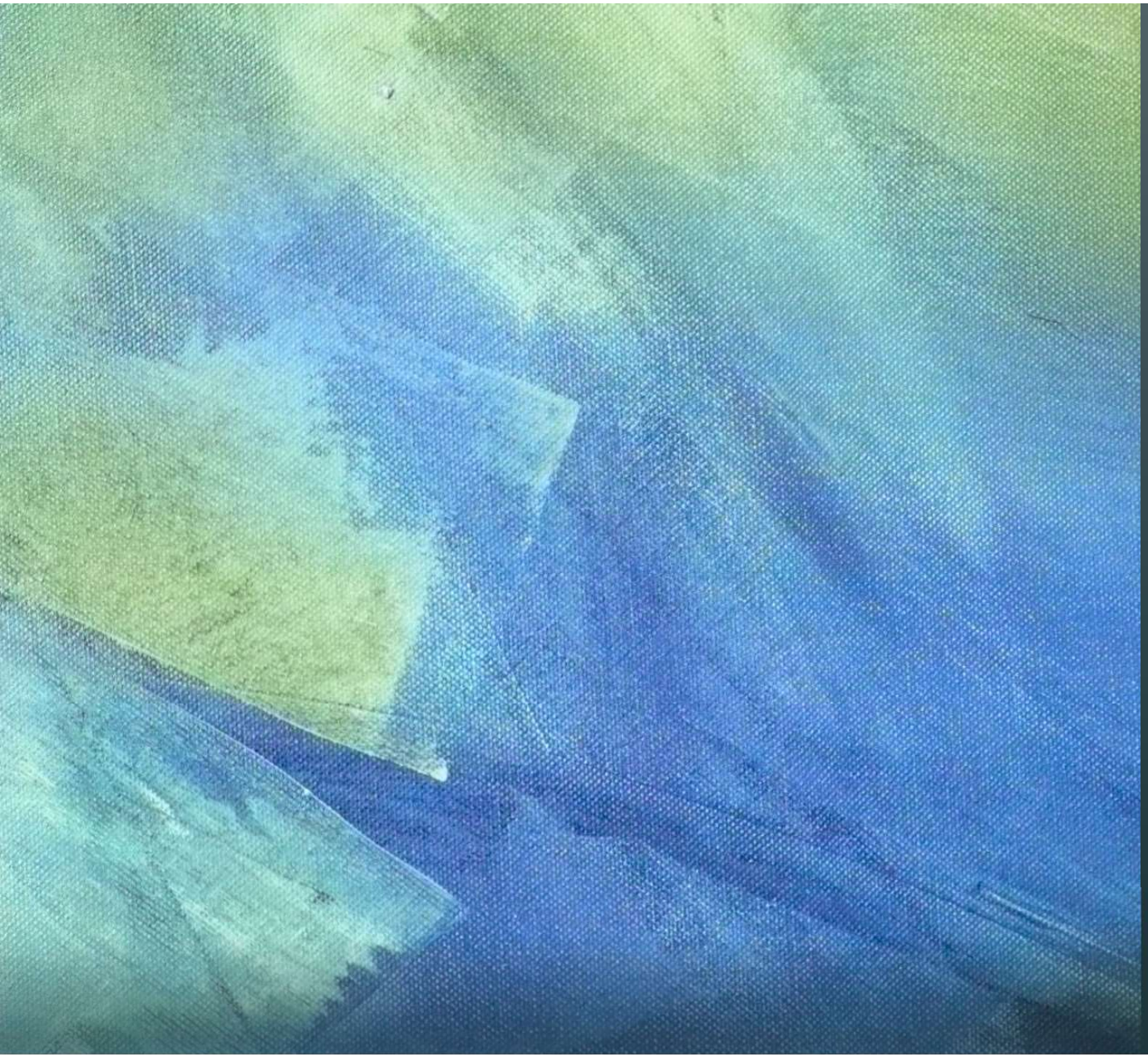


External API Pattern

Wiwat Vatanawood

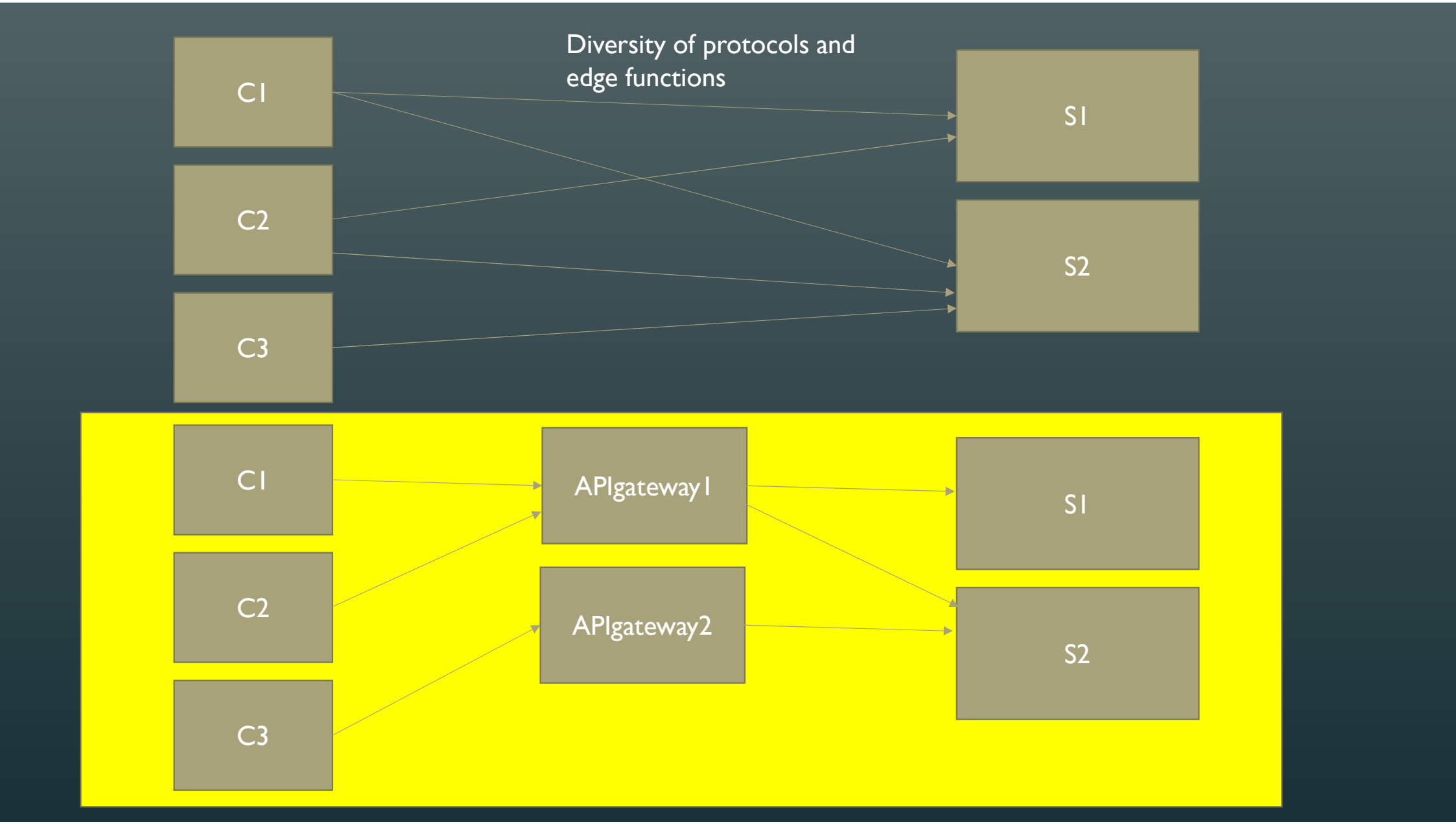
Duangdao Wichadakul

Pittipol Kantavat

Nuengwong Tuaychareon

This chapter covers

- The challenge of designing APIs that support a diverse set of clients
- Applying "API gateway" and "Backends for frontends" patterns
- Designing and implementing an API gateway

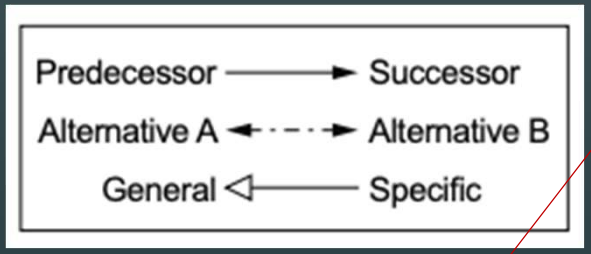

C1
C2
C3
Diversity of protocols and
edge functions
S1
S2
C1
C2
C3
APIgateway1
APIgateway2
S1
S2

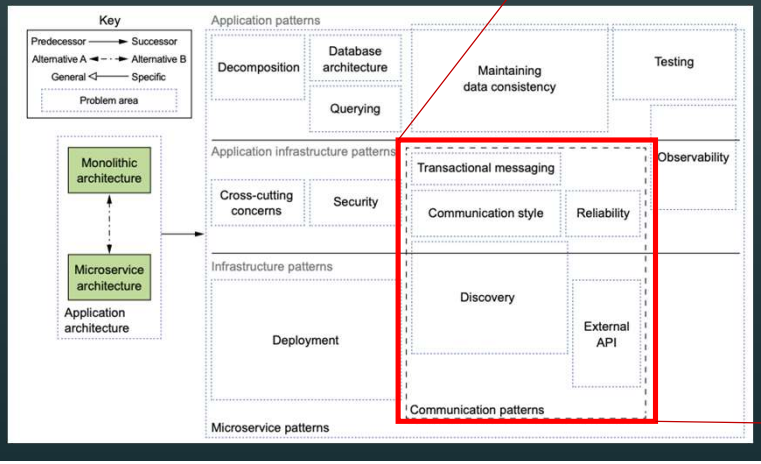
Agenda

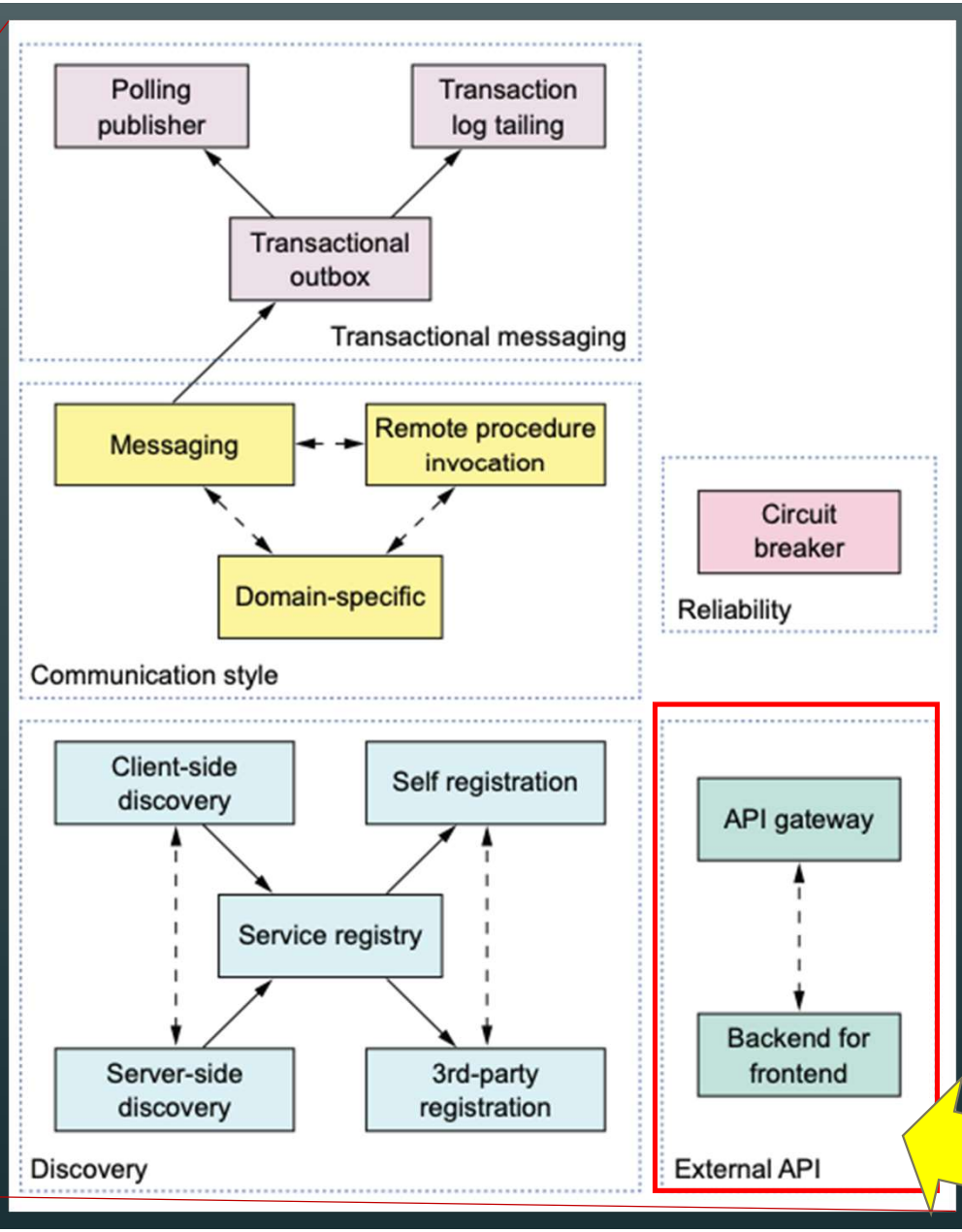
API ที่ออกแบบให้ผู้ใช้บริการภายนอกระบบเรา

- **External API design issues**
- The API gateway pattern
- Implementing an API gateway

The five groups of communication patterns (recall)

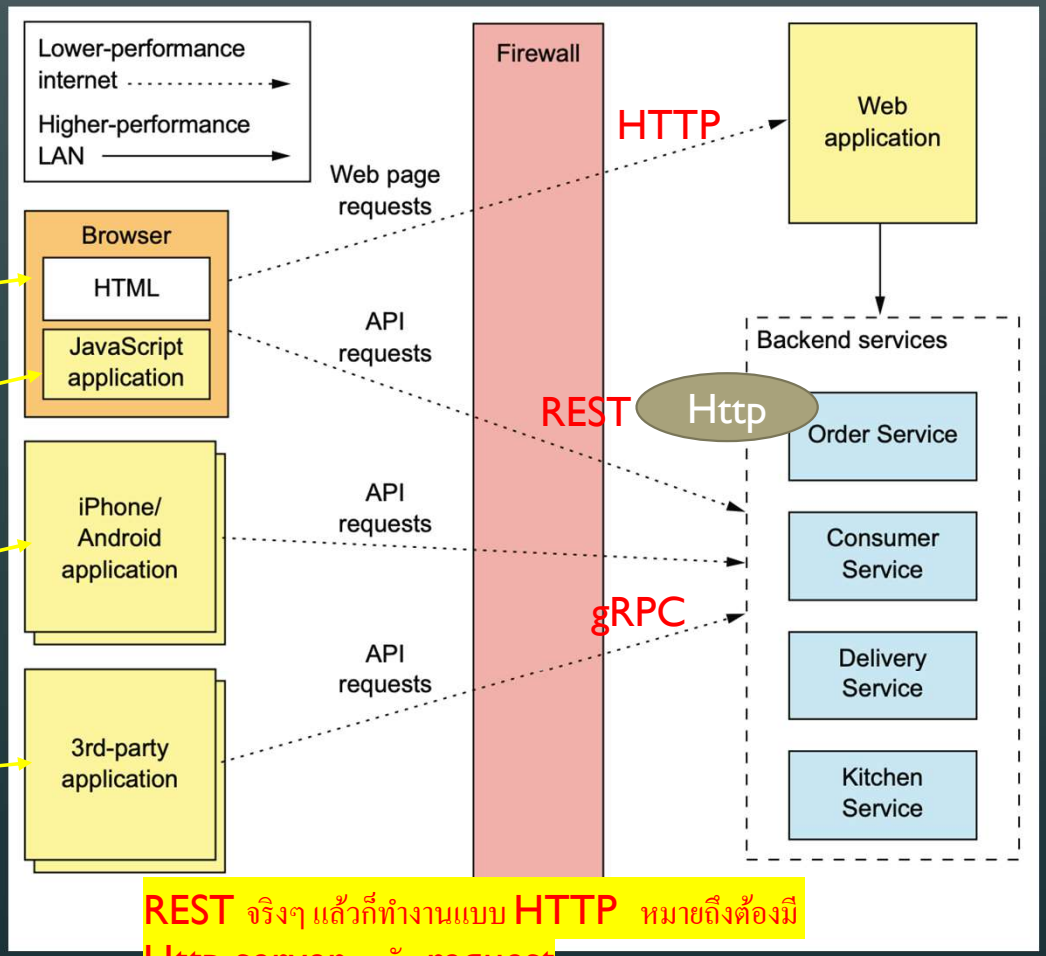






External API design issues (1)

- Let's consider the FTGO application.
- Four kinds of clients:
  1. Web applications
  2. JavaScript applications running in the browser
  3. Mobile applications
  4. Applications written by third-party developers

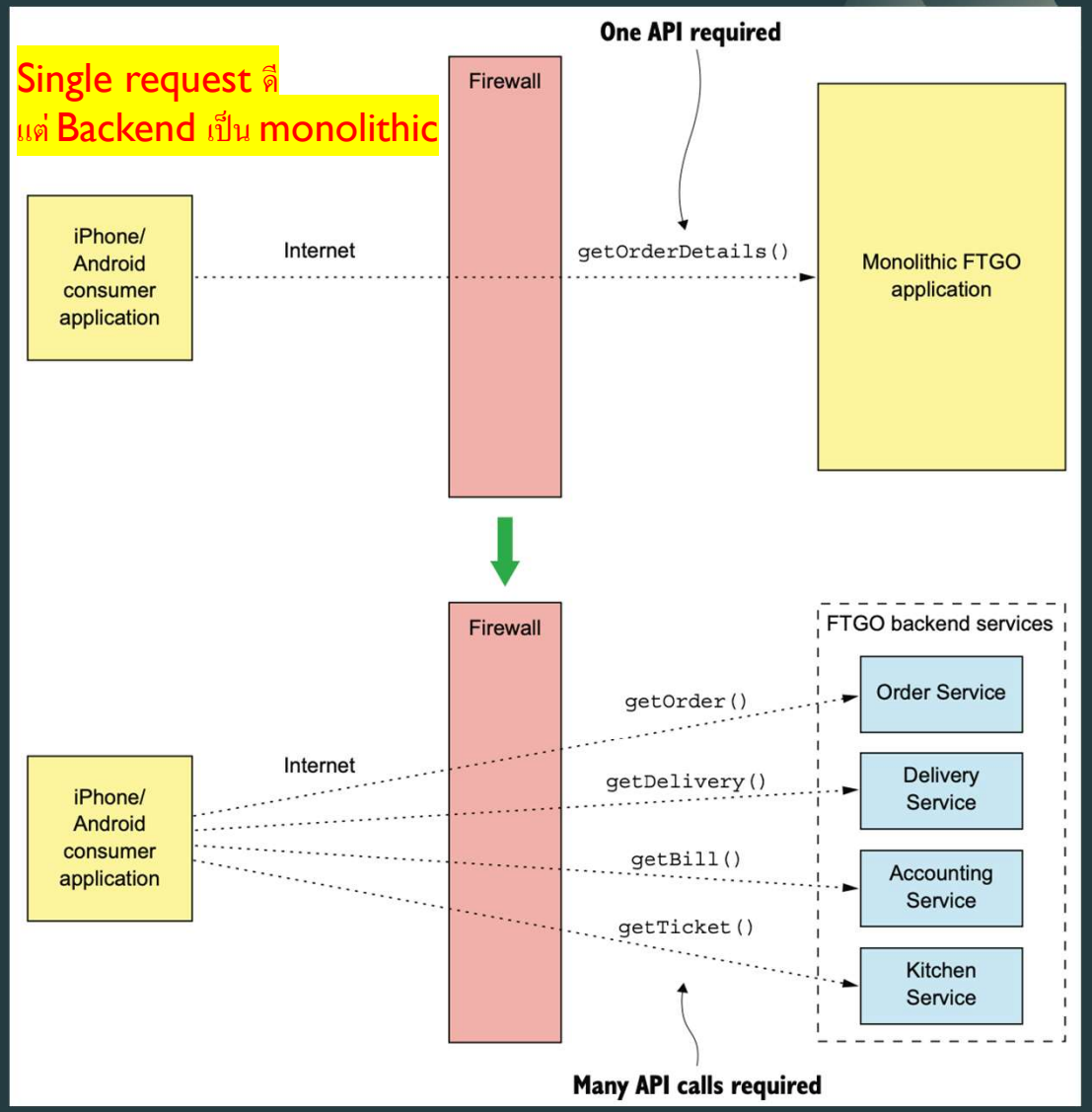


External API design issues (2)

- The other clients run outside the firewall.
  - → lower-bandwidth, higher-latency internet or mobile network
- 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.

API design issues for the FTGO mobile client (1)

- A client can retrieve the order details from the monolithic the FTGO application in a single request.
- But the client must make multiple requests to retrieve the same information in a microservice architecture.



API design issues for the FTGO mobile client (2)

- 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.

ปกติ Thin Client นอก firewall ใช้ HTTP สะดวกที่สุด การใช้ gRPC/AMQP message queue มักใช้ในส่วน Backends ติดต่อกันเอง เช่น เวลาใช้ gRPC ต้องมี proto file วุ่นวายกว่า และต้องใช้ binary mode ใน http2

API design issues for the other kind of client (1)

- Issues for the web application
  - Run within the firewall and access the services over a LAN → no problem
- Issues for the browser-based JavaScript application
  - The same problems with network latency as mobile applications.
- Issues for 3rd-party application
  - 3rd-party developers face the unstable API risk.
  - Typically have to maintain older versions for a long time—possibly forever.

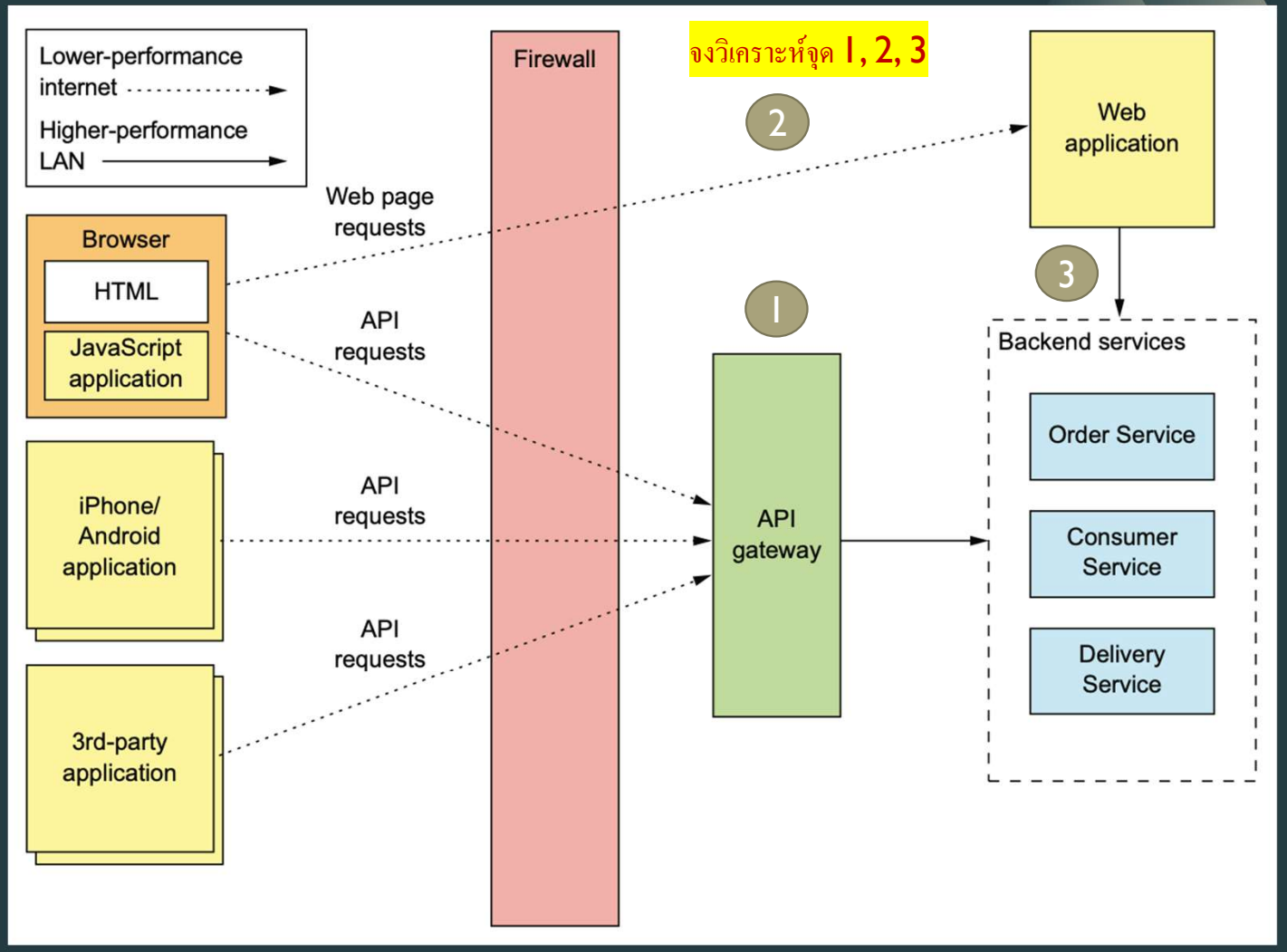
Backend จะเปลี่ยน API หรือเปลี่ยน endpoint address จะกระทบ 3rd party dev เสมอ แต่ถ้าเรามีตัวกลางมาช่วยคือ API gateway จะเสถียรกว่า

Agenda

- External API design issues
- **The API gateway pattern**
- Implementing an API gateway

Overview of the API gateway pattern

- The API gateway is the single-entry point into the application for API calls from outside the firewall.
- API gateway encapsulates the application's internal architecture and provides an API to its clients.



Encapsulate หมายถึงซ่อน Backends Implementation ไม่ให้ภายนอกเห็น หรือควบคุมการเข้าถึงได้ ถ้าเราเปลี่ยน Backends จะไม่กระทบ Client

API gateway responsibility 1. Request routing

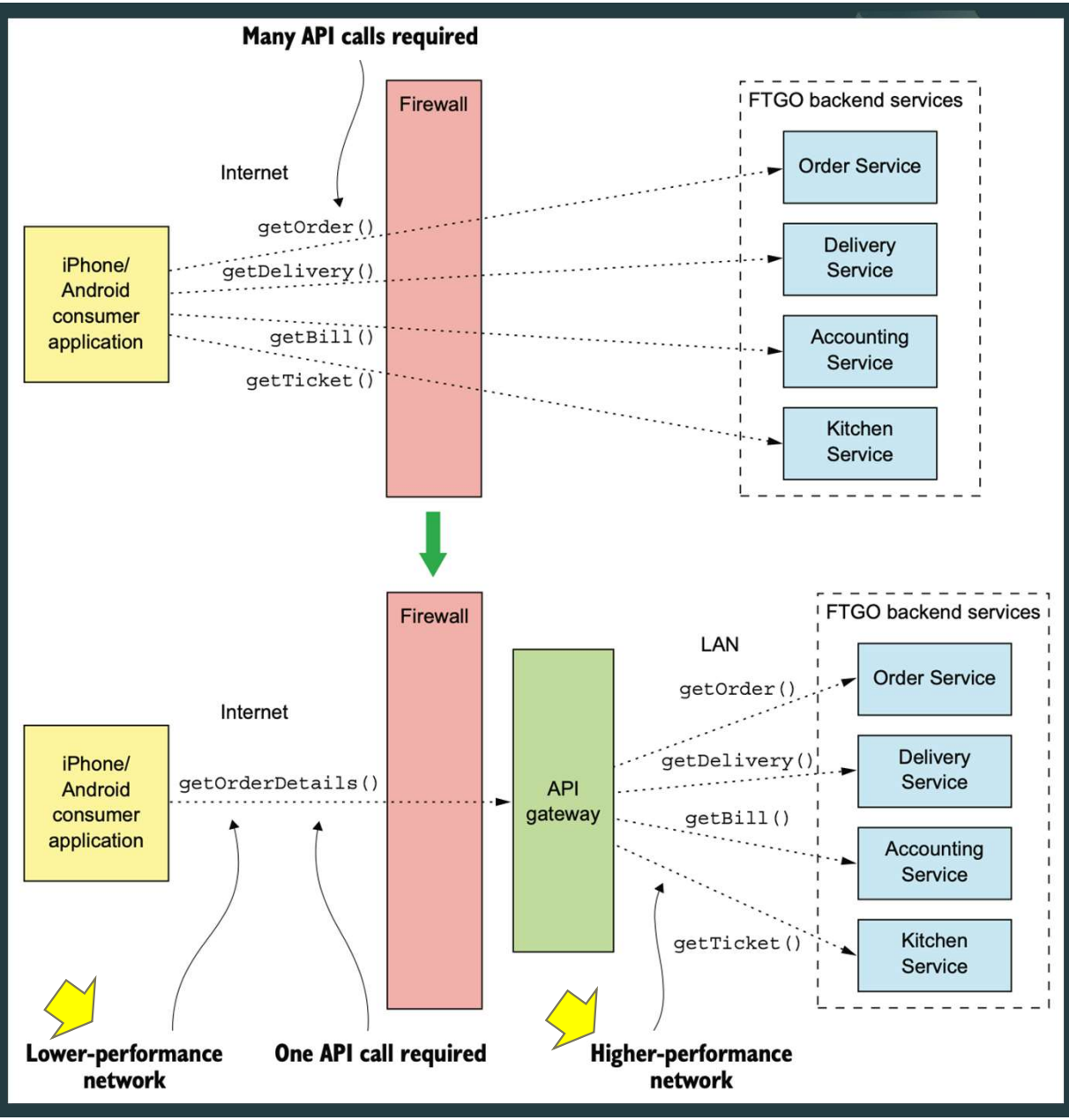
- An API gateway implements some API operations by routing requests to the corresponding service.
- When receiving a request, it consults *a routing map* that specifies which service to route the request to.
- *A routing map* map an HTTP method and path to the HTTP URL of a service.
- This function is identical to the reverse proxying features provided by web servers such as NGINX.

Routing เกี่ยวข้องกับเส้นทางการเข้าถึง Client จะเห็น route ไปยัง API gateway ที่เดียวไม่ต้องจำที่อื่นๆ โดย API gateway จะส่งต่อให้เอง

API gateway responsibility
2. API composition

- API composition enables a client to efficiently retrieve data using a single API request.
- For example, the mobile client makes a single *getOrderDetails()* request to the API gateway

Single request ดี และ API gateway จะทำหน้าที่เข้าถึง API อีกมากมายแทนและรวบรวมส่งคำตอบมาให้ เรียกว่า ทำ data composition ให้ด้วย higher performance network



API gateway responsibility 3. Protocol translation

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

แปลงจาก
REST←→ gRPC

เราอาจจะกำหนดว่า Client ส่งมาแบบ REST ทั้งหมด แล้ว API gateway จะจัดการเรียก Backend API ให้ไม่ว่าจะเป็น REST หรือ gRPC หรือ อื่น ๆ

Single one-size-fits-all (OSFA)

- 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.

Implementing edge functions

- An *edge function* is a request-processing function implemented at the edge of an application.
  - Authentication
  - Authorization
  - Rate limiting — Limiting how many requests per second from either a specific client and/or from all clients.
  - Caching—Cache responses to reduce the number of requests made to the services.
  - Metrics collection—Collect metrics on API usage for billing analytics purposes.
  - 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.

Option 1) Implement in the backend service

- 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.

Option 2) Implement in an edge service

- 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.

Option 3) Implement in an API gateway

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

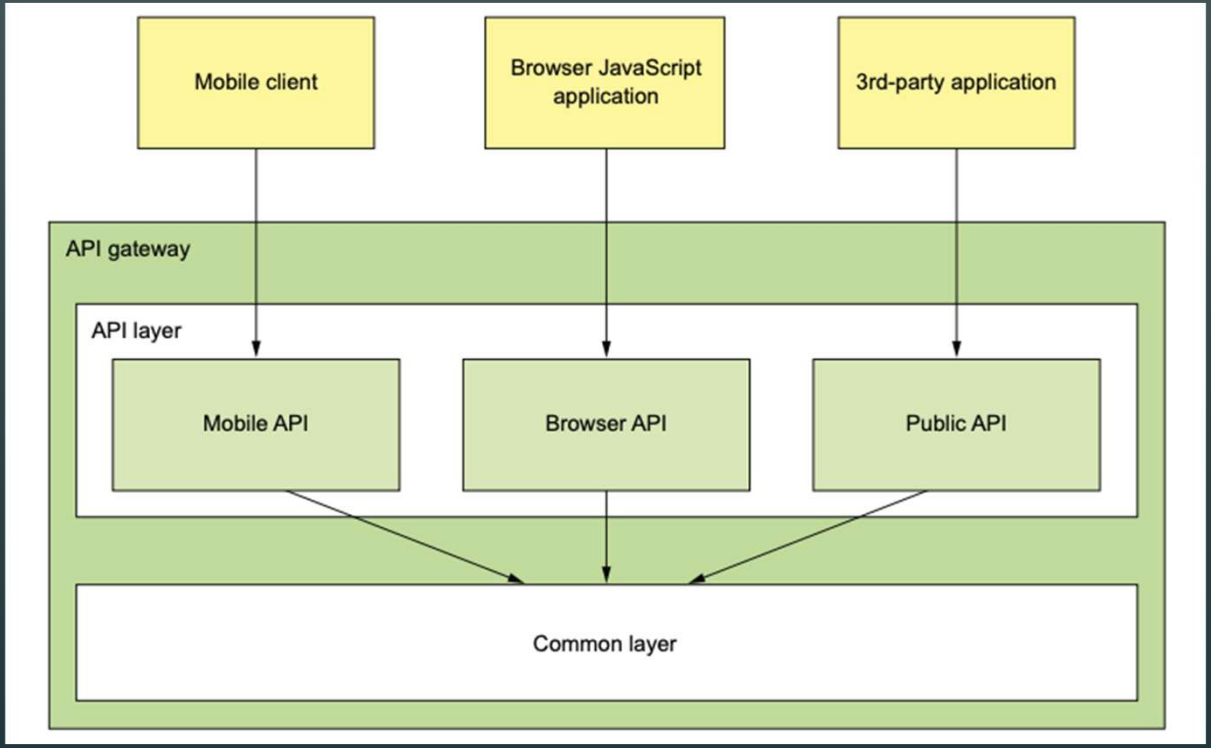
คำถาม

- Edge functions มีอะไรบ้าง
- Edge functions ควร Implemented อยู่ที่ไหน

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.



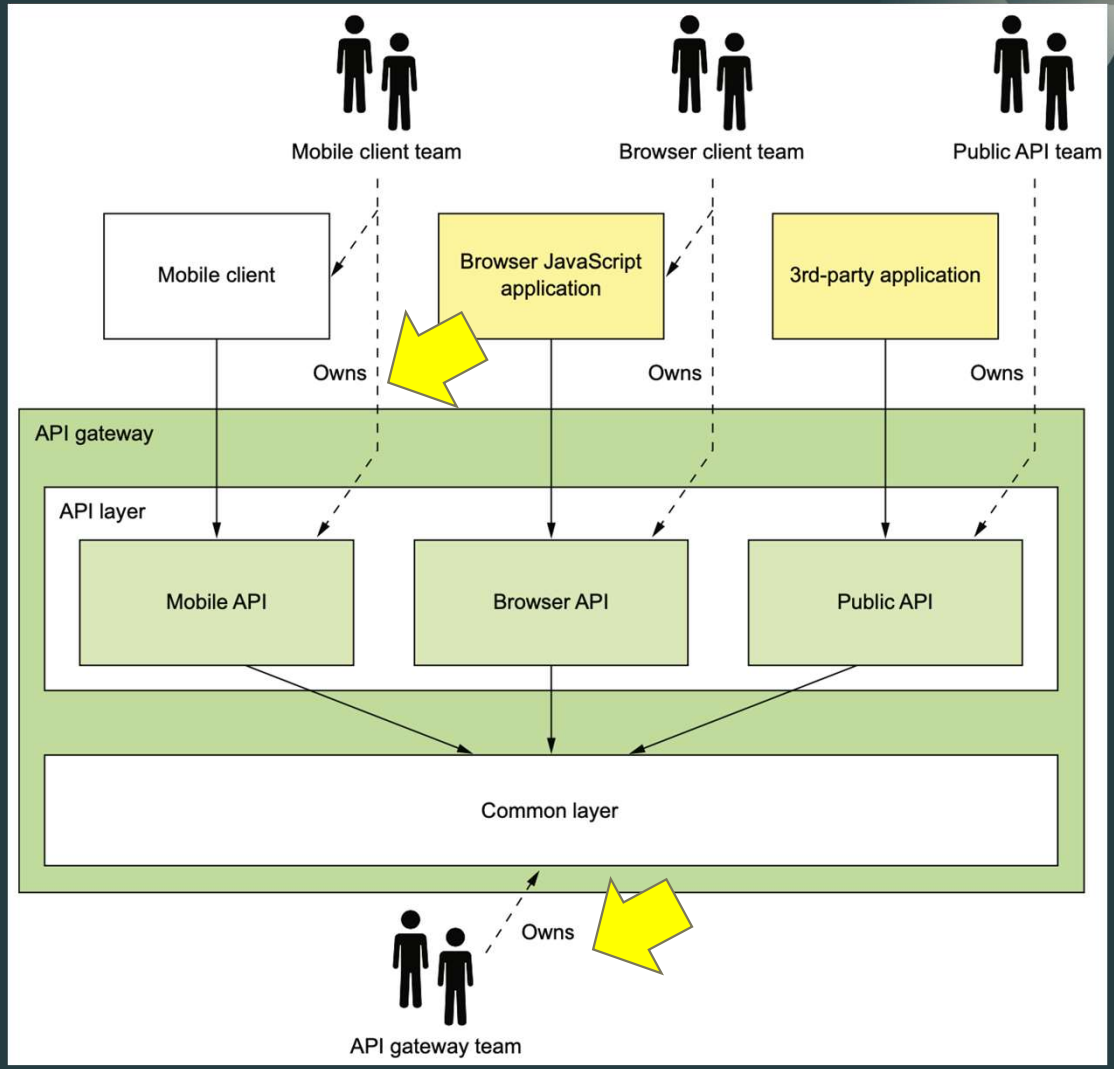
API gateway ownership model (1)

- Who is responsible for the development of the API gateway and its operation?
- Option 1) Separate team to be responsible for the API gateway.
- Drawback : Centralized bottleneck in the organization
  - For example, if a developer working on the mobile application needs access to a particular service, they must submit a request to the API gateway team and wait for them to expose the API.

API Gateway team รับเหมาทุกอย่างทำแบบ Centralized responsibility

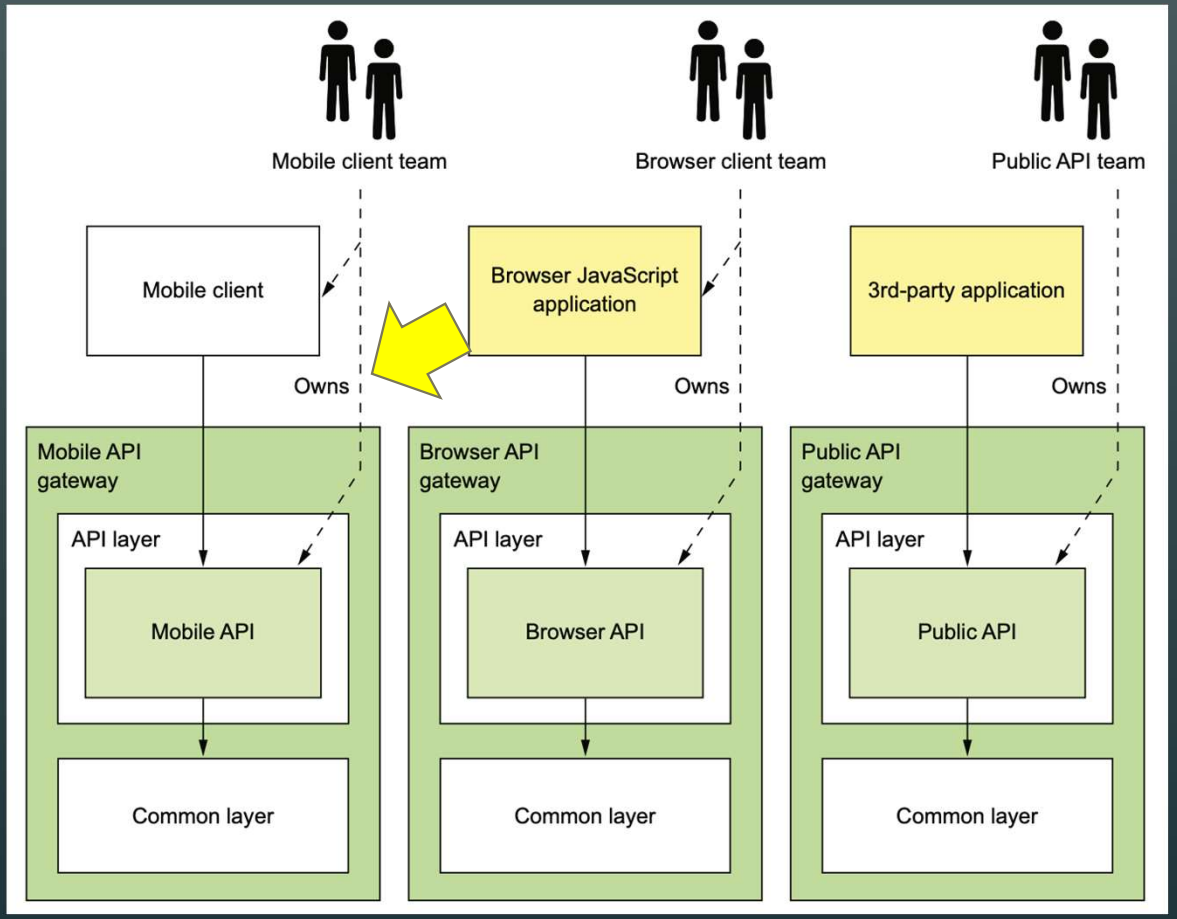
API gateway ownership model (2)

- Option 2) The client teams own the API module and API team own the common module.
  - Promoted by Netflix
- When the client team make changes at API layer, no need to ask the API team.



API gateway ownership model (3)

- Option 3) Using the Backends for Frontend pattern
- Defines a separate API gateway for each client.
  - Each client team owns their API gateway and common layer.



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

API gateway มักจะมีจำนวนมาก ถ้าเราใช้ Backend for Frontend pattern

- 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 ทำน้อย ๆ อย่ามี logic เยอะ รอแก้นาน ทำงานช้าเป็นคอขวด

Netflix as an example of an API gateway (1)

skip

- In the first version of the API gateway
  - Each client team implemented their API using Groovy scripts.
  - Each script invoked one or more service APIs using Java client libraries provided by the service teams.
  - The Netflix API gateway handles billions of requests per day.
  - On average each API call fans out to six or seven backend services.
  - Netflix has found this monolithic architecture to be somewhat cumbersome.

Netflix as an example of an API gateway (2) skip

- Netflix is now moving to an API gateway architecture similar to the Backends for frontends pattern.
  - Client teams write API modules using NodeJS.
  - Each API module runs its own Docker container, but the scripts don't invoke the services directly.
  - They invoke a second "API gateway," which exposes the service APIs using **Netflix Falcor**, an API technology that does declarative, dynamic API composition and enables a client to invoke multiple
  - The API modules are isolated from one another, which improves reliability and observability

API gateway design issues.

- There are several issues to consider when designing an API gateway:
  - Performance and scalability
  - Writing maintainable code by using reactive programming abstractions
  - Handling partial failure
  - Being a good citizen in the application's architecture

Performance and scalability (1)

- A key design decision that affects performance and scalability is whether the API gateway should use synchronous or asynchronous I/O.
- For synchronous I/O model:
  - Each network connection is handled by a 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 จะเป็น Heavyweight เพราะต้อง dedicated thread จองไว้เลย ใช้ง่าย แต่มีจำนวน thread ให้น้อย

Performance and scalability (2)

- For asynchronous (nonblocking) I/O model:
  - 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.

Use reactive programming abstractions (1)

- Consider an API endpoint handler method that call the services in the order.
- Drawback is the response time is the sum of the service response times.
- To minimize response time, the composition logic should invoke services **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.

Use reactive programming abstractions (2)

- The traditional way to write scalable, concurrent code is using **callbacks**.
  - Execute requests concurrently by calling *ExecutorService.submitCallable()*.
  - 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.
  - Leads you to callback hell.

Use reactive programming abstractions (3)

- A much better approach is to write API composition code in a declarative style using a reactive approach.
- Examples of reactive abstractions for the JVM include the following:
  - Java 8 CompletableFutures
  - Project Reactor Monos
  - RxJava (Reactive Extensions for Java) by Netflix
  - ScalaFutures
- Enable you to write concurrent code that's simple and easy to understand

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;
}
...
```

ใช้ Reactive approach ในการระบุให้เกิด Concurrently invoking

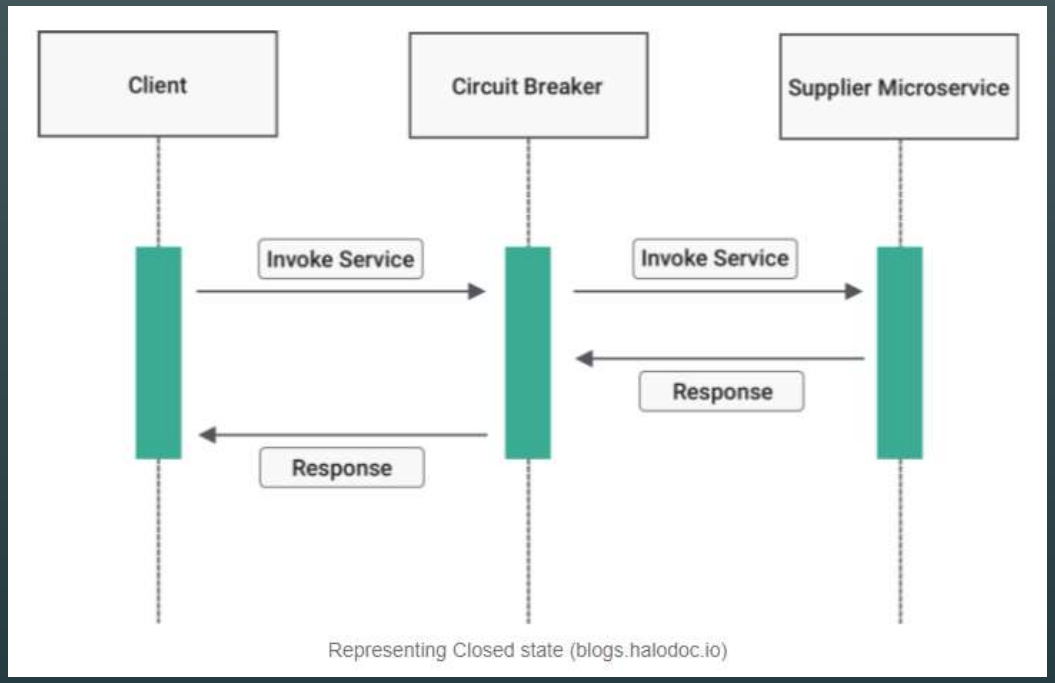
Reactive declared var.

Handling 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.

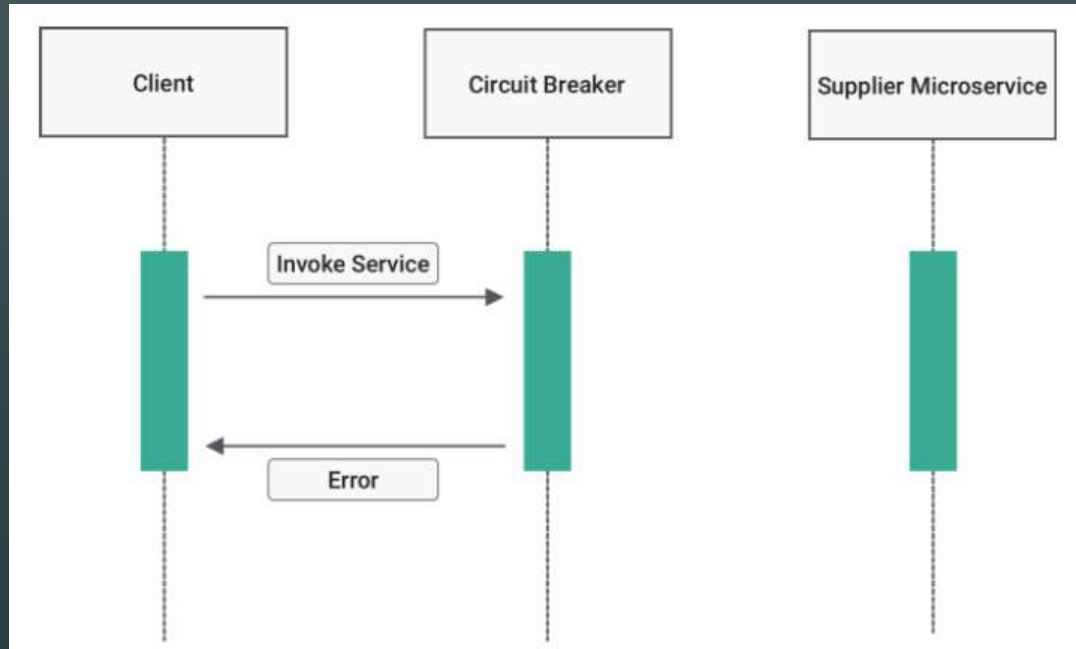
Circuit breaker pattern ทำอย่างไร
มี 3 states – closed, open, half-open state

Closed state



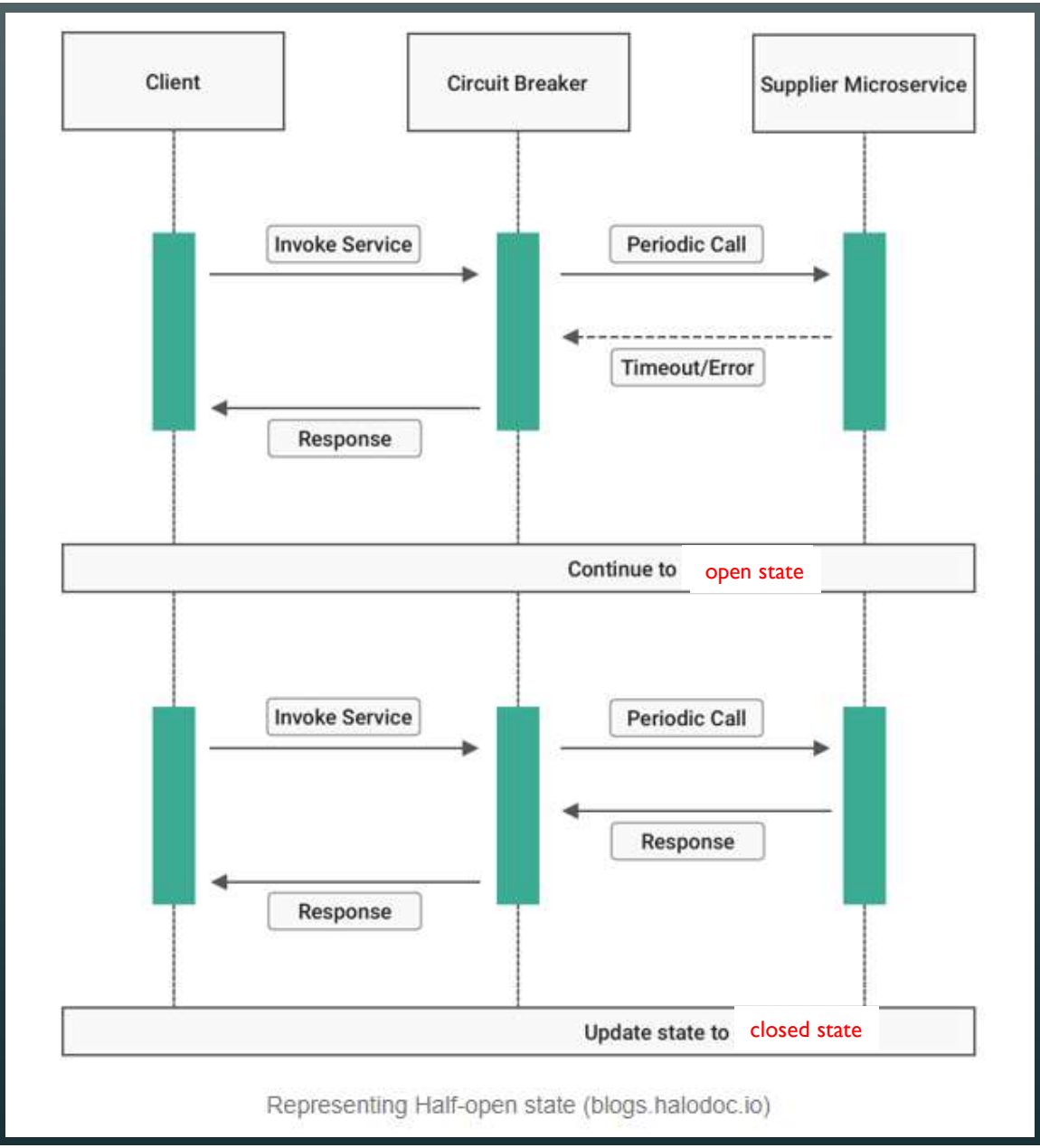
Representing Closed state (blogs.halodoc.io)

Open state



Representing Open state (blogs.halodoc.io)

Half-open state



Representing Half-open state (blogs.halodoc.io)

Being a good citizen in the architecture

เป็น เด็กดี

- An API gateway must implement the patterns that have been selected for the architecture.

Agenda

- External API design issues
- The API gateway pattern
- Implementing an API gateway

Implementing an API gateway

There are a couple of different ways to implement an API gateway:

- Using an off-the-shelf API gateway product/service
  - Requires little or no development but is the least flexible.
- Developing your own API gateway
  - Using either an API gateway framework or a web framework as the start.
  - Flexible approach, though it requires some development effort.

AWS API gateway (Amazon Web Services)

skip

- Provided a set of REST resources, each of which supports one or more HTTP methods.
- Configure the API gateway, and AWS handles everything else, including scaling.
- Drawbacks and limitations:
  - Not support API composition, need implementation in the backend services.
  - Only supports HTTP(S) with a heavy emphasis on JSON.
  - Only supports the Server-side discovery pattern.

AWS Application Load Balancer

- Another 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.

Using An API Gateway Product

- Such as Kong or 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

Developing your own API gateway

อันที่จริง API gateway คือ web app

- It is basically a web application that proxies requests to other services.
- Two key design problems that you'll need to solve:
  1. Implementing a mechanism for defining routing rules in order to minimize the complex coding
  2. Correctly implementing the HTTP proxying behavior, including how HTTP headers are handled
- A better starting point is to use a framework.
  - For example, Netflix Zuul, Spring Cloud Gateway

Using Netflix Zuul

skip

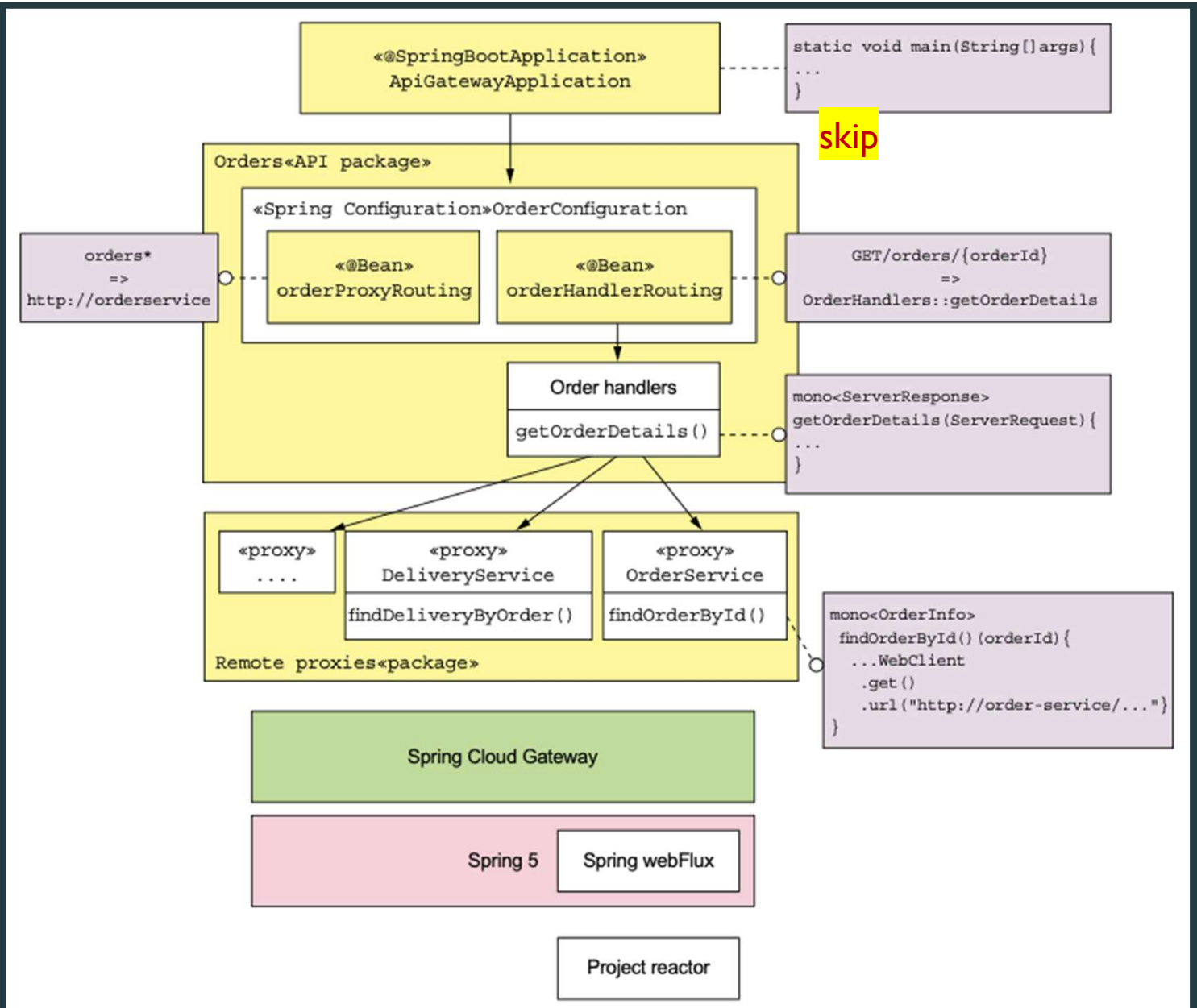
- Zuul framework to implement edge functions such as routing, rate limiting, and authentication (https://github.com/Netflix/zuul)
- Zuul handles an HTTP request by assembling a chain of applicable filters → transform the request → invoke backend services → transform the response → sent back to the client.
- You can use Zuul directly or use Spring Cloud Zuul from Pivotal (easier)
- A major limitation of Zuul is that it can only implement path-based routing.

Using Spring Cloud Gateway

skip

- https://cloud.spring.io/spring-cloud-gateway/
- Built on top of several frameworks, including:
  - Spring Framework 5
  - Spring Boot 2
  - Spring Webflux - a reactive web framework that's part of Spring Framework 5 and built on Project Reactor.
  - Project Reactor - an NIO-based reactive framework for the JVM that provides the Mono abstraction.

- Spring Cloud Gateway can do the following:
  - Route requests to backend services.
  - Implement request handlers that perform API composition.
  - Handle edge functions such as authentication.



Using Spring Cloud Gateway (cont.)

skip

- The API gateway consists of the following packages:
  - ApiGatewayMainpackage—Defines the Main program for the API gateway.
  - One or more API packages—An API package implements a set of API endpoints.
  - Proxy package—Consists of proxy classes that are used by the API packages to invoke the services.

skip

Listing 8.2 The Spring `@Beans` that implement the `/orders` endpoints

```
@Configuration
@EnableConfigurationProperties(OrderDestinations.class)
public class OrderConfiguration {

  @Bean
  public RouteLocator orderProxyRouting(OrderDestinations orderDestinations) {
    return Routes.locator()
            .route("orders")
            .uri(orderDestinations.orderServiceUrl)
            .predicate(path("/orders").or(path("/orders/*")))
             .and()
            ...
            .build();
  }

  @Bean
  public RouterFunction<ServerResponse>
            orderHandlerRouting(OrderHandlers orderHandlers) {
    return RouterFunctions.route(GET("/orders/{orderId}"),
                        orderHandlers::getOrderDetails);
  }

  @Bean
  public OrderHandlers orderHandlers(OrderService orderService,
                                     KitchenService kitchenService,
                                     DeliveryService deliveryService,
                                     AccountingService accountingService) {
    return new OrderHandlers(orderService, kitchenService,
                             deliveryService, accountingService);
  }

}
```

By default, route all requests whose path begins with /orders to the URL orderDestinations.orderServiceUrl.

Route a GET /orders/{orderId} to orderHandlers::getOrderDetails.

The @Bean, which implements the custom request-handling logic

`OrderDestinations`, shown in the following listing, is a Spring `@Configuration-Properties` class that enables the externalized configuration of backend service URLs.

skip

Listing 8.3 The externalized configuration of backend service URLs

```
@ConfigurationProperties(prefix = "order.destinations")
public class OrderDestinations {

  @NotNull
  public String orderServiceUrl;

  public String getOrderServiceUrl() {
    return orderServiceUrl;
  }

  public void setOrderServiceUrl(String orderServiceUrl) {
    this.orderServiceUrl = orderServiceUrl;
  }
  ...
}
```

Listing 8.4 The `OrderHandlers` class implements custom request-handling logic.

skip

```
public class OrderHandlers {

  private OrderService orderService;
  private KitchenService kitchenService;
  private DeliveryService deliveryService;
  private AccountingService accountingService;

  public OrderHandlers(OrderService orderService,
                       KitchenService kitchenService,
                       DeliveryService deliveryService,
                       AccountingService accountingService) {
    this.orderService = orderService;
    this.kitchenService = kitchenService;
    this.deliveryService = deliveryService;
    this.accountingService = accountingService;
  }

  public Mono<ServerResponse> getOrderDetails(ServerRequest serverRequest) {
    String orderId = serverRequest.pathVariable("orderId");

    Mono<OrderInfo> orderInfo = orderService.findOrderById(orderId);

    Mono<Optional<TicketInfo>> ticketInfo =
       kitchenService
            .findTicketByOrderId(orderId)
            .map(Optional::of)
             .onErrorReturn(Optional.empty());
```

Transform a TicketInfo into an Optional<TicketInfo>.

If the service invocation failed, return Optional.empty().

skip

```
    Mono<Optional<DeliveryInfo>> deliveryInfo =
        deliveryService
            .findDeliveryByOrderId(orderId)
            .map(Optional::of)
            .onErrorReturn(Optional.empty());

    Mono<Optional<BillInfo>> billInfo = accountingService
            .findBillByOrderId(orderId)
            .map(Optional::of)
            .onErrorReturn(Optional.empty());

    Mono<Tuple4<OrderInfo, Optional<TicketInfo>,
                Optional<DeliveryInfo>, Optional<BillInfo>>> combined =
            Mono.when(orderInfo, ticketInfo, deliveryInfo, billInfo);

    Mono<OrderDetails> orderDetails =
         combined.map(OrderDetails::makeOrderDetails);

    return orderDetails.flatMap(person -> ServerResponse.ok()
             .contentType(MediaType.APPLICATION_JSON)
            .body(fromObject(person)));
  }

}
```

Combine the four values into a single value, a Tuple4.

Transform the Tuple4 into an OrderDetails.

Transform the OrderDetails into a ServerResponse.

skip

Listing 8.5 OrderService class—a remote proxy for Order Service

```
@Service
public class OrderService {

  private OrderDestinations orderDestinations;

  private WebClient client;

  public OrderService(OrderDestinations orderDestinations, WebClient client)
     {
    this.orderDestinations = orderDestinations;
    this.client = client;
  }

  public Mono<OrderInfo> findOrderById(String orderId) {
    Mono<ClientResponse> response = client
            .get()
            .uri(orderDestinations.orderServiceUrl + "/orders/{orderId}",
                 orderId)
            .exchange();
    return response.flatMap(resp -> resp.bodyToMono(OrderInfo.class));
  }

}
```

Invoke the service.

Convert the response body to an OrderInfo.

skip

Listing 8.6 The `main()` method for the API gateway

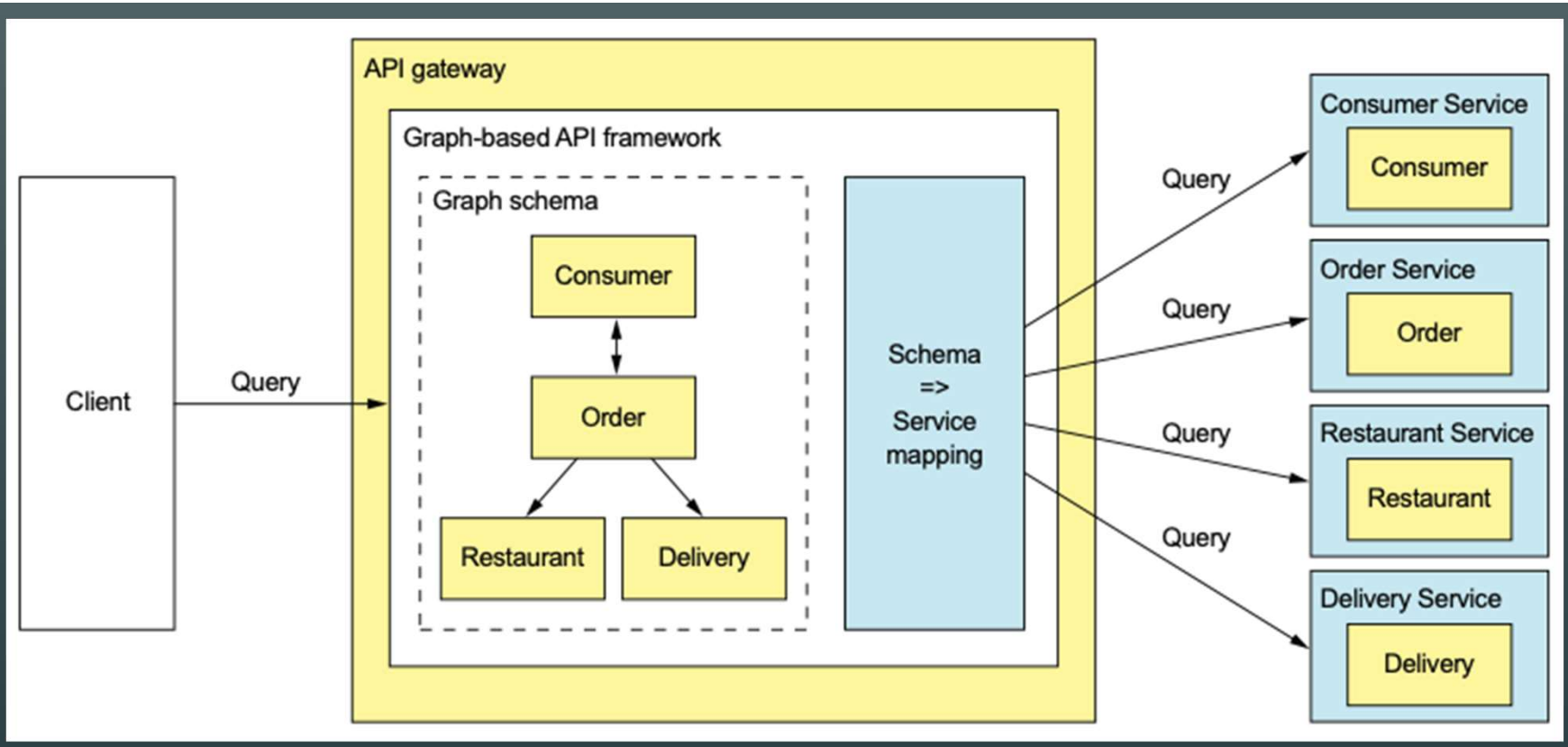
```
@SpringBootConfiguration
@EnableAutoConfiguration
@EnableGateway
@Import(OrdersConfiguration.class)
public class ApiGatewayApplication {

  public static void main(String[] args) {
    SpringApplication.run(ApiGatewayApplication.class, args);
  }
}
```

Implementing an API gateway using GraphQL

skip

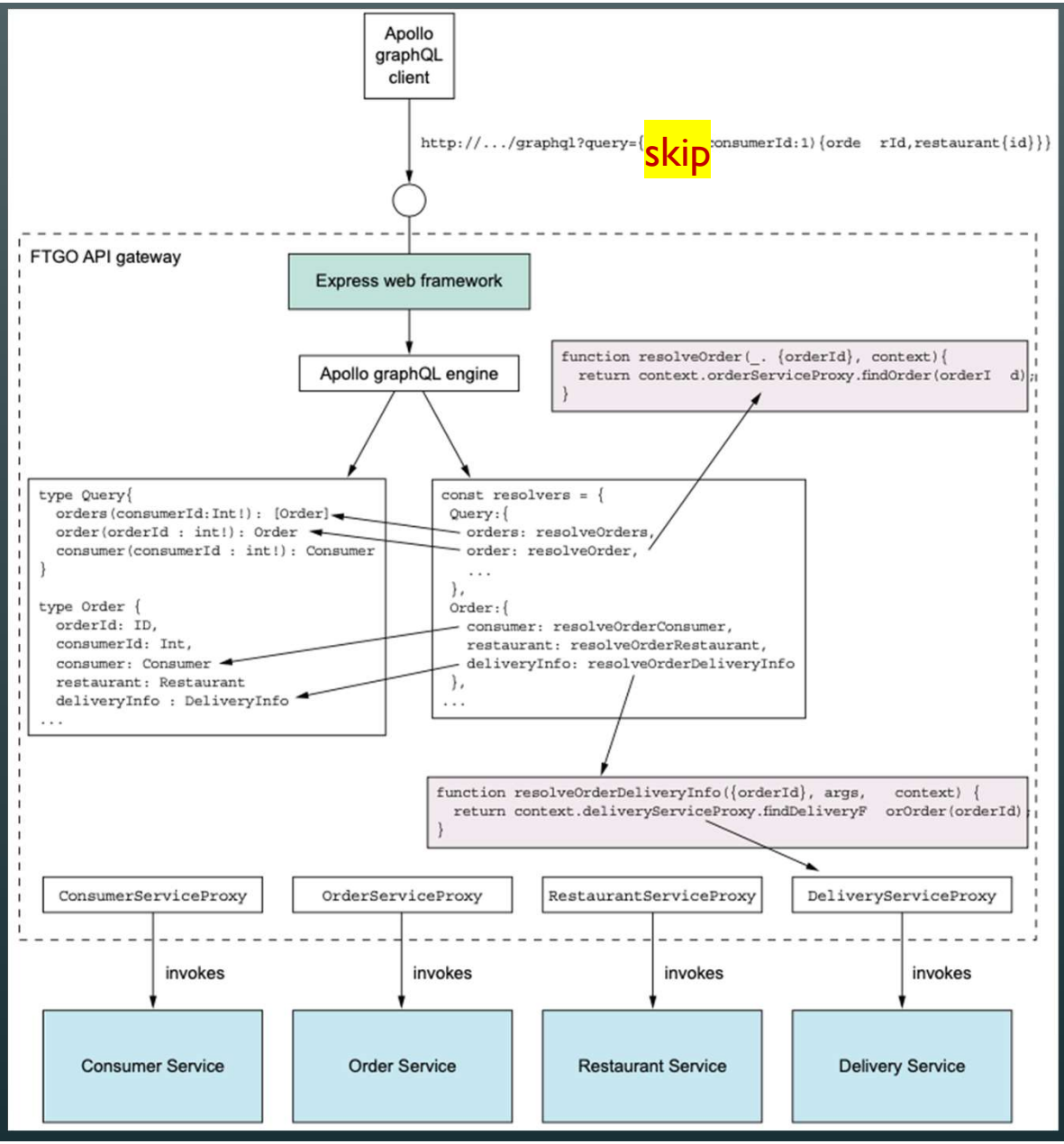
- Imagine that you're responsible for implementing the FTGO's API Gateway's GET /orders/{orderId} endpoint.
  - This endpoint retrieves data from multiple services.
  - You need to use the API composition pattern and write code that invokes the services and combines the results.
- Different clients need slightly different data.
- Implementing with a REST API is time consuming.
- you may consider using a graph-based API framework, such as GraphQL



skip

- The API gateway's API consists of a graph-based schema that's mapped to the services.
- A client issues a query that retrieves multiple graph nodes.
- The graph-based API framework executes the query by retrieving data from one or more services.

- The GraphQL-based API gateway
  - JavaScript using the NodeJS Express web framework
  - Apollo GraphQL server
- The key parts of the design are:
  - GraphQL schema—defines the server-side data model and the queries it supports.
  - Resolver functions—map elements of the schema to the various backend services.
  - Proxy classes—invoke the FTGO application's services.



Listing 8.7 The GraphQL schema for the FTGO API gateway

```
type Query {
  orders(consumerId : Int!): [Order]
  order(orderId : Int!): Order
  consumer(consumerId : Int!): Consumer
}

type Consumer {
  id: ID
  firstName: String
  lastName: String
  orders: [Order]
 }

type Order {
  orderId: ID,
  consumerId : Int,
  consumer: Consumer
  restaurant: Restaurant

  deliveryInfo : DeliveryInfo

  ...
}

type Restaurant {
  id: ID
  name: String
  ...
}
type DeliveryInfo {
  status : DeliveryStatus
  estimatedDeliveryTime : Int
  assignedCourier :String
}

enum DeliveryStatus {
  PREPARING
  READY_FOR_PICKUP
  PICKED_UP
  DELIVERED
}
```

Defines the queries that a client can execute

The unique ID for a Consumer

A consumer has a list of orders.

skip

Listing 8.8 Attaching the resolver functions to fields of the GraphQL schema

```
const resolvers = {
  Query: {
    orders: resolveOrders,
    consumer: resolveConsumer,
    order: resolveOrder
  },
  Order: {
    consumer: resolveOrderConsumer,
    restaurant: resolveOrderRestaurant,
    deliveryInfo: resolveOrderDeliveryInfo
...
};
```

The resolver for the orders query

The resolver for the consumer field of an Order

skip

Listing 8.9 Using a `DataLoader` to optimize calls to `Restaurant Service`

```
const DataLoader = require('dataloader');

class RestaurantServiceProxy {
    constructor() {
        this.dataLoader =
            new DataLoader(restaurantIds =>
             this.batchFindRestaurants(restaurantIds));
    }

    findRestaurant(restaurantId) {
         return this.dataLoader.load(restaurantId);
    }

    batchFindRestaurants(restaurantIds) {
       ...
    }
}
```

Create a DataLoader, which uses batchFindRestaurants() as the batch loading functions.

Load the specified Restaurant via the DataLoader.

Load a batch of Restaurants.

skip

Listing 8.10 Integrating the GraphQL server with the Express web framework

```
const {graphqlExpress} = require("apollo-server-express");

const typeDefs = gql`                          <- Define the GraphQL schema.
   type Query {
    orders: resolveOrders,
   ...
  }

  type Consumer {
   ...

const resolvers = {                            <- Define the resolvers.
   Query: {
  ...
  }
}

const schema = makeExecutableSchema({ typeDefs, resolvers });   <- Combine the schema with the resolvers to create an executable schema.

const app = express();

function makeContextWithDependencies(req) {     <- Inject repositories into the context so they're available to resolvers.
    const orderServiceProxy = new OrderServiceProxy();
    const consumerServiceProxy = new ConsumerServiceProxy();
    const restaurantServiceProxy = new RestaurantServiceProxy();
    ...
    return {orderServiceProxy, consumerServiceProxy,
              restaurantServiceProxy, ...};
}

function makeGraphQLHandler() {                 <- Make an express request handler that executes GraphQL queries against the executable schema.
     return graphqlExpress(req => {
        return {schema: schema, context: makeContextWithDependencies(req)}
    });
}

app.post('/graphql', bodyParser.json(), makeGraphQLHandler());   <- Route POST /graphql and GET /graphql endpoints to the GraphQL server.

app.get('/graphql', makeGraphQLHandler());

app.listen(PORT);
```

skip

Listing 8.11 Using the Apollo GraphQL client to execute queries

```
class FtgoGraphQLClient {

    constructor(...) {
        this.client = new ApolloClient({ ... });
    }

    findConsumer(consumerId) {
        return this.client.query({
            variables: { cid: consumerId},
             query: gql`
              query foo($cid : Int!) {
                 consumer(consumerId: $cid)  {
                     id
                    firstName
                    lastName
                }
            } `,
        })
    }

}
```

Supply the value of the $cid.

Define $cid as a variable of type Int.

Set the value of query parameter consumerid to $cid.

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.

Summary (2)

- Spring Cloud Gateway is a good, easy-to-use framework for developing an API gateway.
  - Spring Cloud Gateway can route a request either directly to a backend service or to a custom handler method.
  - It's built using the scalable, reactive Spring Framework 5 and Project Reactor frameworks.
  - You can write your custom request handlers in a reactive style.
- GraphQL is another excellent foundation for developing an API Gateway for supporting diverse clients.