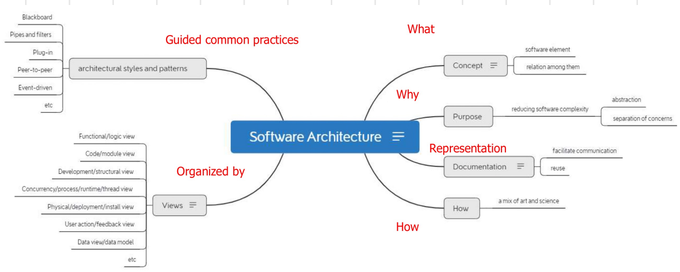
Week 1

ตัวแทนของสิ่งที่เราสนใจ คือ model — แสดงเป็น → Diagram — มีหลาย → View (มีหลายมุมมอง)
สิ่งที่เราสนใจ; Viewpoint แม่แบบของ View

Introduction



Software Architecture as an Architectural Design

Related to Abstraction!

Where Architecture Fits

- Requirements (Functional, Non-Functional)
  - What the system should do
  - What properties it should have
- Architecture
  - High-level design, how properties are achieved
- Detailed design
  - Lower-level design, how system functions
- Code
  - How the system actually works

What ↕ How

เราส่งผ่าน input/output ของกิจกรรมต่าง ๆ ด้วย Model

Architectural Design (High level)

ยังเป็นนามธรรมมาก ๆ รายละเอียดน้อย

- Level of Abstraction
  - Top-down (High-to-Low level of abstraction) approach
- Architectural Design
  - Subsystem Design
- Complexity of the Design **สอบ**
  - Hiding and Omitting to Reduce the Complexity

ต้องเป็น High-level เท่านั้น (ไม่ควรใช้ Low-level)

สอบ!

- Detailed Design อาจจะมีรายละเอียดมากเกินไป
  - Object Design

based on degree of precision and ความเข้มงวด

Model คือตัวแทนสิ่งที่เรากำลังสนใจ ที่เราใช้เพื่อนำเสนอหรือสื่อสารกัน (เพราะจัดหาของจริงมาไม่สะดวก)

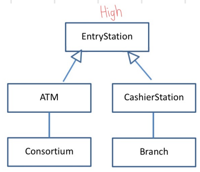
คำถามที่ควรตอบได้

- Model คืออะไร → ตัวแทนของสิ่งที่เราสนใจ
- Model สำคัญอย่างไร → เพราะ model ใช้ในการสื่อสาร
- Informal model → Semi formal model → Formal model คืออะไร

เช่น การใช้ภาษาบรรยายเอา (ใช้ภาษาธรรมชาติ); เช่น การใช้ graphic + description (พวก UML Diagram) ← พวก syntax + บรรยายนิดหน่อย

เช่น ใช้ math model อาทิ Regular Expression (ตีความได้อย่างเดียว ไม่กำกวม แต่เข้าใจยาก)

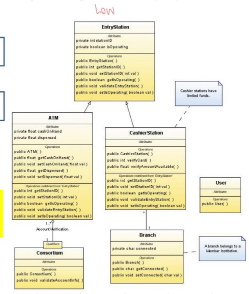
High



No details of classes

Hiding and Omitting to Reduce the Complexity of the Model

Low



Q: เวลาแก้ไขเปลี่ยนแปลง scope ภาพไหนแก้ไขง่ายกว่า?

ตอบ High แก้ง่ายกว่า

About Software Architecture

- The extension of software engineering discipline
- Software Architecture presents a view of software system as components and connectors
  - Components encapsulate some coherent set of functionality
  - Connectors realize the runtime interaction between components

Architecture เน้น Abstraction level สูง

นิยามของ Software Architecture (สอบ!)

IEEE: "Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution."

Bass, Clements, Kazman: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

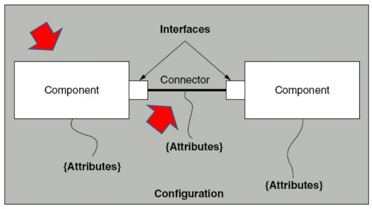


Figure 12-1. Software architecture concept.

- สำหรับ Software Architecture เราใช้ model ที่เขียนด้วย
  - UML Subsystem diagram (design-time) ← ก่อนระบบทำงานได้จริง
  - UML Component diagram (run-time) ← ระบบทำงานได้จริงแล้ว

SW Arch ควรมีกี่ diagram? ← ควรมีหลาย Diagram (ห้ามมี 1 Diagram)

แต่ละ diagram แสดงถึงมุมมอง View ที่แตกต่างกัน ไม่จำเป็นต้องใช้แค่ diagram เดียว

In summary, a model is an abstract representation of a system, and a diagram is a specific visual representation of a model. Diagrams serve as tools for communicating and visualizing models, making them more accessible to various stakeholders in the software development process.

Software Architecture Description

คำอธิบาย

= A set of practice for expressing, communicating, and analyzing software architecture.

- เราบันทึก Software architecture ด้วยการอธิบาย
- เรานิยมเลือกวิธีการอธิบายด้วยแผนภาพและข้อความสั้น ๆ ประกอบกัน (Semi formal model)
- ในอีกนัยหนึ่ง Architecture Description = Architecture Modelling นั่นเอง

What if Software Architecture shown in source code

Source Code Represents Architecture?

→ ใช้ Source code เป็น model ได้ แต่ไม่นิยม เพราะ Detail เยอะ + ซับซ้อนมาก
→ แต่! ถ้าใช้ UML จะง่ายในการเข้าใจภาพรวม

- Software architecture descriptions are organized into Views.
- Each view addresses a set of system concerns
- Each view is drawn following the conventions of its viewpoint. ← ดูวิธีการเขียน view นั้น ๆ
- A viewpoint is a specification that describes the notations, modeling techniques to be used in a view to express the architecture.
- Viewpoints are generic templates, while views are what you actually see.

- Architectural Views → 4+1 view
- Architectural Viewpoints = แม่แบบวิธีการเขียน View

เช่น Functional viewpoint จะได้ Functional view

Architecture not so easy

- Unfortunately, this is not an easy task, and the risks and costs associated with selecting an inappropriate technology are high
- Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled

ADL ← เป็น Formal Model

- An architecture description language (ADL) is any means of expression used to describe a software architecture (ISO/IEC/IEEE 42010).
- Many special-purpose ADLs have been developed since the 1990s, including
  - AADL (SAE standard),
  - Wright (developed by Carnegie Mellon),
  - Acme (developed by Carnegie Mellon),
  - xADL (developed by UCI),
  - Darwin (developed by Imperial College London),
  - ArchC
- UML is an alternative ADL ← พยายามจะบอกว่าเกือบเป็น

สอบ

- Architecture addresses non-functional requirements ← เพราะ ทั้งสิ่งที่ระบบเป็นไปตาม NFR

Use case diagram บอก Non-functional req. ได้ไหม? Class diagram บอกได้ไหม? ← ไม่ได้

Non-functional มักจะมองไม่เห็นในการใช้งานแรก ๆ แต่ปรากฏเมื่อเวลาผ่านไป ในการบำรุงรักษา

Martin fowler เรียกมันว่า Internal quality

Quality Factors (Some NFR)

เช่น

- Scalability
- Security
- Performance
- Modifiability
- Availability
- Integration

เช่น High Available กับ Performance

* Quality Factors บางคู่ไม่เป็นเพื่อนกัน
* ทุกอย่างมี Cost ที่ไม่ใช่แค่ตอน Building แต่เป็นไปถึงตอน maintain ด้วย

สรุปส่วนสำคัญ

1. Architectural styles are high-level design
2. Architectural styles = Components + Connectors + Constraints (How components and connectors may be combined)

คำสำคัญ

- Software Architecture
- Model, Level of Abstraction
- Architectural Views
- Architectural Viewpoints
- Architectural Styles (Patterns)
- Architectural Representation/Documentation
- Architectural Decision Record (ADR)

สิ่งเหล่านี้กำหนดว่าเราจะต้อง model อย่างไร

- Constraints
- Risks
- Non-functional requirements
- เช่น นำระบบ Online shopping ที่เขียนได้มา และเพิ่มเงื่อนไขเหล่านี้
  - Scalability
  - Security
  - Cost constraints
  - Connecting to the Legacy systems
  - 247 Availability
  - …

Week 2

Intro to microservice arch

0. Intro (A brief refresher on software architecture)

– What is software architecture ⇒ "The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both."

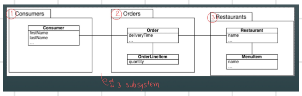
สำคัญ!

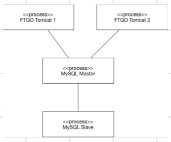
Architecture is multi-dimensional
e.g. Structural, electrical, plumbing, mechanical

⇒ Described by multiple views
View = (elements, relations, properties)

– 4+1 view model by Krutchen

4+1 is a view model used for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views". The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers, system engineers, and project managers.



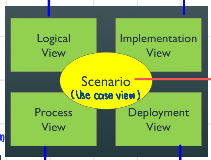


"Subsystem Diagram"

What developers create?
- **Elements: Classes and packages**
- **Relations: The relationships between them**

↳ เช่น Consumer มีความสัมพันธ์กับ Order

"Focus on system functionality as it pertains to end users"



ตอนที่ Run จริงๆ เช่น Activity Diagram

Running components

Elements: Processes

Relations: Inter-processcommunication

"Focus on run-time behavior (process)"

or development view

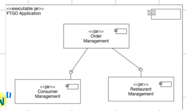
"Component Diagram"

What is produced by the build system?

Elements: Modules, (JAR files) and components (WAR files or executables)

Relations: Their dependencies

"Depict systems from programmer view"



or Physical view

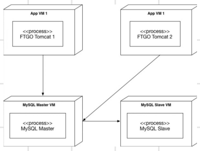
"Deployment Diagram"

Processes running on "machines"

Elements: Machines and processes

Relations: Networking

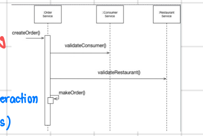
"Depict systems from system engineer view"



Use case diagram

Derived from use cases/stories

Animate the views



Scenario จะมีเรื่องของ Timing มาเกี่ยวข้อง

(Described sequence of interaction between objects and process)

described the functionality of the system from the perspective from outside world. (Black box perspective)

– Why architecture matters ?
- Enables an application to satisfy the second category of requirements: its quality-of-service requirements
- Non-functional requirements
- Also known as quality attributes and are the so-called –ilities

สิ่งที่เราต้องสนใจ

- Maintainability
- Testability
- Deployability

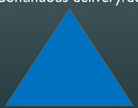
← Development velocity

Businesses must innovate faster ⇒ Build better software faster

- **Reducing** lead time (ลดระยะเวลาที่ Code ไปสู่ Production)
- **Increasing** deployment frequency

– Modern software development

Process: DevOps/Continuous delivery/deployment



Organization: Small, autonomous teams

Architecture: ??

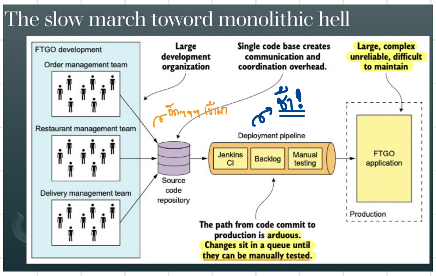
1. Symptoms of monolithic hell

The monolithic architecture is an architectural style that structures the application as a single executable component

Implementation View

	small monoliths	large monoliths
Maintainability	✓	😨
Testability	✓	😨
Deployability	✓	😨

App เราเริ่มโตขึ้นๆๆๆๆ จนในท้ายที่สุด การทำงานแบบ Agile และการ deployment กลายเป็นเรื่อง Impossible ⇒ "Monolithic hell"



Living in Monolithic hell →
1. การแก้บัค/ทำ feature ใหม่กลายเป็นเรื่องยาก + ทำนาน
2. ถ้างานใหญ่มากๆ ⇝ Slow down IDE และใช้เวลานานกว่าจะ Run ขึ้น
3. หมายถึงการ commit ไปสู่ deploy ลำบาก ⇒ Painful merges
4. Scaling กลายเป็นงานยาก
5. Delivering a reliable monolith is challenging due to lack of testability
6. Locked into increasing obsolete technology stack

2. How to escape monolithic by adopting the microservice architecture

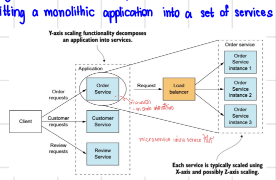
– ความหมายของ Microservice

The microservice architecture is an architectural style that structures an application as a set of loosely coupled, services organized around business capabilities

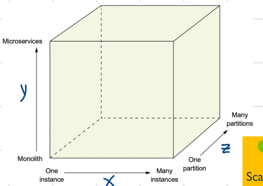
– Scale cube and microservice

Y-axis scaling, a.k.a. functional decomposition
Scale by splitting things that are different, such as by function.

"Scaling functionally decomposes an application into services"
– Splitting a monolithic application into a set of services

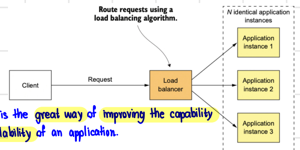


* A service can be scaled using X-axis scaling and, possibly, Z-axis scaling



X-axis scaling, a.k.a. horizontal duplication
Scale by cloning.

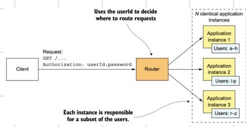
"Scaling load balances requests across multiple instances"
– Common way to scale a monolithic application
– Running multiple instances of the application behind a load balancer



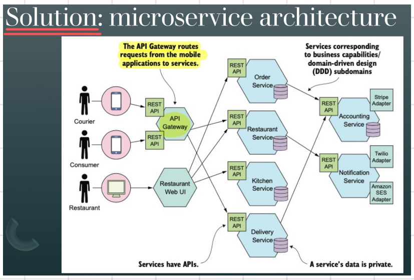
* X-axis is the great way of improving the capability and availability of an application.

Z-axis scaling, a.k.a. data partitioning
Scale by splitting similar things, such as by customer ID.

"Scaling routes requests based on an attribute of the request"
– Also runs multiple instances of the monolith application but each instance is responsible for only a subset of the data



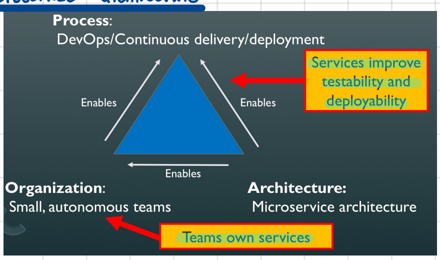
Service = Independently deployable component



③ The essential characteristics of the microservice architecture

-ilities of microservice architecture

- Maintainability
- Testability
- Deployability
- …



Microservices ⇒

Evolve the technology stack

- Pick a new technology when
  - Writing a new service
  - Make major changes to an existing service
- Let you experiment and fail safely

④ Benefits and drawbacks of microservice & How it enables DevOps

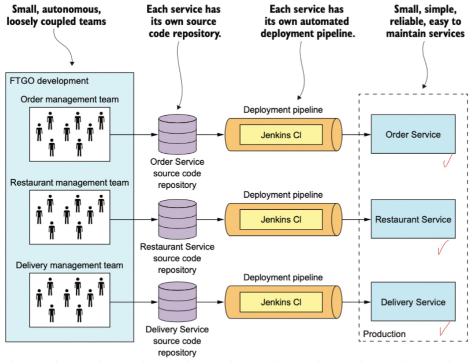
– Benefits : 1) Enable CI/CD (part of DevOps) of large, complex applications เพราะว่า microservice มี testability, deployability (required by CI/CD) และช่วยทำให้ทีมเป็น autonomous + loosely coupled

2) Services are small and easily maintained

3) Services are independently deployable (each service can be scaled independently)

4) It has better fault isolation

5) It allows easy experimenting and adoption of new technologies



Small, autonomous, loosely coupled teams
Each service has its own source code repository.
Each service has its own automated deployment pipeline.
Small, simple, reliable, easy to maintain services
FTGO development
Order management team
Restaurant management team
Delivery management team
Order Service source code repository
Restaurant Service source code repository
Delivery Service source code repository
Deployment pipeline
Jenkins CI
Order Service
Restaurant Service
Delivery Service
Production

– Drawbacks : 1) Finding the right set of service is challenging (ถ้าแบ่งผิด จะกลายเป็นแค่ distributed monolith)

2) Distributed systems are complex →

- Requires the use of unfamiliar techniques
  - Must use *sagas* to maintain data consistency across services
  - Must implement queries using either API composition or CQRS views

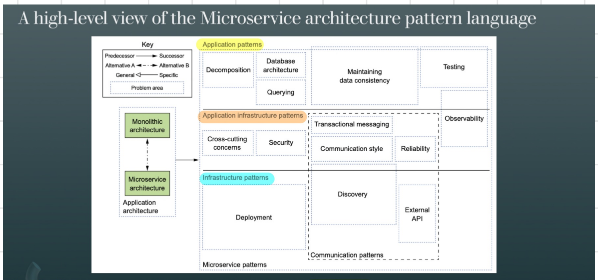
เช่น เวลาจะ join table ถ้า table อยู่คนละ service

3)
- Deploying features that span multiple services requires careful coordination
  - Need to create a rollout plan that orders service deployments based on the dependencies between services

4)
- Deciding when to adopt the microservice architecture is difficult
  - A startup should begin with a monolithic application
  - When the problem is how to handle complexity
    → It makes sense to decompose the application into microservices

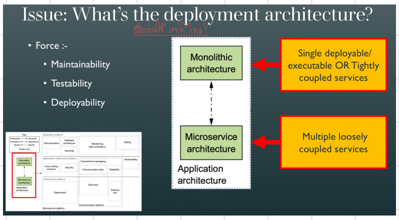
⑤ The microservice architecture pattern language and why you should use it

- A pattern is a reusable solution to a problem that occurs in a particular context.
- The pattern language guides you when developing an architecture.
- What architectural decisions you must make for each decision:
  - Available options
  - Trade-offs of each option

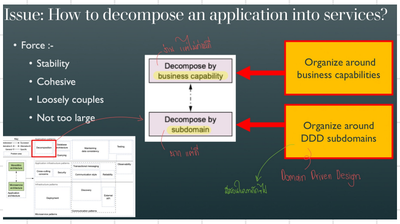


A high-level view of the Microservice architecture pattern language
Key
Predecessor → Successor
Alternative A ↔ Alternative B
General ◁ Specific
Problem area
Application patterns
Decomposition
Database architecture
Querying
Maintaining data consistency
Testing
Monolithic architecture
Microservice architecture
Application architecture
Application infrastructure patterns
Cross-cutting concerns
Security
Transactional messaging
Communication style
Reliability
Observability
Infrastructure patterns
Deployment
Discovery
External API
Communication patterns
Microservice patterns

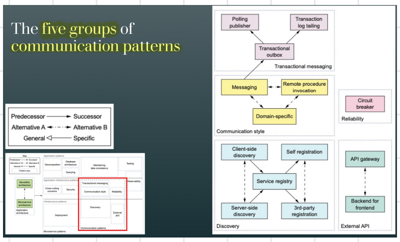
Application patterns —These solve problems faced by developers.
Application infrastructure —These are for infrastructure issues that also impact development.
Infrastructure patterns —These solve problems that are mostly infrastructure issues outside of development.



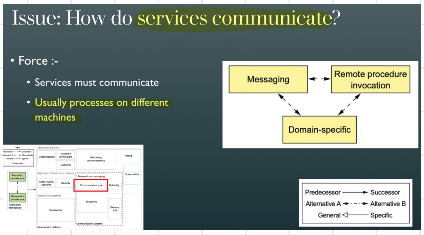
Issue: What's the deployment architecture?
เลือกใช้ arch ไหน?
Force :-
Maintainability
Testability
Deployability
Monolithic architecture
Microservice architecture
Application architecture
Single deployable/ executable OR Tightly coupled services
Multiple loosely coupled services

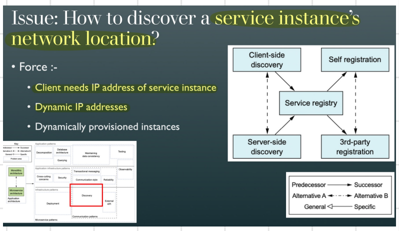


Issue: How to decompose an application into services?
Force :-
Stability
Cohesive
Loosely couples
Not too large
Decompose by business capability
Decompose by subdomain
Organize around business capabilities
Organize around DDD subdomains
Domain Driven Design

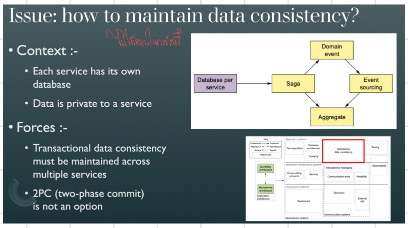


The five groups of communication patterns
Predecessor → Successor
Alternative A ↔ Alternative B
General ◁ Specific
Polling publisher
Transaction log tailing
Transactional outbox
Transactional messaging
Messaging
Remote procedure invocation
Domain-specific
Communication style
Circuit breaker
Reliability
Client-side discovery
Self registration
Service registry
Server-side discovery
3rd-party registration
Discovery
API gateway
Backend for frontend
External API





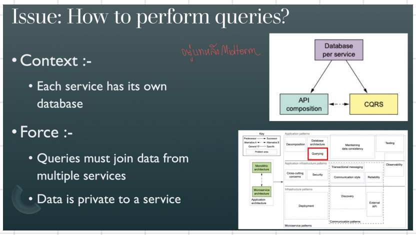
Issue: How to discover a service instance's network location?
Force :-
Client needs IP address of service instance
Dynamic IP addresses
Dynamically provisioned instances
Client-side discovery
Self registration
Service registry
Server-side discovery
3rd-party registration
Predecessor → Successor
Alternative A ↔ Alternative B
General ◁ Specific

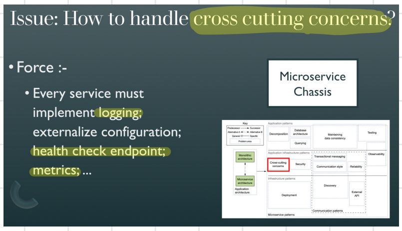


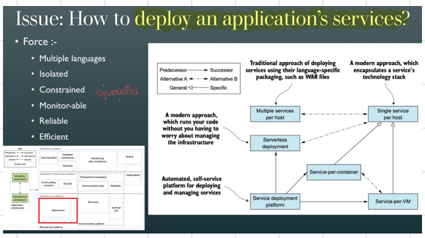
Issue: how to maintain data consistency?
Context :-
Each service has its own database
Data is private to a service
Forces :-
Transactional data consistency must be maintained across multiple services
2PC (two-phase commit) is not an option
Domain event
Database per service
Saga
Event sourcing
Aggregate

Summary

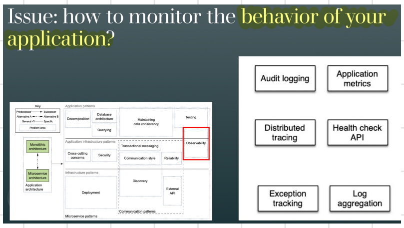
- The goal of architecture is to satisfy non-functional requirements
- For continuous delivery/deployment use the appropriate architectural style
  - Small applications ⇒ Monolithic architecture
  - Complex applications ⇒ Microservice architecture
- Use the pattern language to guide your decision making







Issue: How to deploy an application's services?
Force :-
Multiple languages
Isolated
Constrained
Monitor-able
Reliable
Efficient
Predecessor → Successor
Alternative A ↔ Alternative B
General ◁ Specific
Traditional approach of deploying services using their language-specific packaging, such as WAR files
A modern approach, which encapsulates a service's technology stack
Multiple services per host
Single service per host
A modern approach, which runs your code without you having to worry about managing the infrastructure
Serverless deployment
Service-per-container
Automated, self-service platform for deploying and managing services
Service deployment platform
Service-per-VM



Architectural Decision Record (ADR)

ADR (Architectural Design Record)

- **AD (Architectural decision)** — Architectural design choice that addresses **significant requirements**
- **Context** — Describes the force at play, e.g., technological, political, social and project local, team expertise, business situations, priority
- **Consequence** — Describe what needs to be done

An architecture decision record (ADR) : a document that captures an important **architectural decision (AD)** made along with its **context** and **consequence**.

An architecture decision (AD) is a software **design choice** that addresses a significant requirement.

An architectural decision log (ADL) : **collection of ADRs** created and maintained for a particular project or organization.

An architecturally-significant requirement (ASR) : a requirement that has a measurable effect on a software system's architecture.

Architecture knowledge management (AKM) : the included topics of all above.

How to start using ADRs ?

1. Decision Identification → เร่งด่วน ? / สำคัญ ? / ยังไง
2. Decision Making → เช่น การทำ Dialog Mapping / Group decision making
3. Decision Enactment and enforcement → ต้องมีการคุยแลกเปลี่ยนกับ Stakeholders ของ system / code review เกี่ยวกับ architecture
4. Decision Sharing (optional) ⇝ Many ADs recur across project
5. Decision Documentation → เช่น M. Nygard's ADRs.

How to start ADRs with tools

- Google drive; google doc, google sheet, w/ online editing among team members
- Version control tools, e.g., git
- Project planning tools, e.g., Atlassian Jira w/ tool planning tracker
- wikis, e.g., MediaWiki

What else?

How to write a good ADR ?

Characteristics of a good ADR

- **Point in Time** – Identify **when** the AD was made
- **Rationality** – Explain **why** when making the particular AD
- **Immutable record** – The previously published AD should not be altered
- **Specificity** – Each ADR should be about a **single AD**

Characteristics of a good "Context" section in ADR

- Explain your organization's **situation** and business **priorities**
- Include rationale and considerations based on social and skill makeups of your teams

บอกสถานการณ์ปัจจุบัน ลำดับความสำคัญของงาน รวมทั้งบริบทแวดล้อม เช่น ทีมที่มีอยู่มีทีมอะไรบ้าง มีทักษะอะไรบ้าง

Characteristics of a good "Consequences" section in ADR

- ✓ Right approach – "We need to start doing X instead of Y"
- ✗ Wrong approach – Do not explain the AD in terms of "Pros" and "Cons"

อย่าบอกแค่ข้อดี ข้อเสีย แต่ต้องบอกว่าตัดสินใจว่าเลือกอะไร

Decision record template by Jeff Tyree and Art Akerman

- **Issue :** Why do we address this?
- **Decision :** The architecture's direction; the position we selected.
- **Status :** The decision's status; pending, decided, approved
- **Group :** Assigned category for the decision; help organize the set of decisions
- **Assumptions :** Assumptions when the decision is made; schedule, cost, tech,...
- **Constraints :** Capture any additional constraints that the decision might pose.
- **Positions :** Other options / alternatives; to avoid question "Did you think about..?"
- **Argument :** Outlines why you selected a position; implementation cost, total ownership cost, time to market, and required development resources' availability
- **Implication :** A decision may come with many implications; creating new requirements; modifying existing requirements, pose additional constraints, require negotiating scope or schedule with customer, require additional staff training

Related = สิ่งอื่นๆ เกี่ยวกับการตัดสินใจนี้

- **Related decisions :** List the related decisions here; traceability matrix, decision trees, metamodels
- **Related requirements :** Decisions should be business-driven. To show accountability, map decisions to the objectives or requirements. Assess each architecture decision's contribution to meet each requirement, then, assess how well if a requirement is met across all decisions.
- **Related artifacts :** List the related architecture, design or scope documents that this decision impacts.
- **Related principles :** Make sure the decisions are aligned with the enterprise's principles if available.
- **Notes :** Capture notes during the team discussions and the socialization process.

14 หัวข้อ!!!

แต่ที่เราจะเขียนกันจะเป็น ver. minimal

Decision record template by Michael Nygard

Title

Context
Describes the forces at play, including technological, political, social, and project local. It is simply describing facts.

Decision
Describes our responses to these forces. State it in full sentences, with active voices. "We will..."

Status
proposed : if the project stakeholders haven't agreed with it yet.
accepted: if it is agreed.
deprecated or superseded : if it is replaced by a new ADR.

Consequences
describes the resulting context, after applying the decision. All consequences: positives

1-2 pages long; write an ADR as if it is a conversation with a future developer.

Application ที่ดี ควรมีคุณสมบัติ

The Twelve-Factor App

The 12-Factor App Framework was created in 2012 by developers at early cloud pioneer Heroku as a set of rules and guidelines for organizations building modern web applications that run "as a service." The framework was inspired by Martin Fowler's work on application architecture and what he saw as suboptimal application development processes.

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes

Telemetry data refers to the logs, metrics and traces in observability — what is sometimes called "the three pillars of observability". Logs are text records of events that include a timestamp (when the event happened) and a payload that offers context.

Then, what is the Fifteen-Factor App??

Telemetry focuses on real-time performance metrics, offering insights into a system's current health and behavior, aiding in proactive issue detection. Logging, on the other hand, records specific events and transactions, creating a historical record for post-incident analysis and debugging.

Logging ต่างจาก Telemetry อย่างไร

XIII. API First
XIV. 14 Telemetry
XV. Authentication and Authorization
by Hoffman

https://github.com/ciuddi/15-factor-app-workshop

Week 4

Summary

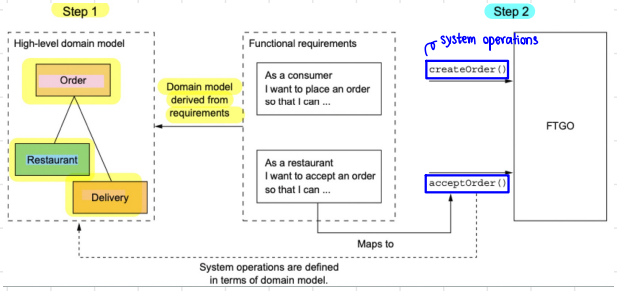
- Services in a microservice architecture are organized around business concerns—business capabilities or subdomains—rather than technical concerns.
- There are two patterns for decomposition:
  - Decompose by business capability, which has its origins in business architecture
  - Decompose by subdomain, based on concepts from domain-driven design
- You can eliminate god classes by applying DDD and defining a separate domain model for each service.

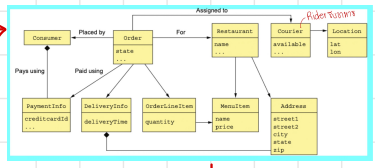
Decomposition Strategies

Defining an application's microservice architecture

Three-step process

- **Step 0**: เตรียม Requirements
- **Step 1**: Identify system operations
  - = an abstraction of a request that the application must handle (a.k.a. method)
  - illustrate how the services collaborate.
  - represents external requests
  - **Step 1**: Create a high-level domain model.
  - **Step 2**: Define the system operations in terms of the domain model





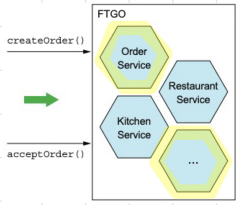
Actor	Story	Command	Description
Consumer	Creates Order	*createsOrder()*	Creates an order
Restaurant	Accept Order	*acceptOrder()*	Indicates that the restaurant has accepted the order and is committed to preparing it by the indicated time
Restaurant	Order Ready for Pickup	*noteOrderReadyForPickup()*	Indicates that the order is ready for pickup
Courier	Update Location	*noteUpdatedLocation()*	Updates the current location of the courier
Courier	Delivery picked up	*noteDeliveryPickedUp()*	Indicates that the courier has picked up the order
Courier	Delivery delivered	*noteDeliveryDelivered()*	Indicates that the courier has delivered the order

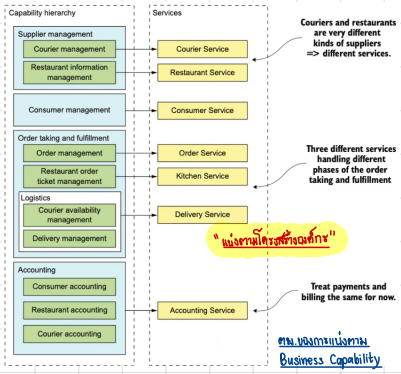
- **Step 2**: Identify services ⇒ แยก service อย่างไร

The end result is services that are organized around business concepts rather than technical concepts

มี 2 วิธี คือ 1) Business capabilities

Business capability = something that a business does to generate value
↳ Identified by analyzing the organization's purpose, structure, and business process





"แบ่งตามโครงสร้างองค์กร"

ตาม business capability

เพราะ business แต่ละ application มีหน้าที่ไม่ค่อยเปลี่ยน

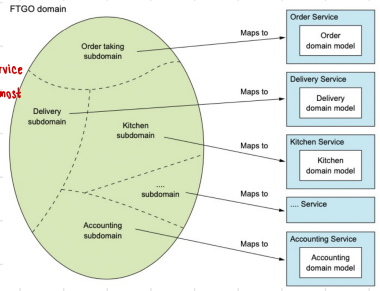
ข้อดี: - The resulting architecture are relatively stable.

ข้อเสีย: - เกิดความไร้ประสิทธิภาพ จาก interprocess communication
- นาง service อาจใหญ่ๆ ซับซ้อน

2) Domain-driven design subdomains (90% ดีกว่า)

- Domain-driven design (DDD) = An approach for building complex software applications that is centered on the development of an object-oriented domain model
  ↳ = is mirrored in the design and implementation of application

- มี 2 concepts ให้ใช้ (1) subdomains → แยกออกเป็น subdomain เล็กๆให้ โดยผู้เชี่ยวชาญศึกษาหน้าที่อันนี้นะ
  (2) bounded context → scope ของ domain model



** DDD and the microservice architecture are in almost perfect alignment.

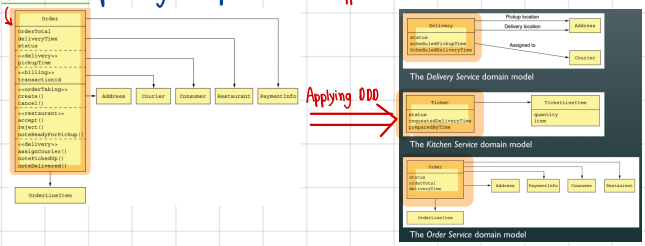
Patient Management, Appointment Scheduling, Billing and Insurance

"Identify service by business capability" focuses on real-world business activities and tasks, while "Identify servic e by DDD subdomains" emphasizes DDD's bounded contexts and subdomains.

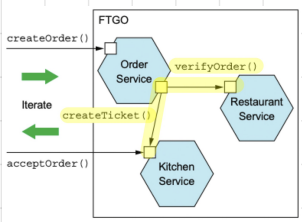
Patient Context, Billing Context

Obstacles to decomposing an application into services

1. Network latency — แก้: Implementing a batch API
2. Reduced availability due to synchronous communication — แก้: Using asynchronous messaging
3. Hard to maintain data consistency
4. Inability to obtain a consistent view of data across multiple databases
5. God classes preventing decomposition (คลาสหนวม ที่แทรกมา) → Most applications have at least one of these classes



- **Step 3**: Define service APIs and collaborations



1) Map each system operation to a service. → To decide which service is the initial entry point for a request →

Service	Operations
Consumer Service	createConsumer()
Order Service	createOrder()
Restaurant Service	findAvailableRestaurants()
Kitchen Service	* acceptOrder() * noteOrderReadyForPickup()
Delivery Service	* noteUpdatedLocation() * noteDeliveryPickedUp() * noteDeliveryDelivered()

2) Then, decide whether a service needs to collaborate with others to implement a system operation.

3) If collaboration is required, then determine what APIs those other services must provide in order to support collaboration

Determine the APIs to support collaboration between service

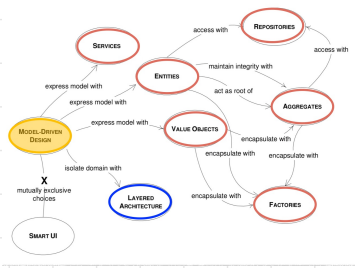
Service	Operations	Collaborators
Consumer Service	verifyConsumerDetails()	—
Order Service	createOrder()	* Consumer Service verifyConsumerDetails() * Restaurant Service verifyOrderDetails() * Kitchen Service createTicket() * Accounting Service authorizeCard()
Restaurant Service	* findAvailableRestaurants() * verifyOrderDetails()	—
Kitchen Service	* createTicket() * acceptOrder() * noteOrderReadyForPickup()	* Delivery Service scheduleDelivery()
Delivery Service	* scheduleDelivery() * noteUpdatedLocation() * noteDeliveryPickedUp() * noteDeliveryDelivered()	—
Accounting Service	* authorizeCard()	—

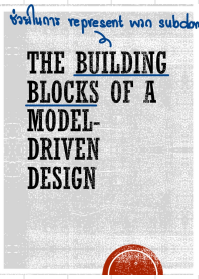
- The data might be scattered around multiple services
  - For example, in order to implement the *createOrder()* operation, the Order Service must invoke some other services

Domain Driven Design (DDD)

- DDD focus on modelling the software that match the business's core domain.
- Software needs to incorporate the core concepts and elements of the domain, and to precisely realize the relationship between them.
- DDD emphasizes collaboration between domain experts and software developer.

- **Ubiquitous Language** : A common language between developers and domain experts (ภาษาที่ทุกคนในทีมเข้าใจตรงกัน)
- **Model-Driven Design (MDD)** : emphasizes the use of models to describe various aspects of a software system, including its structure, behavior, and interactions.

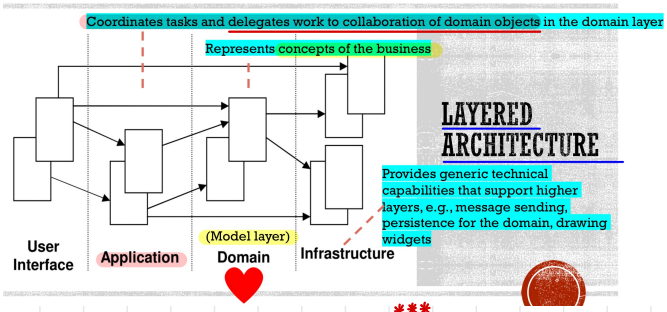




ช่วยในการ represent ของ subdomain

- MDD is an approach to software development where extensive models are created before source code is written.
- MDD ช่วยในการสร้าง model ที่แสดงถึง aspects ต่างๆ ของ software ซึ่งรวมไปถึง domain ด้วย
  - incorporate concepts from DDD

- **Layered Architecture** : A common architectural solution for DDD contain 4 conceptual layers



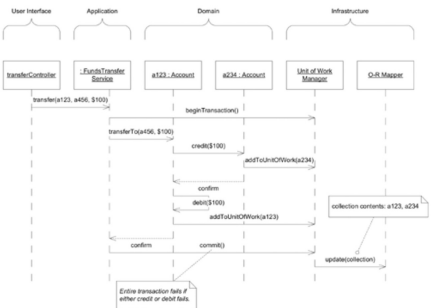
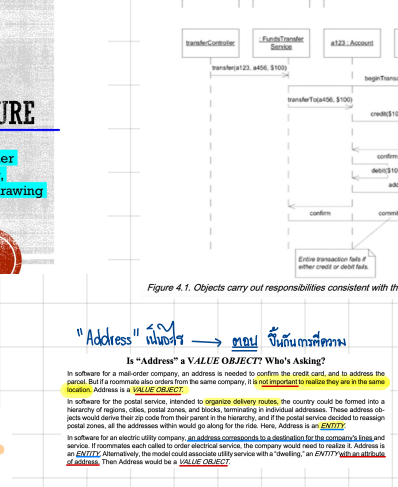


Figure 4.1. Objects carry out responsibilities consistent with their layer and are more coupled to other objects in their layer.

- **Entities** :
  - A category of objects which seem to have an identity which remains the same throughout the states of the software.
  - For these objects it is not attributes which matters but a thread of continuity and identity, which spans the life of a system.

* Entities represents persistent information tracked by the system.

"Address" เป็นได้ทั้ง → ตอบ ขึ้นกับการตีความ



- **Value Objects** :
  - An object used to describe certain aspects of a domain, and which does not have identity.
  - It is recommended to select as entities only those objects conform to the entity definition and make the rest of the objects to be Value Objects.
  - It is highly recommended that value objects are immutable.
  - Examples: Air travel booking system create object for each flight, the flight code can be shared among the customers, who book the same flight.

เพื่อน attribute แทนเป็น object

- **Aggregates** :
  - A group of associated objects which are considered as one unit regarding data changes.
  - The Aggregate is demarcated by a boundary which separates the objects inside from outside.
  - Each Aggregate has one root, an Entity, the only accessible object from outside.

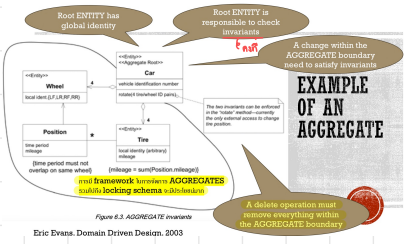


Figure 6.3. AGGREGATE invariants

Eric Evans. Domain Driven Design. 2003

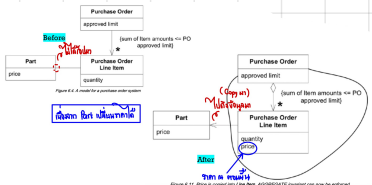


Figure 6.11. Prior to repeal the Line Item, AGGREGATE invariant can now be enforced.

- **Services** :
  - Service is used to represent behavior within a domain.
  - It can group related functionalities which serves the Entities and Value objects.
  - It is much better to declare service explicitly.
  - Services act as interfaces which provide operations, which should be stateless.
  - Services provide a point of connection for many objects.

- **Factories** :
  - Factories are used to encapsulate the knowledge necessary for object creation, and they are especially useful to create Aggregates.
  - Factories is responsible to create other objects.
  - For immutable Value Objects, all attributes are initialized to their valid state.

Encapsulate the creation and initialization of complex objects or aggregates.

- **Repositories** : ตัวที่เหมาะ database ที่เหมาะ ช่วย encapsulate logic ต่างๆ
  - They present clients with a simple model for obtaining persistent objects and maintaining their life cycle.
  - They decouple application and domain design from persistence technology, multiple database strategies, or even multiple data sources.
  - They communicate design decisions about object access.
  - They allow easy substitutions of a dummy implementation, for use in testing.

เพื่อที่

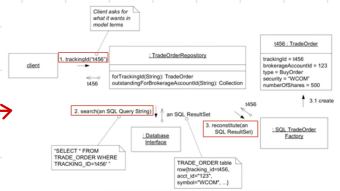
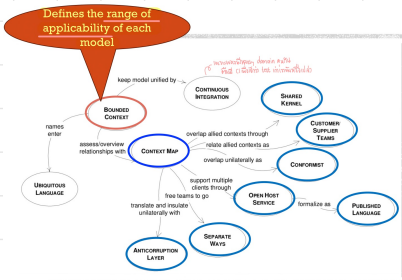


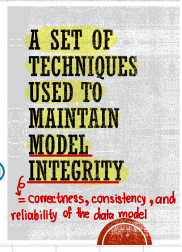
Figure 6.21. The REPOSITORY encapsulates the underlying data store.

- **Domain Events** :
  - A domain event is representation of something that happened in the domain.
  - Something happened that domain experts care about.
  - Each event is represented as a domain object.

เช่น ของ Diagram นี้แสดงเห็นถึง domain event

- **Preserving model integrity**

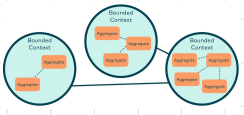




Boundary ของ domain

- **Bounded Context** :
  - A Bounded Context defines the range of applicability of each model.
  - A Bounded Context provides the logical frame inside of which the model evolves.
- **Continuous Integration** :
  - All work within the context is being merged and made consistent frequently enough that t when splinters happen they are caught and corrected quickly.
    - Continuous Integration separates at two levels:
      - the integration of model concepts
      - the integration of the implementation.

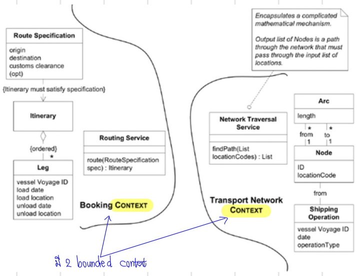
- Aggregates represent *business entities*, and thus are smaller in scope than Bounded Contexts, which represent entire *domains*
- Aggregates encapsulate data, whereas Bounded Contexts encapsulate services, team members, and their shared language(s)



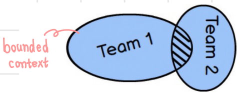
● **Context Map** = Document which outlines the different Bounded Context and relationships between them.
(→ รูปแบบการพูดคุยระหว่าง Bounded Contexts)

Code reuse between BOUNDED CONTEXTS is a hazard to be avoided.

เพราะ 1. แต่ละ Bounded Context อาจจะมี business logic และ rule ที่แตกต่างกัน ดังนั้น code ที่เขาเขียนอาจไม่ work กับ context ของเราก็ได้
2. ตามนิยาม แต่ละBounded Context จะมีความเป็นอิสระจากกัน ซึ่งการทำ code reuse จะทำให้เกิด coupling ระหว่าง context (ถ้าอันนี้ อีกอันต้องแก้ด้วย)

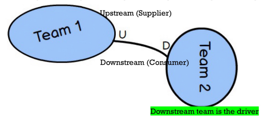


1. Shared Kernel : 2 or more bounded context can share a common model (ex. shared libs, shared services)



The teams must agree on what model elements they are to share.

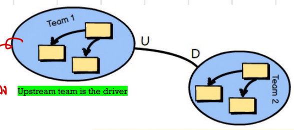
2. Customer-Supplier : supplier ส่งข้อมูลให้ Consumer ตามที่ Consumer ต้องการ



The *Supplier* must provide what the *Customer* needs.

3. Conformist : One bounded context makes a decision to conform to another bouned context's model, rule, or standards.
(downstream ต้องทำตามทุกอย่างที่upstreamให้มา; v.ทำตาม)

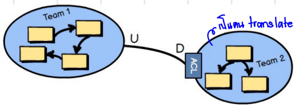
ถ้าสิ่งที่ได้จะมีรูปแบบที่ชัดเจน ซึ่งผู้รับต้องทำตามให้ได้เอง



For various reasons the downstream team cannot sustain an effort to translate the Ubiquitous Language of the upstream model to fit its specific needs, so the team conforms to the upstream model as is.

(ขยายต่อ)

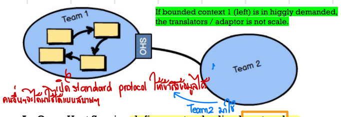
4. Anticorruption layer (ACL) : เพิ่ม ACL ตรงฝั่ง downstream เพื่อป้องกันอันตรายจาก upstream



An Anticorruption Layer is the most defensive Context Mapping relationship. The layer isolates the downstream model from the upstream model and translates between the two.

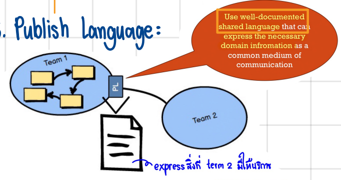
5. Open Host Service : supplier มี OHS consumer ต้องส่ง request มาขอข้อมูล

(part which exposes as a service that other contexts can use)
(ต้องอยู่ในรูปแบบที่กำหนด)



An Open Host Service defines a standardized protocol or interface that gives access to your Bounded Context as a set of services.

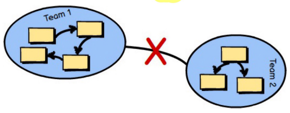
6. Publish Language :



Such a Published Language can be defined with XML Schema, JSON Schema, or a more optimal wire format, such as Protobuf or Avro.

7. Separate Ways : ต่างคนต่างทำงานของตัวเอง ไม่คุยกัน

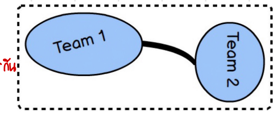
Allowing developers to find simple, specialized solutions within this small scope.



In this case produce your own specialized solution in your Bounded Context and forget integrating for this special case.

8. Partnership : ≥ 2 bounded context have a strategic collaboration.
(เช่น share goal ร่วมกัน)

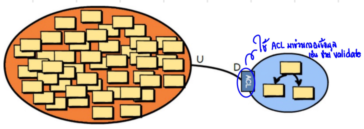
ถ้าอันใดอันหนึ่งพัง → พังทั้งกลุ่มที่ Partner กัน



Each team is responsible for one Bounded Context.

When teams in two contexts will succeed or fail together, a cooperative relationship often emerges.

9. Big Ball Of Mud : ก้อน context ที่ใหญ่มากๆๆ



Avoid to create big ball of mud, and if need to use it, try ACL.

TAKE HOME MESSAGE

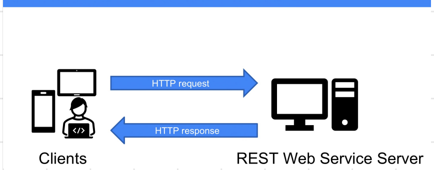
- "DDD is less about software design patterns and more about problem solving through collaboration.
- Evans presents techniques to use software design patterns to enable models created by the development team and business experts to be implemented using the UL.
- However, without the practices of analysis, and collaboration, the coding implementation really means very little on its own
- DDD is not code centric; its purpose is not to make elegant code. Software is merely an artifact of DDD.
- Strategic design focuses on defining the bounded contexts, ubuquitous languages, and context maps
- Tactical design is a set of technical resources used in constructing domain model in a bounded context.

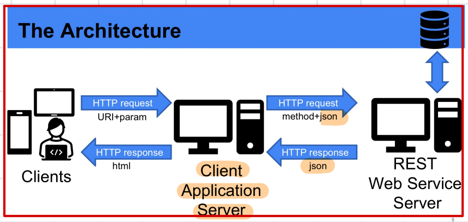
Week 5 & 6

REST API

Architecture

The Client-Server Architecture





HTTP Request Methods

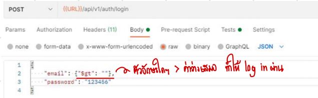
GET	Retrieve Resource
POST	Submit Resource
PUT/PATCH	Update Resource
DELETE	Delete/Destroy Resource

Status Code

- Informational responses (100–199)
- Successful responses (200–299)
  - 200 Success
  - 201 Created
  - 204 No Content
- Redirects (300–399)
  - 304 Not Modified
- Client errors (400–499)
  - 400 Bad Request
  - 401 Unauthorized
  - 404 Not Found
- Server errors (500–599)
  - 500 Internal Server Error

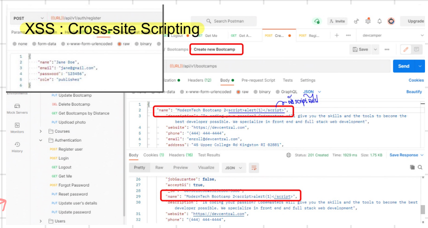
API Security

1. Logout to clear token cookie
2. Prevent NoSQL Injection & Sanitize Data → เครื่องมือ เช่น express-mongo-sanitize



3. Security Headers → เครื่องมือ เช่น helmet

XSS : Cross-site Scripting



4. XSS Protection → เครื่องมือ เช่น xss-clean
5. Rate Limiting & HPP → ป้องกันไม่ให้เรียก API มากเกินไป

HPP: prevent HTTP parameter polution attack
http://www.aaa.com/search?p=cat&p=dog&p=pig&...

เครื่องมือ เช่น express-rate-limit-hpp

6. CORS : Cross-Origin Resource Sharing → เครื่องมือ เช่น cors
(อนุญาตเฉพาะที่มาจากที่ที่ระบุไว้เท่านั้น)

Creating OpenAPI Document with Swagger

OpenAPI = A specification language for HTTP APIs that defines structure and syntax

- A **standard, language-agnostic** interface to **REST APIs** which allows both **humans** and **computers** to **discover** and **understand** the capabilities of the service **without access** to source code documentation or through network traffic inspection.

OpenAPI Specification

- Allow machines/tools to integrate with API
- Allow humans to implement API code
- Allow humans to read and generate API documentation and test case

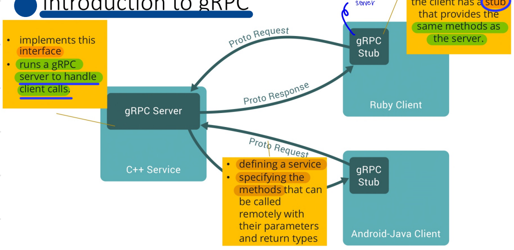
- OpenAPI = Specification
- Swagger = tools

gRPC

= Framework ที่ใช้สื่อสารกันของ distributed system — Support to many languages!

- **gRPC R**emote **P**rocedure **C**alls
- gRPC is a modern, open source remote procedure call (RPC) framework that can run anywhere. It enables client and server applications to communicate transparently, and makes it easier to build connected systems.
- It has been used by Google, Square, Netflix, CoreOS, Docker, CockroachDB, Cisco, Juniper Networks and many other organizations

Introduction to gRPC



4 kinds of gRPC Service Method

#	Kind	Client's Request	Server's Response
1	Unary RPCs	Single	Single
2	Server streaming RPCs	Single	Stream
3	Client streaming RPCs	Stream	Single
4	Bidirectional streaming RPCs	Stream	Stream

*gRPC guarantee message order

Protocol Buffer (.proto file)

: Interface Definition Language (IDL) for describing both the service interface and the structure of the payload messages

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

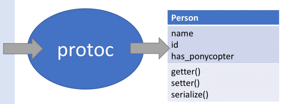
- required [1]
- optional [0..1]
- repeated [0..*]

unique number
- From 1 to 2^{29} - 1, or 536,870,911
- Except 19000 through 19999

- double, float, Bool, String
- bytes, int32/int64, uint32/uint64, sint32/sint64

ex.

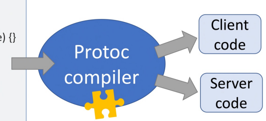
```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```




```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloResponse) {}
}

// The request message containing the user's name.
message HelloRequest {
  string greeting = 1;
}

// The response message containing the greetings
message HelloResponse {
  string reply = 1;
}
```



	Unary	Server Streaming	Client Streaming	Bidirectional Streming
Server file	- รับ 2 parameter คือ call กับ callback	- รับ parameter แค่ตัวเดียว คือ call - มี call.write (ข้อมูลที่จะส่ง) ตามด้วย call.end()	- รับ 2 parameter คือ call กับ callback - มี call.on('data', function(data){...}) - มี call.on('end', function(){...})	- รับ parameter แค่ตัวเดียว คือ call - มี call.on('data', function(data){...; call.write(data) }); - มี call.on('end', function(data){call.write(data) });
Client file	- รับ 1 parameter คือ callback	- รับ 1 parameter คือ callback - มี call.on('data', function(data){...}) call.on('end', function(){...}) call.on('error', function(e){...}) call.on('status', function(status){...})	- รับ 1 parameter คือ callback - มี call.write (ข้อมูลที่จะส่ง) ตามด้วย call.end()	- รับ 1 parameter คือ callback - มี call.on('ประเภท', function), call.write(data), call.end()

Week 7

Interprocess communication (IPC)

Overview

- IPC
  - Synchronous : request/respond-based communication mechanism (เช่น REST, gRPC) — HTTP based
  - Asynchronous : message-based communication mechanism (เช่น Advanced Message Queuing Protocol (AMQP), STOMP)
- Different messages formats
  - Human-readable : text-based formats (เช่น JSON, XML)
  - Binary format : เช่น Avro, Protocol Buffers
- Interaction styles
  - First Dimension
    - One-to-one → Each client request is processed by exactly one service.
    - One-to-many → Each request is processed by multiple services.
  - Second Dimension
    - Synchronous → The client expects a timely response from service and might even block while it waits.
    - Asynchronous → The client doesn't block, and the response, if any, isn't necessarily sent immediately.

	one-to-one	one-to-many
Synchronous	Request/response	—
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/async responses

1. One-to-one interactions
   - Request/response : waits for response + might event block while waiting
   - Asynchronous request/response : NOT block while waiting
   - One-way notifications : A service client sends a request to a service, but no reply is expected or sent.
2. One-to-many interactions
   - Publish/Subscribe : A client publishes a notification message + consumed by ≥0 interested service
   - Publish/async responses : A client publishes a request message + Waits for a certain amount of time for responses from interested services.

- Defining APIs in microservice
  - A well-designed interface exposes useful functionality while hiding the implementation
  - Enables the implementation to change without impacting clients
  - Define a service's API using Interface Definition Language (IDL)
    - เช่น IDL ของ REST ที่นิยมมากที่สุด คือ OpenAPI Specification (REST did not originally have an IDL)
    - ส่วน gRPC ใช้ Protocol Buffers-based IDL

Communicating using the Synchronous Remote procedure invocation pattern (RPI)

- Using a remote procedure invocation-based IPC mechanism : client ส่ง request → service ส่ง response กลับมา + client ควร block เพื่อรอ response + Client ควรมีข้อ response จะ "arrive in a timely fashion"
- REST vs. gRPC

	REST	gRPC
Benefits	– simple & familiar – Directly supports request/response style communication – Firewall friendly – Doesn't require an intermediate broker (∴ Simplifies the system's architecture)	– Straightforward to design an API with a rich set of update operations – Efficient especially when exchanging large messages – Bidirectional Streming enables both RPI and messaging styles of communication – Enables interoperability between clients and services written in a wide range of languages. (ภาษาที่งานต่างกัน)

Drawback	– Only supports request/response style communication
	– Reduced availability
	  – Client and service communicate directly without an intermediary to buffer messages
	  – They must both be running for the duration of the exchange
	– Client must know the location (URLs) of the service instance(s) (→ Using "service discovery" mechanism)
	– Fetching multiple resources in a single request is challenging
		– More work for javascript clients to consume gRPC-based API
		– Older firewalls might not support HTTP/2

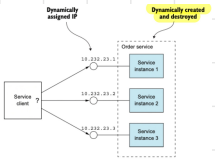
- Handing partial failure (1,2,3 ที่2 คือ = partial failure)
  - * กรณี synchronous request → มีแนวโน้มเกิด partial failure

① Using the Circuit breaker pattern

มี 2 solution ใช้กัน

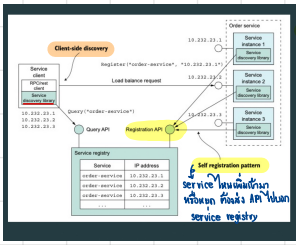
1. Design RPI proxies to handle unresponsive remote services.
   1. Network timeouts
      - Never block indefinitely and always use timeouts when waiting for a response.
   2. Limiting the number of outstanding requests from a client to a service (เช่น กำหนดไว้ไม่เกิน 5 คำขอพร้อมกัน)
      - Impose an upper bound on the number of outstanding requests that a client can make to a particular service.
   3. Circuit breaker pattern
      - Track the number of successful and failed requests
      - If the error rate exceeds some threshold, trip the circuit breaker so that further attempts fail immediately.
2. Design how to recover from a failed remote service
   - We must also decide how your services should recover from an unresponsive remote service
   - Option 1 - Return an error to its client
   - Option 2 - Returning a fallback value

② using Service Discovery



- สมัยก่อน → Network location ของ service instance มักเป็นแบบ static
- สมัยนี้ → เพิ่มขึ้นเป็น cloud-based ที่ที่มี Network location เป็นแบบ dynamic
-

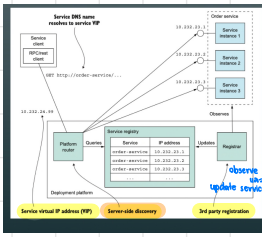
- Service discovery key component is a service registry → เก็บว่า service นี้มี ip address อะไร
  - A database of the network locations of an application's service instances.
- The service discovery mechanism updates the service registry when service instances start and stop
- There are two main ways to implement service discovery:
  1. The services and their clients interact directly with the service registry.
  2. The deployment infrastructure handles service discovery.



Option 1 – Apply the application-level service discovery pattern

- Implement service discovery for the application's services and their clients
- This approach combine two patterns
  - Self registration pattern
  - Client-side discovery pattern

Client เป็นคน discover service



Option 2 – Apply the platform-provided service discovery pattern

- Use a built-in service registry of modern deployment platforms
  - e.g. Docker and Kubernetes
- This approach combine two patterns
  - 3rd party registration pattern
  - Server-side discovery pattern

observe และ update service registry

ตรงนี้ทำให้ network addr ของ service ไม่ขึ้นกับ clients

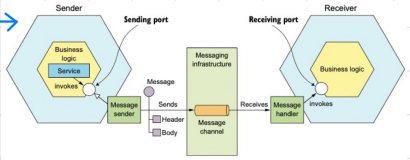
The client doesn't block waiting for a reply + assuming that the reply won't be received immediately.

Communicating using the Asynchronous messaging pattern

Overview

- Messages are exchanged over message channels.
  - A sender (an application or service) writes a message to a channel.
  - A receiver (an application or service) reads messages from a channel.
- There are several different kinds of messages:
  - Document—A generic message that contains only data.
  - Command—A message that's the equivalent of an RPC request.
  - Event—A message indicating that something notable has occurred in the sender.

1. The business logic in the sender invokes a *sending port* interface.
2. The *message sender* sends a message to a receiver via a *message channel*.
3. The *message channel* is an abstraction of messaging infrastructure.
4. A *message handler* adapter in the receiver is invoked to handle the message.
5. It invokes the *receiving port* interface implemented by the receiver's business logic.



- ตัว Channel มี 2 ประเภท ได้แก่ 1. A point-to-point channel : ส่ง message ให้แค่ 1 consumer (เช่น command message)
  2. A publish-subscribe channel : ส่งให้หลาย consumer (เช่น event message)

Implementing the interaction style using messaging

1. Request/response and Asynchronous Request/response

- Messaging is inherently asynchronous, so only provides asynchronous request/response.
- But a client could block until a reply is received. → Synchronous

2. One-way notification

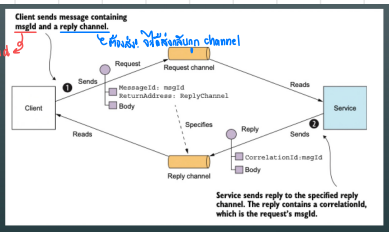
- The service subscribes to the channel and processes the message.
- But it doesn't send back a reply.

3. Publish/subscribe

- A client publishes a message to a publish-subscribe channel that is read by multiple consumers
- A service that's interested in a particular domain object's events only has to subscribe to the appropriate channel.

4. Publish/Asynchronous subscribe

- Combining elements of publish/subscribe and request/response



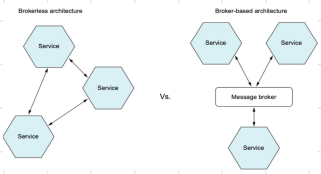
1. Implementing asynchronous request/response by including a *reply channel* and *message identifier* in the request message.
2. The receiver processes the message and sends the reply to the *specified reply channel*.

Brokerless vs. Broker-based architectures

ตัวที่ว่า เช่น rabbitmq, Apache Kafka

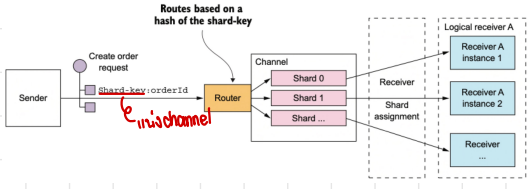
	Brokerless	Broker-based
Benefits	• Allows lighter network traffic and better latency. • Eliminates the possibility a performance bottleneck or a single point of failure • Features less operational complexity	ตรง sender ไม่จำเป็นต้องรู้ว่าแต่ละตัว consumer • Loose coupling – The client doesn't need to use a discovery mechanism • Message buffering (จนกว่า client จะพร้อมรับ) • Flexible communication - Messaging supports all the interaction styles • Explicit interprocess communication
Drawbacks	• Services must use one of the discovery mechanisms • Availability is reduced, because both the sender and receiver must be available while the message is being exchanged.	• Potential performance bottleneck • Potential single point of failure • Additional operational complexity

(ตรงข้ามกับ brokerless)



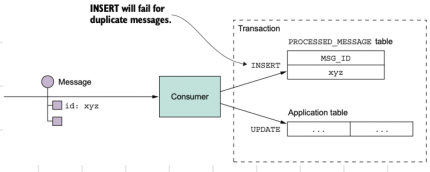
Competing receivers and message ordering

- A common solution, used by modern message brokers like Apache Kafka and AWS Kinesis, is to use sharded (partitioned) channels
-
  - Each Order event message has the *orderId* as its shard key → จะได้ไปที่ instance เดียวกัน
  - Each event is published to the same shard, which is read by a single consumer instance



Handing duplicate messages

- A message consumer has to discard any duplicates message → Tracking by the message id
-
  - A consumer record the IDs of processed messages in a database table.
  - If a message has been processed before, the INSERT into the PROCESSED_MESSAGES table will fail.



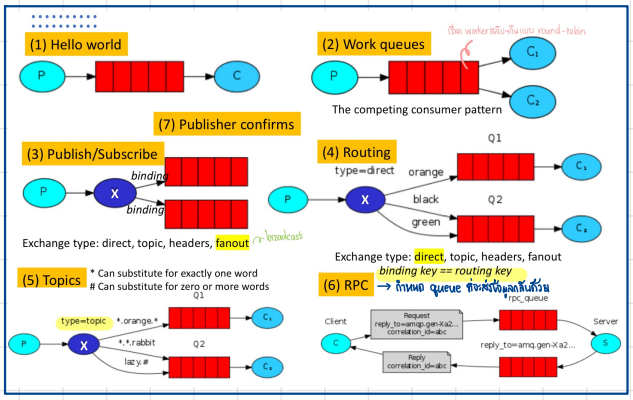
Summary

- The microservice architecture is a distributed architecture, so interprocess communication plays a key role.
- It's essential to carefully manage the evolution of a service's API.
- There are numerous IPC technologies, each with different trade-offs.
- To prevent failures, a service client that uses a synchronous protocol must be designed to handle partial failures
- An architecture that uses synchronous protocols must include a service discovery mechanism to determine the network location of a service instance.
- A good way to design a messaging-based architecture is to use the messages and channels model, typically message broker–based
- A key challenge when using messaging is atomically updating the database and publishing a message and a good solution is to use the Transactional outbox pattern

RabbitMQ

- RabbitMQ is an open-source distributed message broker.
- It is built on AMQP (Advanced Messaging Queuing Protocol).
- AMQP is an open standard application layer protocol.
- RabbitMQ accepts and forwards messages.
- It accepts, stores, and forwards binary blobs of data – *messages.*

Producing A program which sends message is a *producer.*	
Queue : a post box living inside RabbitMQ.	
Consuming A *consumer* is a program that mostly waits to receive messages.	



RabbitMQ & Apache Kafka

RabbitMQ	Kafka
Direct messaging: users can set sophisticated rules for message delivery.	Kafka is ideal for handling large amounts of homogeneous messages, such as logs or metric, and it is the right choice for instances with high throughput.
RabbitMQ will be usually used with Cassandra for message playback.	Replay messages
Multiprotocol supports: AMQP, STOMP, MQTT, Web sockets and others.	Big data consideration with e.g., Elastic search, Hadoop
Flexibility: varied point-to-point, request/reply, publish/subscribe	Scaling capability; topics can be split into partitions
Communication: supports both async and sync	Batches: Kafka works best when messages are batched.
Security	Security
Mature platform	Mature platform
Slower than Kafka	Don't come with user-friendly GUI but can be monitored via Kibana

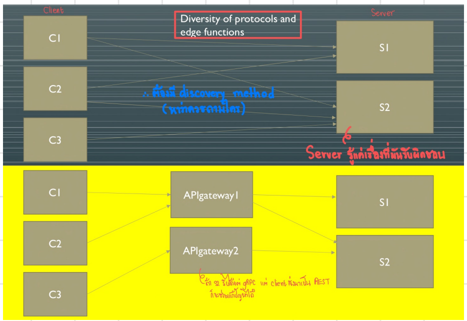
Kafka: Best used for basic streaming without complex routing with maximum throughput, ideal for event-sourcing, stream processing, multi-stage pipelines, routinely audited systems, real-time processing and analyzing data

RabbitMQ: Used for high throughput and reliable background jobs, integration and intercommunication between and within applications, perform complex routing to consumers, integrate multiple applications and services with non-trivial routing logic.

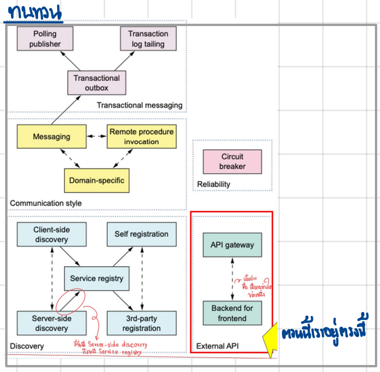
Week 8

External API Patterns

ถ้าไม่ต้องมี API Gateway

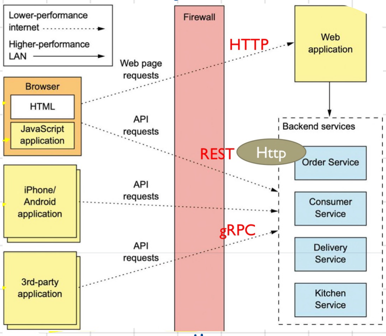


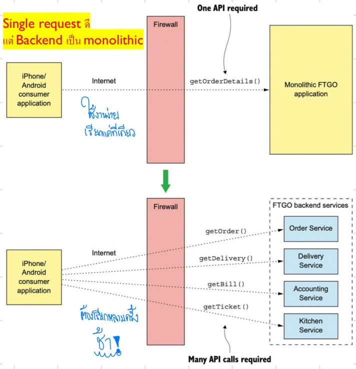
ทบทวน



ตอนนี้เราอยู่ตรงนี้

External API Design Issues





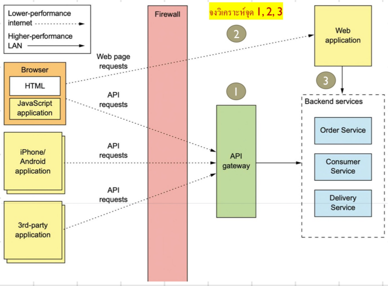
- One approach to API design is for clients to invoke the services directly.
  - Inefficient and can result in a poor user experience
  - Difficult to change the architecture and the APIs.
  - Inconvenient or impractical for clients to use, especially those clients outside the firewall.

ปัญหาที่เกิดขึ้น

- Poor user experience due to the client making multiple request
- Lack of encapsulation requires frontend developers to change their code in the lockstep with the backend
- Service might use client-unfriendly IPC mechanisms
  - Client applications that run outside the firewall typically use HTTP or Web Sockets.
  - Application's services might use gRPC or AMQP messaging protocol.

รวมถึงปัญหา lower-bandwidth, higher-latency ของฝั่ง client
และ Backend เปลี่ยน API หรือเปลี่ยน Endpoint Address
จากการมี 3rd party dev

The API Gateway Pattern



Overview

→ Encapsulates the application's internal architecture and provides an API to its clients.

API Gateway = Single-entry point into the application for API calls from outside the firewalls.

Encapsulate หมายถึงซ่อน Backends Implementation ไม่ให้ภายนอกเห็น หรือควบคุมการเข้าถึงได้ ถ้าเราเปลี่ยน Backends จะไม่กระทบ Client

API Gateway Responsibility

1. Request Routing = การหาเส้นทาง
   - When receiving a request, it consults a routing map that specifies which service to route the request to.
   - A routing map map an HTTP method and path to the HTTP URL of a service.
   - This function is identical to the reverse proxying features provided by web servers such as NGINX.

2. API Composition

   Client ส่งไป request เดียว แล้ว Gateway จะจัดการหลายๆ Request ให้

   - API composition enables a client to efficiently retrieve data using a single API request.

   Single request ผ่าน API gateway แล้ว API gateway ทำ data composition ให้ ทำให้ higher performance network

3. Protocol Translation

   เช่น REST แปลงเป็น gRPC

   - It might provide a RESTful API to external clients,
   - Even though the application services use a mixture of protocols, including REST and gRPC.

Single one-size-fits-all (OSFA) = ที่ระบบเรามี API Gateway ตัวเดียว สำหรับทุก Client

แต่ละ Client ต้องการไม่เหมือนกัน ไม่ fit all ได้

- An API gateway could provide a single one-size-fits-all (OSFA) API.
- But the different clients often have different requirements.
- A better approach is providing each client with its own API.
- The API gateway will also implement a public API for third-party developers to use.
- The Backends for frontends pattern takes this concept of an API-per-client even further by defining a separate API gateway for each client. ***

สรุป Kong ทำ Edge Function อะไรได้/ไม่ได้บ้าง

Implementing edge functions

Edge Function = A request-processing function implemented at the edge of an application

→ Authentication, Authorization, Rate Limit, Caching, Metrics collection, Request logging

Place to implement edge functions

There are three different places:

1. Implement in the backend services.
2. Implement in an edge service that's upstream from the API gateway.
3. Implement in the API gateway itself.

- This might make sense for some functions.
  - Caching
  - Metrics collection
  - Authorization
- But it's more secure if the application authenticates requests on the edge before they reach the services.

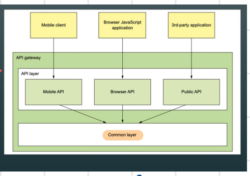
- The edge service is the first point of contact for an external client.
  - Authenticates the request and performs other edge processing before passing it to the API gateway.
- Benefits:
  - It separates concerns. The API gateway focuses on API routing and composition.
- Drawback:
  - It increases network latency because of the extra hop.
  - It also adds to the complexity of the application.

- There's one less network hop, which improves latency.
- There are also fewer moving parts, which reduces complexity.

API Gateway Architecture

API gateway = API layer + API Common layer

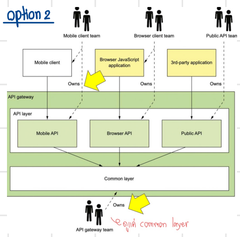
- An API gateway has a layered modular architecture.
  - The API for each client is implemented by a separate module.
  - The common layer implements functionality common to all APIs, such as authentication.



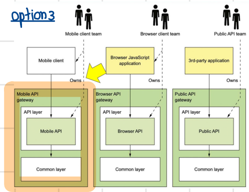
API Gateway Ownership Model → ใครเป็นคนรับผิดชอบ API Gateway

Option 1: Separate Team (มีทีม 1 ทีม ดูแล API Gateway โดยเฉพาะ) → ข้อเสีย: เกิด Centralized bottleneck

Option 2: Client Team ที่ทำ Service ต่างๆ จะทำ API Gateway ด้วย (ทำขึ้น API layer)
ส่วน Common Layer ดูแลโดย API Gateway Team (เช่น Netflix ใช้วิธีนี้)



Option 3: Using "Backends for Frontend" pattern
→ Each client team owns their API Gateway and common layer



API gateway มักจะมีจำนวนมากกว่าหนึ่งตัว ใช้ Backend for Frontend pattern

– Benefits of an API Gateway

- Encapsulates internal structure of the application.
- Reduces the number of round-trips between the client and application.
- Simplifies the client code.

– Drawbacks of an API Gateway

- Another highly available component that must be developed, deployed, and managed.
- Also a risk that the API gateway becomes a development bottleneck.
- Developers must update the API gateway in order to expose their services's API.
- The process for updating the API gateway be as lightweight as possible.
  → Otherwise, developers will be forced to wait in line in order to update the gateway.

lightweight API gateway = thin gateway คือมีการเขียน logic ตัว gateway น้อยที่สุดเท่าที่เป็นไปได้

– API Gateway Design Issues

1. Performance and scalability

"API Gateway ควรใช้ synchronous หรือ asynchronous I/O !?"

– Synchronous I/O model → program's execution is blocked, or synchronized while waiting for I/O operation to finish

- Each network connection is handled by a dedicated thread → Special thread for specific task เพราะ คือมอบหมายงานนั้น dedicated thread
- This is a simple programming model and works reasonably well.
- Limitation is that operating system threads are heavyweight.
- So there is a limit on the number of threads, and hence concurrent connections, that an API gateway can have.

Thread แต่ละ Heavyweight เพราะต้อง dedicated thread เพราะใช้ทรัพยากรที่เยอะ thread ให้น้อย

– Asynchronous I/O model (non blocking)

- A single event loop thread dispatches I/O requests to event handlers.
- Much more scalable because it doesn't have the overhead of using multiple threads.
- The drawback is the callback-based programming model is much more complex.
- Event handlers must return quickly to avoid blocking the event loop thread.

2. Writing maintainable code by using reactive programming abstractions

มีการเรียก service แบบ Sequential → ข้อเสีย: Response Time = Sum(Service Response Time)
↳ เพื่อจะ minimize เวลา เราควรเรียกแต่ละ service แบบ Concurrently

Listing 8.1 Fetching the order details by calling the backend services sequentially

```
@RestController
public class OrderDetailsController {
@RequestMapping("/order/{orderId}")
public OrderDetails getOrderDetails(@PathVariable String orderId) {

  OrderInfo orderInfo = orderService.findOrderById(orderId);

  TicketInfo ticketInfo = kitchenService
      .findTicketByOrderId(orderId);

  DeliveryInfo deliveryInfo = deliveryService
      .findDeliveryByOrderId(orderId);

  BillInfo billInfo = accountingService
      .findBillByOrderId(orderId);

  OrderDetails orderDetails =
      OrderDetails.makeOrderDetails(orderInfo, ticketInfo,
                                    deliveryInfo, billInfo);

  return orderDetails;
}
...
```

Sequential calls cause the long response time … concurrently calls are the solution but difficult to implement.

– ทางเลือกแรกที่เพิ่มเติม คือ ก็ใช้ callback

- The problem is this method returns a *Future*, which has a blocking API.
- The callback accumulates results.
- Once all of them have been received, it sends back the response to the client.

⇓
Can lead to callback hell
(Callback ซ้อนกันหลายชั้นจนอ่านยาก)

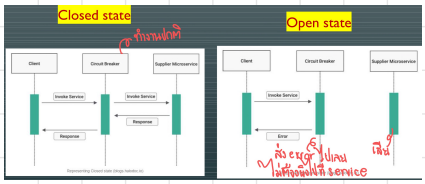
– ทางที่ดีกว่า คือ Write API composition Code in a declarative style using a reactive approach
☑ Code simple and easy to understand

Examples of reactive abstractions for the JVM include the following:
- Java 8 CompletableFutures
- Project Reactor Monos
- RxJava (Reactive Extensions for Java) by Netflix
- ScalaFutures

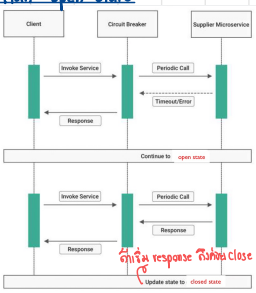
3. Handing partial failure

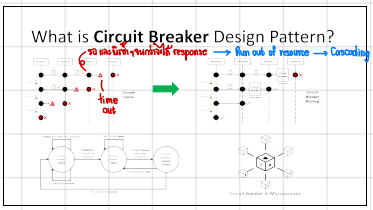
- API gateway must be reliable.
- One way is to run multiple instances of the gateway behind a load balancer.
- Another way is to properly handle failed requests and requests that have unacceptably high latency.
  - Using the Circuit breaker pattern when invoking services.

Circuit breaker pattern ทำหน้าที่ = Allow developers to prevent cascading failures
มี 3 states – closed, open, half-open state



Half-open state





Implementing an API Gateway

มี หลายวิธี ได้แก่ 1. Using an off-the-shelf API gateway product/service → Requires little or no development but is the least flexible
2. Developing your own API gateway → อาจใช้ Framework ต่างๆในการเริ่มขึ้น → Flexible but it requires some development effort.

ตัวอย่างผลิตภัณฑ์

1. AWS Application Load Balancer → AWS service that provides API gateway-like functionality
   - A load balancer for HTTP, HTTPS, WebSocket, and HTTP/2
   - You define routing rules that route requests to backend services
   - Drawback:
     - It does not implement HTTP method-based routing.
     - Nor does it implement API composition or authentication.

2. Kong หรือ Traefik
   - Kong is based on the NGINX HTTP server
   - Traefik is written in GoLang
   - Support edge functions and powerful routing capabilities
   - Drawback:
     - They don't support API composition **

Summary (1)

- Your application's external clients usually access the application's services via an API gateway.
- Your application can have a single API gateway or it can use the Backends for frontends pattern.
- The main advantage of the Backends for frontends pattern is that it gives the client teams greater autonomy because they develop, deploy, and operate their own API gateway.
- There are numerous technologies you can use to implement an API gateway, including off-the-shelf API gateway products or develop your own API gateway using a framework.

Week 9

Implementing Queries

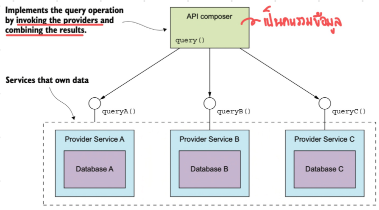
Overview of querying data in a microservice architecture

- An Application implements a variety of query operations → Querying in Monolithic → Straightforward (Because it has a single database)
  → Querying in Microservice → ยากกว่า เพราะ Queries often need to retrieve data scattered (กระจัดกระจาย) among databases owned by multiple services.
  ↳ Queries must join data from multiple services (แถมบาง data ก็ถือว่า private ไม่ควรดึงออกมาข้างนอก service!?)

- 2 Pattern for Implement query operations
  1. The API composition pattern
     • The simplest approach and should be used whenever possible.
  2. The Command query responsibility segregation (CQRS) pattern
     • More powerful than the API composition pattern, but also more complex.

Querying using the API composition pattern

→ ประกอบด้วย API Composer + ≥2 provider services

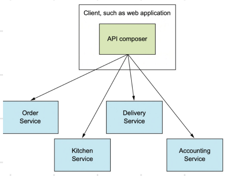


- This pattern implements a query operation by → 1. Invoking the services that own the data
  → 2. Combining the results

- The structure of this pattern has 2 types of participants
  → 1. Provider Service = A service that owns some of the data that the query returns
  → 2. API Composer (เป็น aggregator) = Implements the query operation by querying the provider services
    → might be
      – A client, such as a web application
      – A service, such as an API gateway
    → might need to perform in-memory join of large datasets

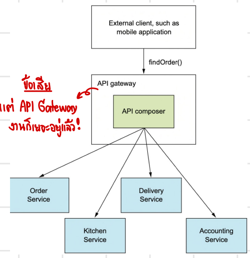
- To implement a particular query operation depends on
  - How the data is partitioned
  - The capabilities of the APIs exposed by the services that own the data
  - The capabilities of the databases used by the services

- *Design issues → 1. Which component in architecture is the query operation's API composer?
  ↳ 1) Using a Client to be the API Composer



Drawbacks: Probably not practical for Clients that are outside of the firewall and access services via a slower network + บางข้อมูลเราก็ไม่อยากให้ client รู้

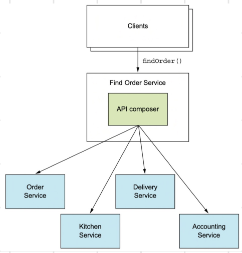
2) Using an API Gateway to play the role of an API Composer



ข้อเสีย แต่ API Gateway งานก็เยอะอยู่แล้ว!

- Make sense if the query operation is part of the application's external API
- Enables a client that is running outside of the firewall to efficiently retrieve data from numerous services with a single API call

3) Implement an API composer as a stand-alone service



- Use this option for a query operation that's used internally by multiple services
- Can also be used for externally accessible query operations whose aggregation logic is too complex to be part of an API gateway

→ 2. How to write efficient aggregation logic?

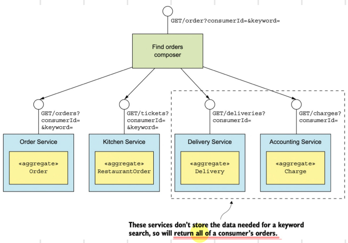
- API composer should use a reactive programming model
  - To program using and relying on events. (ถ้าไม่จำเป็นอย่าเขียนเป็น sequence เพราะต้องรอไปประมวล)
  - Instead of the order of lines in the code.
- An API composer should call provider services in parallel in order to minimize the response time for a query operation.
- The logic to efficiently execute a mixture of sequential and parallel service invocations can be complex.

Drawbacks of the API composition pattern

- Increased overhead
  - Computing and network resources are required.
  - Increasing the cost of running the application.
- Risk of reduced availability
  - The availability of an operation declines with the number of services that are involved.
- Lack of transactional data consistency
  - An API composer might not always be able to detect inconsistent data.

Querying using the CQRS pattern

- Motivations for using CQRS → 1. Not all services store the attributes that are used for filtering or sorting



Solution I: Do an in-memory join (Retrieves all orders จาก Delivery กับ Accounting Service —ต่อมา→ performs a join with orders ที่มาจาก Order และ Kitchen Service)
↳ Drawbacks: API composer must retrieve and join large datasets, which is inefficient.

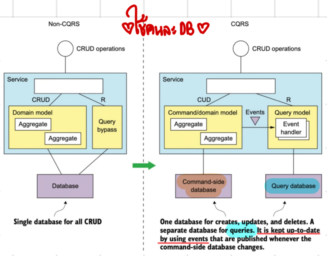
Solution II: Matching then request by fetching ID (Matching orders จาก Order และ Kitchen Service —ต่อมา→ request order จาก service อื่นๆ จาก ID)
Drawbacks: – Only practical if those services have a bulk fetch API
– Inefficient because of excessive network traffic

→ 2. Even queries that are local to a single service can be difficult to implement → เช่น findAvailableRestaurants() ที่ต้องการ efficient geospatial query
เพราะ → 1) Not appropriate for the service that owns the data to implement the query (There is a need to separate concerns)
2) Sometimes a service's database (or data model) doesn't efficiently support the query

Command Query Responsibility Segregation
↑
- Overview of CQRS

- Three problems commonly encountered when implementing queries in a microservice architecture :-
  1. Using the API composition pattern to retrieve data scattered across multiple services results in expensive, inefficient in-memory joins.
  2. The service that owns the data stores the data in a form or in a database that doesn't efficiently support the required query.
  3. **The need to separate concerns means that the service that owns the data isn't the service that should implement the query operation. (ฝั่ง read และ write ควรแยกออกจากกัน)
- The solution to all these problems is to use the **CQRS pattern**!!

- CQRS separates command from queries
- Restructures a service into command-side and query-side modules, which have separate databases.

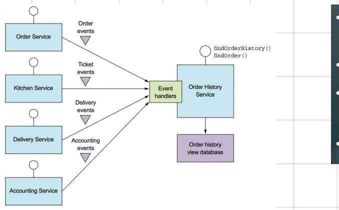


query-only service (no command operation)

We can use CQRS pattern to define query services.
→ Keep it up-to-date by subscribing to events published by ≥1 other Services
→ This kind of view doesn't belong to any particular service
So, it makes sense to implement it as a standalone service

— Example

- It implements the *findOrderHistory()* query operation by querying a database, which it maintains by subscribing to events published by multiple other services.
- The popular approach :-
  - Use RDBMS as the system of record
  - Use a text search engine, such as Elasticsearch, to handle text queries



— Benefits

- Enables the efficient implementation of queries in a microservice architecture.
- Enables the efficient implementation of diverse queries.
- Makes querying possible in an event sourcing-based application.
  - Capture all changes to an application state as a sequence of events.
- Improves separation of concerns.

— Drawbacks

- More complex architecture
  - Developers must write the query-side services that update and query the views.
  - An application might use different types of databases
- Dealing with the replication lag → ความล่าช้า
  - There's delay between the command side publishes an event and the query side processes the view updated. ← อัปเดตให้ไม่ทัน

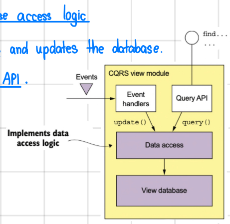
⇓

Solution to the "Replication Lag"

- Solution 1 - Supply the client with version information.
  - Enables the client to tell that the query side is out-of-date.
  - A client can poll the query-side view until it's up-to-date. (ต้องรอ)
- Solution 2 - Updating its local model once the command is successful without issuing a query. ← ยังอัปเดตไม่ถึง แต่ใช้ local model ช่วยจัดผลที่แสดง
  - Update its model using data returned by the command.
  - Drawback - The UI code may need to duplicate server-side code in order to update its model.

Designing CQRS views

- CQRS view module = 1 view database + 3 submodules
  - The data access module implements the database access logic.
  - The event handlers module subscribes to events and updates the database.
  - The query API module implements the query API.



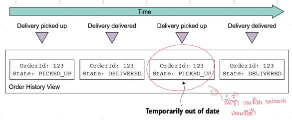
- Important decisions when developing a view module

1. Choose a database and design the schema
   - Issue 1 : SQL vs. NoSQL
     - NoSQL database is often a good choice for a CQRS view
       - More flexible data model + Better performance and scalability
     - Query-side view stores

If you need	Use	Example
PK-based lookup of JSON objects	A document store such as MongoDB or DynamoDB, or a key value store such as Redis	Implement order history by maintaining a MongoDB document containing the per-customer.
Query-based lookup of JSON objects	A document store such as MongoDB or DynamoDB	Implement customer view using MongoDB or DynamoDB.
Text queries	A text search engine such as Elasticsearch	Implement text search for orders by maintaining a per-order Elasticsearch document.
Graph queries	A graph database such as Neo4j	Implement fraud detection by maintaining a graph of customers, orders, and other data.
Traditional SQL reporting/BI	An RDBMS	Standard business reports and analytics.

   - Issue 2 : Supporting update Operations
     - Implement the efficient update operations executed by the event handlers
       - Using the primary key
       - Sometimes, using foreign key
     - Some types of database efficiently support foreign-key-based update operations.
       - RDBMS or MongoDB can create an index on the necessary columns

2. Designing the data access module ⇒ Ensuring that updates are idempotent (ได้ค่าเดิมเสมอ) + Handling concurrent updates
   - Issue 1 : Handling concurrency
     - The event handlers use the data access module, consisting of a data access object (DAO)
     - DAO must handle concurrent updates to the same database record.
       - If a view subscribes to events published by multiple aggregate types, It must not allow one update to overwrite another.



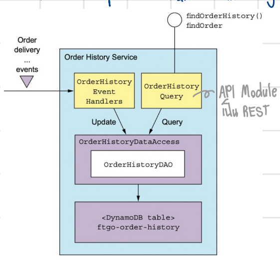
     - ควรได้เหมือนเดิม Idempotent event handlers
       - An event handler may be invoked with the same event more than once.
         - ถ้า query-side event handler เป็น idempotent → ไม่มีปัญหา
         - ถ้าเป็น non-idempotent ตัว event handler ต้อง detect และ discard duplicate events
           - Recording the IDs of processed events in the view datastore.
           - Recording the event ID and update the datastore atomically.
   - Issue 2 : Enable a client to use an eventually consistent view
     - A client that updates the command side and then immediately executes a query might not see its own update.
     - The command and query module APIs can enable the client to detect an inconsistency
       - A command-side operation returns a token containing the ID of the published event to the client.
       - The client then passes the token to a query operation, which returns an error if the view hasn't been updated by that event.

3. When implementing a new view (สร้างเพื่อ support new query) or changing the schema ⇒ Must implement a mechanism to efficiently buid or rebuild the view
   (รวมถึงการ re-create view ที่อาจเกิดขึ้นได้ เมื่อ Schema change หรือ เป็นการแก้ bug บางอย่างที่อัปเดต View)
   - To create a new view,
     - Develop the query-side module → set up the datastore, → deploy the service
   - To update an existing view
     - Change the event handlers → rebuild the view from scratch or update incrementally

4. Decide how to enable a client of the view to cope with the replication lag

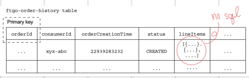
Implementing a CQRS view with AWS DynamoDB (→ NoSQL)

จะเป็นตัวอย่างการ Implement CQRS โดยมี Design เป็นดังภาพ



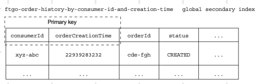
Design issues ที่เราต้องจัดการ

1. Design ftgo-order-history table



2. Define an index for the findOrderHistory query (return a consumer's orders sorted by increasing age)
   - Partition key → Primary key that has the consumerId
   - Sort key → orderCreationDate
   - เพราะในระบบ distributed data จะกระจายไปอยู่ตาม node ต่างๆ Partition key = ใช้ Identify ว่า node ไหนมี data นี้อยู่

ปรับใหม่ →



3. Implementing the findOrderHistory query (has a filter parameter that specifies the search criteria)
   - *key condition expression* supports a range restriction on the **sort key**.
   - The other filter criteria (Boolean) can be implemented using a *filter expression*.
   - The *OrderHistoryDaoDynamoDb* also enables the keyword search.
     - By tokenizing the restaurant name and menu items.
     - Storing the set of keywords in a set-valued attribute called keywords.

4. Paginate the query result (เพราะ consumers บางคนมีจำนวน order เยอะมากๆ)
   - The DynamoDB Query operation has an operation *pageSize* parameter
     - specifies the maximum number of items to return
     - If there are more items, the result of the query has a non-null *LastEvaluatedKey* attribute

5. Update order
   - DynamoDB supports two operations for adding and updating items: *PutItem()* and *UpdateItem()*.
     - The *PutItem()* operation creates or replaces an entire item by its primary key.
     - The *UpdateItem()* operation updates individual attributes of the item, creating the item if necessary.
   - The *UpdateItem()* operation is more efficient because there's no need to first retrieve the order from the table.

6. Detect duplicate events
   - *OrderHistoryDaoDynamoDb* can detect duplicate events by recording in each item the events that have caused it to be updated.
   - It can then use the *UpdateItem()* operation's conditional update mechanism to only update an item if an event isn't a duplicate.
   - A conditional update is only performed if a condition expression is true.

"Summary"

- Implementing queries that retrieve data from multiple services is challenging because each service's data is private.
- There are two ways to implement these kinds of query:
  - The API composition pattern
  - Command query responsibility segregation (CQRS) pattern
- The API composition pattern is the simplest way to implement queries and should be used whenever possible.
- A limitation of the API composition pattern is that some complex queries require inefficient in-memory joins of large datasets.
- The CQRS pattern, which implements queries using view databases, is more powerful but more complex to implement.
- A CQRS view module must handle concurrent updates as well as detect and discard duplicate events.
- CQRS improves separation of concerns by enabling a service to implement a query that returns data owned by a different service.
- Clients must handle the eventual consistency of CQRS views.

Week 10

Deploying MicroService

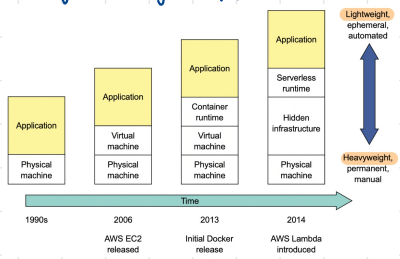
An overview of Deploying Microservice

Deployment = combination of two interrelated concepts ⇒ **process + architecture**

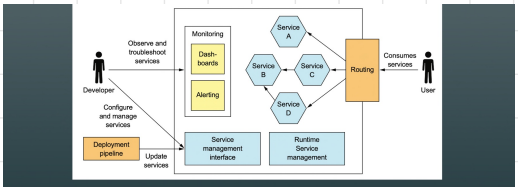
- **Process** consists of the steps that must be performed by people— developers and operations—in order to get software into production. → ใครทำอะไร เมื่อไหร่ อย่างไร
- **Architecture** defines the structure of the environment in which that software runs.

* Both aspects of deployment have changed radically since 1990s.

Heavyweight and long-lived physical machines has been abstracted away



by increasingly lightweight และ ephemeral technologies



- A simplified view of the production environment.
  1. **Service management interface** enables developers to deploy and manage their services
  2. **Runtime service management** ensures that the services are running
  3. **Monitoring** visualizes service behavior and generates alerts
  4. **Request routing** routes requests from users to the services

Deploying services using the Language-specific packaging format pattern

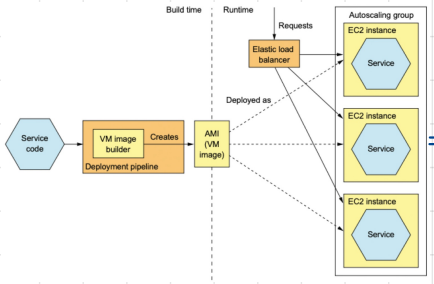
- Deploy ลงบนเครื่องที่เป็น Bare Machine ซึ่งแต่ละภาษามีวิธีการแตกต่างกัน เช่น ถ้า Spring Boot-based Java → ก็เป็น JAR file แล้วค่อยเอาไป Deploy

- **Benefits**
  - Fast deployment
  - Efficient resource utilization
- **Drawbacks**
  - Lack of encapsulation of the technology stack
  - No ability to constrain the resources consumed by a service instance (จำกัด)
  - Lack of isolation when running multiple service instances on the same machine
  - Automatically determining where to place service instances is challenging

Encapsulation in the context of software deployment typically involves packaging applications and their dependencies in a self-contained unit, ensuring that they can run independently without interfering with other services or applications on the same machine. This is a common feature provided by virtualization technologies, containers, and other deployment solutions.

Deploying services using the Service as a virtual machine pattern

- เช่น สร้างและ Config EC2 instance แล้วลากเอา Executable File ไฟล์นั้นมา
  - Created from Amazon Machine Image (AMI) <Better Option>
  - Managed by an AWS Auto Scaling Group
    - The desired number of healthy instances is always running

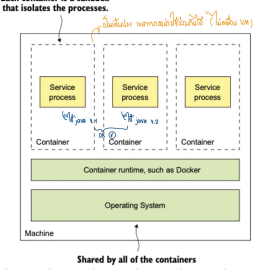


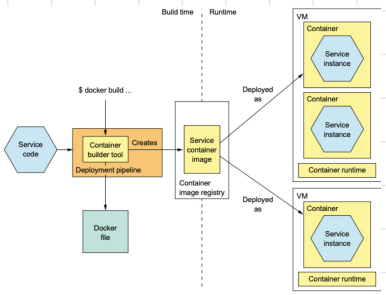
⇒
- The deployment pipeline packages a service as a VM image, such as an EC2 AMI
- At runtime, each service instance is a VM, such as an EC2 instance.
- An EC2 Elastic Load Balancer routes requests to the instances.

- **Benefits**
  - The VM image encapsulates the technology stack
  - Service instances are isolated
  - Uses mature cloud infrastructure → cloud VM is stable
- **Drawbacks**
  - Less-efficient resource utilization (resource ที่ถูกจองแต่ไม่ได้ใช้งานใน VM)
  - Relatively slow deployments → เพิ่ม/ลด VM ช้า + Set ตอนแรกเพื่อ Deploy ก็นานๆ
  - System administration overhead → ต้องมีคนคอยจัดการ VM

Deploying services using the Service as a container pattern

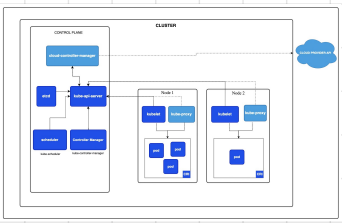
(more modern + lightweight เพราะ share OS กัน)





- A service is packaged as a container image
- At runtime the service consists of multiple containers instantiated from that image.
- A single VM will usually run multiple containers.

- **Benefits**
  - Many benefits of VM but more lightweight
  - Typically build faster than VM
- **Drawbacks**
  - Responsible for the heavy lifting of administering the container images, patching the operating system and runtime. → ดีกับจัดการ image
  - Also administering the container infrastructure and possibly the VM infrastructure it runs on. → ดีกับจัดการ Infrastructure ของ VM อีกชั้น



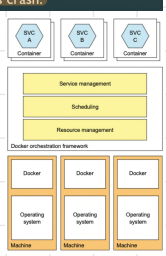
* Placing container into pods to run on Nodes

Deploying services using Kubernetes

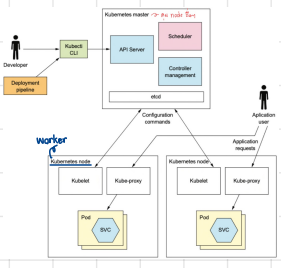
- **Kubernetes** is a **Docker orchestration framework**,
  - A layer of software on top of Docker that turns a set of machines into a single pool of resources for running services.
- The desired number of instances of each service keeps running at all times,
  - Even when service instances or machines crash. (มี Self-Healing)

3 Main Functions

- **Resource management**—Treats a cluster of machines as a pool of CPU, memory, and storage volumes
- **Scheduling**—Selects the machine to run your container
- **Service management**—Implements the concept of named and versioned services that map directly to services in the microservice architecture.



K8S Architecture



- **API Server**—The REST API for deploying and managing services
- **Etcd**—A key-value NoSQL database that stores the cluster data.
- **Scheduler**—Selects a node to run a pod.
- **Controller manager**—ensure that the state of the cluster matches the intended state.

- **Kubelet**—Creates and manages the pods running on the node
- **Kube-proxy**—Manages networking, including load balancing across pods
- **Pods**—The application services

♥ K8s มี Service discovery mechanism ให้เป็น built-in

♥ Service = K8s object that provides the clients of one or more pods with a stable endpoint (DNS Name)

- Deploying the API Gateway (Route traffic from the outside world to the service)
  - **ClusterIP service** ⇒ A type of service that exposes an application or set of pods within the cluster to other pods within the same cluster → for internal communication (เป็นตัว default)
  - **NodePort service** ⇒ A type of service that exposes an application or set of pods to the external world, allowing external traffic to reach the service on a specific port
  - **LoadBalancer** ⇒ (เหมือน nodeport) ‖, but it provisions an external load balancer from a cloud provider to distribute traffic across the nodes in your cluster.

- Using a service mesh to separate deployment from release
  - Deployment—Running in the production environment
  - Releasing a service—Making it available to end-users

  → แต่ไม่ release ให้คนหลายคน เพื่อลองใช้ User ข้างนอกเห็น 10% ก่อน เสถียรก่อนเป็น 20% เป็นต้น

  - **Steps :**
    1. Deploy the new version into production without routing any end-user requests to it.
    2. Test it in production.
    3. Release it to a small number of end-users.
    4. Incrementally release it to an increasingly larger number of users until it's handling all the production traffic.
    5. If at any point there's an issue, revert to the old version
  - using "service mesh" to make this style of deployment is a lot of easier ⇒ ตัวอย่างเครื่องมือ : Istio
    - A service mesh is a networking infrastructure that mediates all communication between a service and other services and external applications.
    - A service mesh provides rule-based load balancing and traffic routing
      - let you safely run multiple versions of your services simultaneously. ***

- Istio as an "An open platform to connect, manage, and secure microservices" (https://istio.io)
- Istio has a rich set of features organized into four main categories:
  - Traffic management—Includes service discovery, load balancing, routing rules, and circuit breakers
  - Security—Secures interservice communication using Transport Layer Security (TLS)
  - Telemetry—Captures metrics about network traffic and implements distributed tracing
  - Policy enforcement—Enforces quotas and rate limits

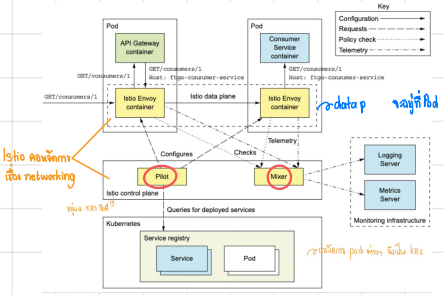
Deploying services using the Serverless deployment pattern

- Undifferentiated "Heavy Lifting"
  - Heavy lifting = การที่เราต้องดูแลของต่ำๆ (VM, Container, Physical Machine) ที่ต้องมีคนที่มีทักษะมาดูแล

  แต่มันมีสิ่งนึงที่ไม่ต้องทำ "Serverless"

- AWS Lambda = example of serverless deployment technology
  - AWS Lambda was initially for deploying event-driven services.
  - AWS Lambda automatically runs enough instances of your microservice to handle incoming requests.
    - Billed for each request based on the time taken and the memory consumed.
  - No needs to worry about any aspect of servers, virtual machines, or containers!!

Architecture



Istio คอนฟิกการเชื่อม networking

Pilot = คนคุมกับ data plane ที่อยู่ข้างบน
Mixer = คอย monitoring เก็บ เช่น ๆ Log

- It consists of a control plane and a data plane.
  - The control plane implements management functions, including configuring the data plane to route traffic.
  - The data plane consists of Envoy proxies, one per service instance.

เป็น proxy server รูปแบบหนึ่งที่พบใน Cloud

A **proxy server** is an intermediary server that acts as a gateway between client devices (such as computers or smartphones) and other servers (typically web servers or other online services). It sits between the client and the destination server, forwarding client requests to the server and returning the server's responses back to the client.

- The two main components of the control plane are the Pilot and the Mixer.
  - The Pilot extracts information about deployed services from the underlying infrastructure.
  - The Mixer collects telemetry from the Envoy proxies and enforces policies.

Benefits of AWS Lambda

- Integrated with many AWS services
- Eliminates many system administration tasks
- Elasticity—No need to predict the capacity of VMs or containers.
- Usage-based pricing

Drawbacks of AWS Lambda

- Long-tail latency — งานที่ Serious เช่นเวลา เช่น ซื้อหุ้น → ไม่เหมาะ
  - Because AWS Lambda dynamically runs your code,
  - Some requests have high latency because of the time it takes for AWS to provision an instance of your application to start.
- Limited event/request-based programming model
  - AWS Lambda isn't intended to be used to deploy long-running services,
  - Such as a service that consumes messages from a third-party message broker.

Summary

- You should choose the most lightweight deployment pattern that supports your service's requirements.
- Evaluate the options in the following order:
  1. serverless
  2. containers
  3. virtual machines
  4. language-specific packages

- A serverless deployment eliminates the need to administer operating systems and runtimes.
- It provides automated elastic provisioning and request-based pricing.
- A serverless deployment isn't a good fit for every service.
  - Long-tail latencies
  - Required to use an event/request-based programming model.

- Docker containers, a lightweight, OS-level virtualization technology, are more flexible than serverless deployment and have more predictable latency.
- It's best to use a Docker orchestration framework such as Kubernetes.
- The drawback is that you must administer the operating systems and runtimes.

- Deploying your service as a virtual machine is a heavyweight deployment option and will most likely use more resources than Docker containers.
- Deploying your services as language-specific packages is generally best avoided unless you only have a small number of services.

Week 11

Testing in Microservices Part I

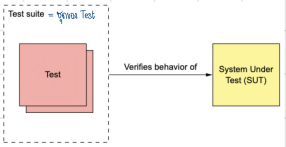
Overview of testing microservice

- Testing is primarily an activity that happends after development
- Why we need automated testing → ① Manual testing is extremely inefficient (กลับไป-กลับมา เสียเวลา) ② Manual testing is done far too late in the delivery process
- Workflow should be : edit code → run tests (ideally with a single keystroke) → repeat

Testing strategies for microservice architectures

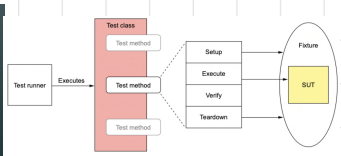
- Goal : verify the behavior of the System Under Test (SUT) — might be as small as a class or as large as an entire application

Test suite = A collection of related tests



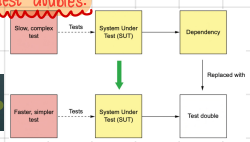
- Writing Automated Tests

- An automated test typically consists of four phases:
  1. Setup — Initialize the test fixture, which consists of the SUT and its dependencies, to the desired initial state.
  2. Exercise — Invoke the SUT—for example, invoke a method on the class under test. (ใช้งาน method ที่ต้องการทดสอบ)
  3. Verify — Make assertions about the invocation's outcome and the state of the SUT.
  4. (optional) Teardown — Clean up the test fixture, if necessary. Many tests omit this phase, but some types of database test will, for example, roll back a transaction initiated by the setup phase.



- Testing using mocks and stub
  - Replace the SUT's dependencies with Test doubles.*
  - ∴ SUT can be tested in isolation
  - A stub is a test double that returns values to the SUT.
  - A mock is a test double that a test uses to verify that the SUT correctly invokes a dependency. Also, a mock is often a stub.

(แต่ที่เห็นว่า Stub กับ mock ก็ใช้คล้ายๆกัน)



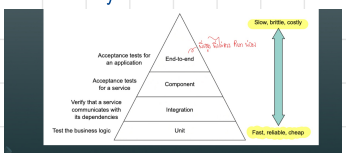
- The different type of tests

(We focus on automated tests that verify the functional aspects (functional requirement) of the application or service)

- There are four different types of tests:
  1. Unit tests — Test a small part of a service, such as a class.
  2. Integration tests — Verify that a service can interact with infrastructure services such as databases and other application services.
  3. Component tests — Acceptance tests for an individual service.
  4. End-to-end tests — Acceptance tests for the entire application.

(As a user, I want .... (Acceptance ตามที่เราทดสอบนี้))

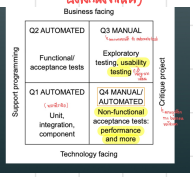
- Test Pyramid



- The test pyramid describes the relative proportions of each type of test.
- As you move up the pyramid, you should write fewer and fewer tests.

- Brian Marick's test quadrant (manual test ยังไงก็มีจำเป็น)

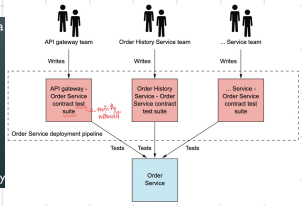
- The test quadrant categorizes tests along two dimensions.
- The first dimension is whether a test is business facing or technology facing.
- The second is whether the purpose of the test is to support programming or critique the application.



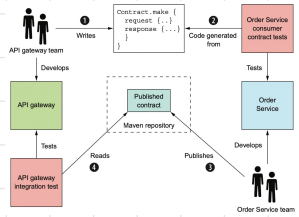
The challenge of testing microservices → ปัญหา: Many communication !?

→ Customer-Driven Contract Testing

- A consumer contract test focuses on verifying that the "shape" of a provider's API meets the consumer's expectations.
- For a REST endpoint, a contract test verifies that the provider implements an endpoint (for the consumer) that
  - Has the expected HTTP method and path
  - Accepts the expected headers, if any
  - Accepts a request body, if any
  - Returns a response with the expected status code, headers, and body



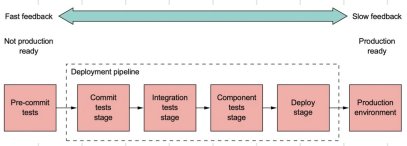
- The interaction between a consumer and provider is defined by a set of examples, known as contracts.
  - Each contract consists of example messages that are exchanged during one interaction.
- Even though the focus of consumer-driven contract testing is to test a provider, contracts are also used to verify that the consumer conforms to the contract.
  - Testing both sides of interaction ensures that the consumer and provider agree on the API



The deployment pipeline

- Every service has a deployment pipeline.
- Continuous Delivery (CD) describes a deployment pipeline as the automated process of getting code from the developer's desktop into production.
  - Ideally, it's fully automated, but it might contain manual steps.
- A deployment pipeline is often implemented using a Continuous Integration (CI) server, such as **Jenkins**.

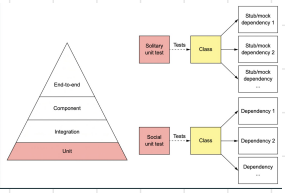
1. Pre-commit tests stage — Runs the unit tests. This is executed by the developer before committing their changes.
2. Commit tests stage — Compiles the service, runs the unit tests, and performs static code analysis.
3. Integration tests stage — Runs the integration tests.
4. Component tests stage — Runs the component tests for the service.
5. Deploy stage — Deploys the service into production.



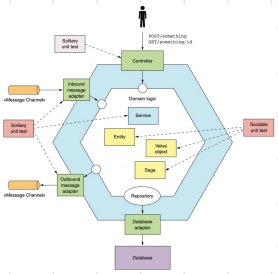
Writing unit tests for a service

- Unit tests are the lowest level of the test pyramid.
- They're technology-facing tests that support development.
- A unit test verifies that a unit, which is a very small part of a service, works correctly.
- A unit is typically a class, so the goal of unit testing is to verify that it behaves as expected.

- There are two types of unit tests*
  - Solitary unit test — Tests a class in isolation using mock objects for the class's dependencies
  - Sociable unit test — Tests a class and its dependencies



- The responsibilities of the class and its role determine which type of test to use.
  - Controller and service classes are often tested using solitary unit tests.
  - Domain objects, such as entities and value objects, are typically tested using sociable unit tests.



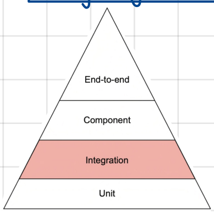
- Entities, such as Order, are objects with persistent identity → sociable unit tests.
- Value objects, such as Money, are objects that are collections of values → sociable unit tests.
- Sagas, such as CreateOrderSaga, maintain data consistency across services → sociable unit tests.
- Domain services, such as OrderService, are classes that implement business logic that doesn't belong in entities or value objects → solitary unit tests.
- Controllers, such as OrderController, which handle HTTP requests → solitary unit tests.
- Inbound and outbound messaging gateways → solitary unit tests.

Summary

- Automated testing is the key foundation of rapid, safe delivery of software
- The purpose of a test is to verify the behavior of the system under test (SUT).
- A good way to simplify and speed up a test is to use test doubles.
- Use the test pyramid to determine where to focus your testing efforts for your services.

Testing in Microservices Part II

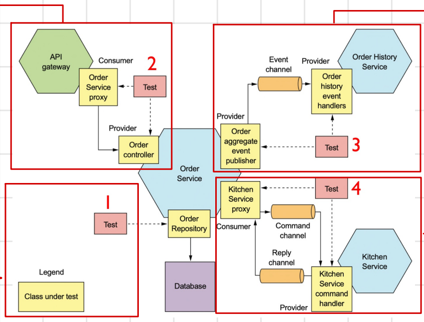
Writing integration tests



- They verify that a service can communicate with its dependencies and clients.

(Includes infrastructure services such as the database, application services)

- ~~Testing whole services~~, The strategy is to test the individual adapter classes that implement the communication

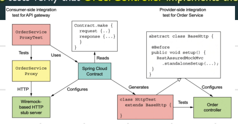


1. Persistence Integration tests

- Verifying that a service's database access logic works as expected.
- Each phase of a persistence integration test behaves as follows:
  - Setup — Set up the database by creating the database schema and initializing it to a known state. It might also begin a database transaction.
  - Execute — Perform a database operation.
  - Verify — Make assertions about the state of the database and objects retrieved from the database.
  - Teardown — An optional phase that might undo the changes made to the database.
    - for example, rolling back the transaction that was started by the setup phase.

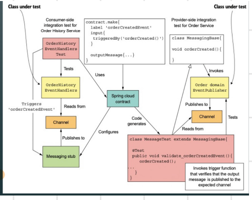
2. Integration testing REST-based request/response style interactions

- The contracts are used to verify that the adapter classes on both sides.
- The consumer-side tests verify that OrderServiceProxy invokes Order Service correctly.
- The provider-side tests verify that OrderController implements the REST API endpoints correctly.



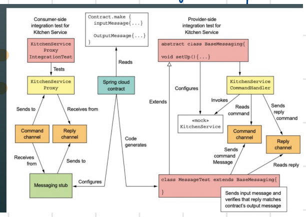
3. Integration testing publish/subscribe-style interactions

- The contracts are used to test both sides of the publish/subscribe interaction.
- The provider-side tests verify that OrderDomainEventPublisher publishes events that confirm to the contract.
- The consumer-side tests verify that OrderHistoryEventHandlers consume the example events from the contract.



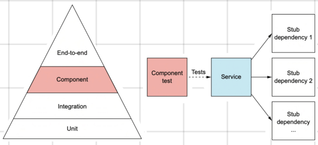
4. Integration tests for asynchronous request/response interactions

- The contracts are used to test the adaptor classes.
- The provider-side tests verify that KitchenServiceCommandHandler handles commands and sends back replies.
- The consumer-side tests verify KitchenServiceProxy sends commands that conform to the contract, and that it handles the example replies from the contract.



Developing component tests

- We want to write the service's acceptance tests, which treat it as a black box and verify its behavior through its API
- Component testing verifies the behavior of a service in isolation.
- It replaces a service's dependencies with stubs that simulate their behavior



- Acceptance Tests = Business-facing tests for a software development →
  - They describe the desired externally visible behavior from the perspective of the component's clients rather than in terms of the internal implementation.
  - These tests are derived from user stories or use cases.

using → Gherkin ⇒
- Gherkin is a DSL (domain specific language) for writing executable specifications.
- When using Gherkin, you define your acceptance tests using English-like scenarios.
- You then execute the specifications using (for Run Gherkin) Cucumber, a test automation framework for Gherkin.
- Gherkin and Cucumber eliminate the need to manually translate scenarios into runnable code.

Story

```
As a consumer of the Order Service
I should be able to place an order
```

⇒ Scenarios (Given → test's setup)

```
Given a valid consumer
Given using a valid credit card
Given the restaurant is accepting orders
When I place an order for Chicken Vindaloo at Ajanta
Then the order should be APPROVED
And an OrderAuthorized event should be published
```

When → Map to execute phase

Then / And → verification phase
→ "ถ้าเป็น integration test จะมี then แค่ 1 อัน"

Writing end-to-end tests

- End-to-end tests are business-facing tests.
- You can write the end-to-end tests using Gherkin and execute them using Cucumber.
- The main difference is that rather than a single "Then", this test has multiple actions.

- End-to-end tests must run the entire application, including any required infrastructure services.
- The implementation of the end-to-end test is quite similar to the implementation of the component tests.
- These tests are written using Gherkin and executed using Cucumber.

Summary (1)

- Use contracts to drive the testing of interactions between services.
- Write tests that verify that the adapters of both services conform to the contracts.
- Write component tests to verify the behavior of a service via its API.
- You should simplify and speed up component tests by testing a service in isolation, using stubs for its dependencies.

Summary (2)

- Write user journey tests to minimize the number of end-to-end tests, which are slow, brittle, and time consuming.
- A user journey test simulates a user's journey through the application and verifies high-level behavior of a relatively large slice of the application's functionality.