Chapter 08 External API Gateway

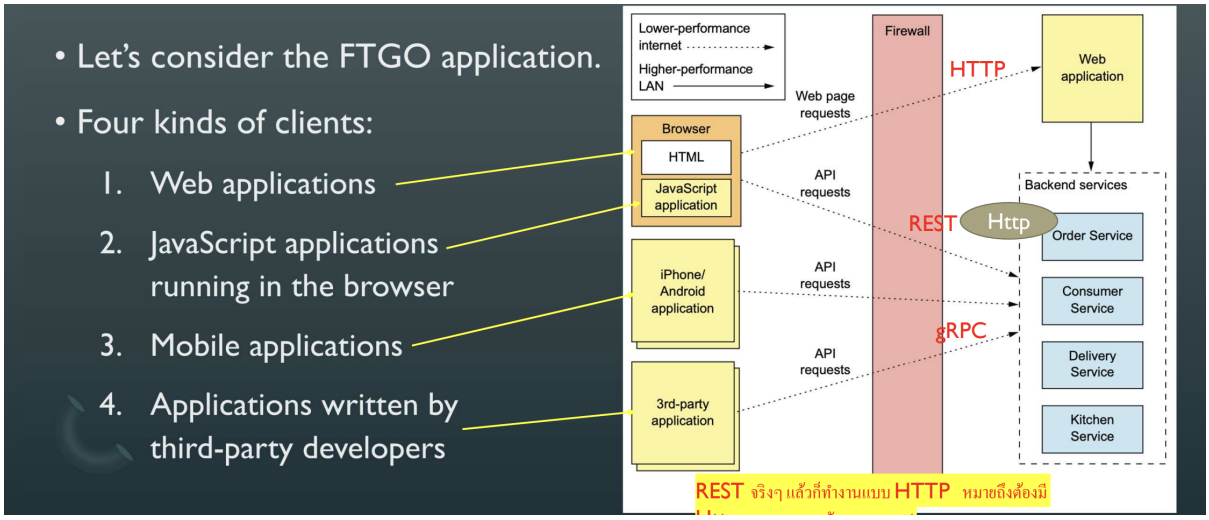
Introduction

เมื่อก่อนการเชื่อม Client กับ Server เป็นการเชื่อมต่อแบบ **Point to Point**

- มีหลาย **Protocol** และมีหลาย **End-point address** ที่ Client จะต้องรู้ อีกทั้งต้องรู้ด้วยว่า Endpoint ไหนคือ Service ไหนและให้บริการอะไร
  - เช่นการทำ **Edge Function** ต่าง ๆ (Authentication, Authorization) ว่าควรจะทำที่ตรงไหน
- **Decoupled** ระหว่าง Client - Server โดยการผ่านตัวกลางคือ **API Gateway**

External API Design Issues

> **External API** >> API (ภาษากลาง) ที่ผู้ใช้บริการนอกระบบของเรามาเรียกใช้



External API Gateway Issues

1. **Lower bandwidth, Higher Latency** >> Clients ยิงเข้ามาหา Services ด้วย WAN ซึ่งช้า
2. **Poor user experience**
3. **Difficult to change** >> แก้ไข Architecture และ API ได้ยาก
4. **Lack of Encapsulation** >> Clients รู้ว่า Backend มีกี่ Services และข้อมูลอื่น ๆ
5. **(Mobile client)**
   - **Single Request** >> ส่ง API request ไปที่ Monolithic Server เพื่อขอข้อมูลทุกอย่างมาในครั้งเดียว
     - ง่าย แต่ต้องเป็น Monolithic ซึ่ง Scalability ไม่ดี
   - **Multiple Request** >> ส่ง API request หลาย ๆ อันไปที่หลาย ๆ Microservices เพื่อค่อย ๆ เอาข้อมูลทีละส่วนตามต้องการ
     - Scalability ดี แต่การจะได้มาซึ่งข้อมูลบางอย่าง ต้องส่งหลาย ๆ requests ทำให้ใช้งานยากขึ้น (Client ต้องทำ Data Composition คือเป็นคนรวมข้อมูลเอง)
   - **Unfriendly IPC Mechanism** >> Web App อาจจะใช้ AMQP, gRPC ในการติดต่อกับ Server (มีหลาย Protocol แล้วยุ่งยาก)
6. **(Web application)** ถ้า web server อยู่หลัง firewall จะไม่มีปัญหา เพราะใช้ LAN คุยกับ Microservices ได้
7. **(Browser-based JS application)** **Network Latency**
8. **(Third-party application)** **Unstable API risk** >> จากการที่ Backend API อาจมีการเปลี่ยนแปลง (การมี Gateway มาเป็นตัวคั่นกลางจะสามารถช่วย thrid-party user ได้)

API Gateway Pattern

ทำ Single Entry Point (API Gateway) ซึ่งก่อให้เกิดการ Encapsulates ตัว Services มีหน้าที่ คือ

1. จัดการ **Request Routing**
   - **Routing Map** >> specify which service to route the request to (Mapping HTTP method & Path)
2. จัดการ **API Composition** ให้ >> user ส่ง single request แต่ API gateway ทำทั้ง multiple request และ data composition ให้
3. ทำ **Protocol Transition** เช่น การแปลงจาก REST (Client - API Gateway) เป็น gRPC (API Gateway - Services) ให้

Single One-Size-Fits-All (OSFA)

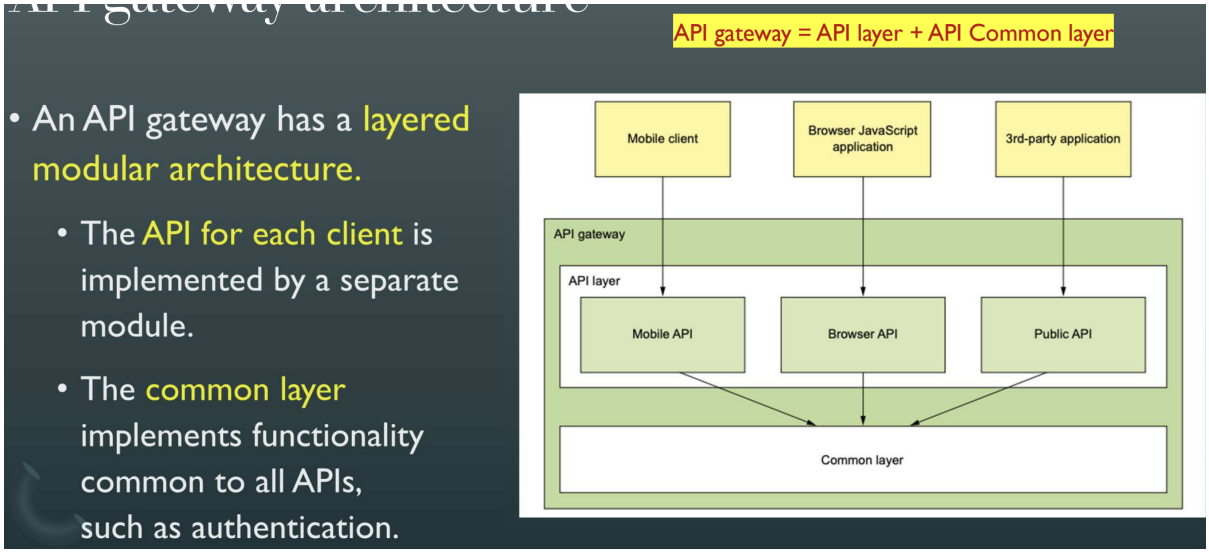
- API Gateway แค่ตัวเดียว อาจจะเรียกได้ว่าเป็น OSFA API ซึ่งอาจไม่พอสำหรับ requirements ของผู้ใช้
- แต่ละประเภทของ Client ควรมี API ที่เหมาะสมกับตัวเอง
  - **Backends for Frontend Pattern** เป็น Design pattern ที่ตอบโจทย์ในส่วนนี้

Implementing Edge Functions

- **Edge Function** >> function ที่ทำงานที่ edge ก่อนจะส่ง Request ไปทำต่อ เช่น
  1. Authentication & Authorization
  2. Rate limit & Caching
  3. Request Logging & Collect Metric
- Implementation ได้ 3 ที่
  1. ที่ **Backend Service** >> อาจจะไม่ดี เพราะ Service อื่น ๆ ต้องมาทำ Authentication
     - เหมาะสมแค่บาง function เช่น Caching, Authorization, Metric collection
  2. ที่ **Edge Service** >> สร้าง service แยกที่ทำ Edge function ก่อนส่งไป Backend service (หรืออาจจะวางไว้ก่อน API Gateway ก็ได้)
     - เป็นหลักการที่เรียกว่า Separation of Concerns
     - ข้อดี >> ลดภาระ API Gateway (Gateway focus ไปที่ Routing อย่างเดียว)
     - ข้อเสีย >> เกิด Latency ขึ้น ซับซ้อนขึ้น
  3. ที่ **API Gateway**
     - ข้อดี >> ไม่มี Latency ไม่มี Extra hop
     - ข้อเสีย >> เป็น Bottleneck, Single Point of Failure

> 💡 API Gateway แก้ข้อเสียยังไง >> ใช้ Load balancer และมีหลาย API Gateway Instance

API Gateway Architecture and Ownership Model



- แบ่งออกเป็น **2 layer**
  - **API Layer** >> กลุ่มของ API สำหรับให้บริการ Client ต่าง ๆ
  - **Common Layer** >> ส่วนของ Edge Function และ Business logic ต่าง ๆ
- **Ownership** >> ทีมใดรับหน้าที่ในการ Implement API Logic
  1. **Separated team** >> เป็นทีมเฉพาะกิจ ตั้งขึ้นมาเพื่อดูแล API Gateway หากใครจะแก้ไขอะไรต้องมาที่ทีม ๆ นี้
     - **ข้อเสีย** >> Centrailized Bottleneck
  2. **Client teams owns API module, API team own the common module**
     - **ข้อดี** >> Client teams แก้ไข API Layer ได้โดยตรง
  3. **Backend for Frontend pattern** >> ต่างทีมต่างดูแลทั้ง 2 Layer ของตนเองเลย ไม่ต้องมี API Team
     - **ข้อดี** >> Bottleneck น้อยสุด

Benefits and Drawbacks

ข้อดี

1. Encapsulates internal structure
2. Reduce the number of round-trips ระหว่าง Client - Application (Single request to API Gateway)
3. Simplifies code

ข้อเสีย

1. ต้องมี Availability สูง และอาจเป็น Bottleneck
2. การอัพเดทต้องรีบทำ รีบเสร็จ เพราะรอนานไม่ได้ >> AIP Gateway ต้องมีความเป็น lightweight (thin gateway ทำ logic น้อย ๆ)

Implementing an API Gateway

เวลาออกแบบมีสิ่งที่ต้องคำนึงถึง

- **Performance and Scalability**
  - ถ้าเป็น **Synchronous** จะ Heavyweight เพราะถูก handle ด้วย **dedicated thread** (จอง connection ไว้ ทำให้มีจำนวนได้จำกัด)
  - ถ้าเป็น **Asynchronous** เมื่อไหร่ที่ Callback ค่อยส่งข้อมูลกลับไป
- Write maintainable code using **reactive programming**
- **Handling Partial Failure**
  - มี load balancer และ multiple instance
  - ลด latency เมื่อเกิด failure ด้วย Circuit Breaker (ไม่ต้องรอ timeout)
    - Closed state >> ทำงานปกติ
    - Open state >> Reject ทันที
    - Half-open state >> ทำ Periodically calls เพื่อตรวจสอบว่ายังพังอยู่หรือไม่ หรือหายพังแล้ว
- Being a **good citizen** (in the architecture)