่วคราว trade-off อาจไม่คุ้มค่า

Performance Overhead

ประเด็น	Monolithic	Layered	สรุป / หมายเหตุ
Function Call	โค้ดอยู่ไฟล์เดียว เรียกตรง	เรียกข้าม Layer (Controller ⟶ Service ⟶ Model)	Layered มี overhead เล็กน้อย
ผลกระทบต่อ Performance	ต่ำมาก / ไม่มี noticeable delay	น้อยมากในเว็บแอปทั่วไป	เว็บทั่วไปแทบไม่สังเกต
แอปที่สำคัญต่อ Overhead	ทุกแอป	Real-time, High-performance, Embedded, IoT	ต้อง optimize Layer หรือใช้ approach อื่น
เหมาะสมสำหรับ	เว็บแอปขนาดเล็ก-กลาง	เว็บแอปทั่วไป, โปรเจกต์ใหญ่ที่ต้อง maintain	Layered คุ้มค่า maintainability มากกว่า overhead

<u>คำถาม 5: การตัดสินใจเลือกใช้</u>

<u>ขนาดทีม?</u>

1-2 คน ⟶ Monolithic ทีมเล็ก ดูแลไฟล์เดียวง่าย , ลด overhead ในการจัดการหลาย Layer

3+ คน ⟶ Layered ทีมใหญ่ต้องทำงานพร้อมกันหลายส่วน,Layered ช่วย ลด conflict และแยกความรับผิดชอบชัดเจน

<u>ขนาดโปรเจกต์?</u>

เล็ก (< 1000 บรรทัด) ⟶ Monolitich โปรเจกต์เล็ก ไม่มีความซับซ้อนมาก

กลาง (1000-10000 บรรทัด) ⟶ Layered โปรเจกต์ใหญ่ขึ้น โค้ดเยอะ แยก Layer ลดความซ้ำซ้อนและง่ายต่อการบำรุงรักษา

ใหญ่ (> 10000 บรรทัด) ⟶ Layered Layered Architecture จำเป็น เพราะช่วยจัดการ complexity, maintainability และทีมงานจำนวนมาก

<u>ระยะเวลาพัฒนา?</u>

ต้องการเร็ว (< 1 เดือน) ⟶ Monolitch ริ่มเร็ว เหมาะกับโปรเจกต์สั้น/ทดลอง

มีเวลา (> 1 เดือน) ⟶ Layered ใช้เวลาเพิ่มเล็กน้อย แต่แลกกับ maintainability, testability, และ scalability ที่ดีขึ้น

<u>ต้องการ maintainability สูง?</u>

ใช่ ⟶ Layered แก้ไข เพิ่มฟีเจอร์ หรือ refactor ง่าย

ไม่ ⟶ Monolitch โค้ดรวมไฟล์เดียว เรียบง่าย , เหมาะกับโปรเจกต์สั้น, ทีมเล็ก, หรือทดลอง