External APIs and Deployment Patterns

Owerya
RESOURCING

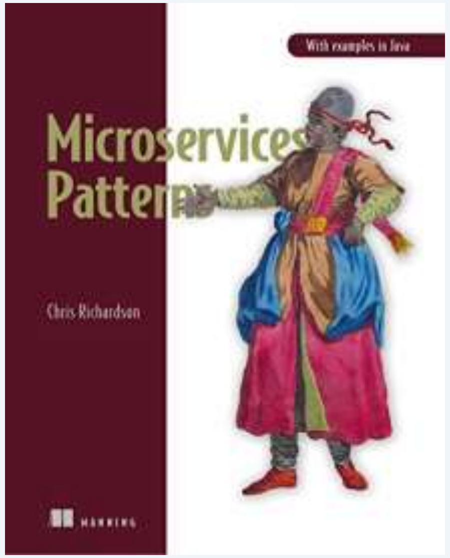
Objectives

- External API Patterns

  Issues with External APIs for microservices
  The API Gateway and Backend for Frontend Patterns

- Service Deployment Patterns

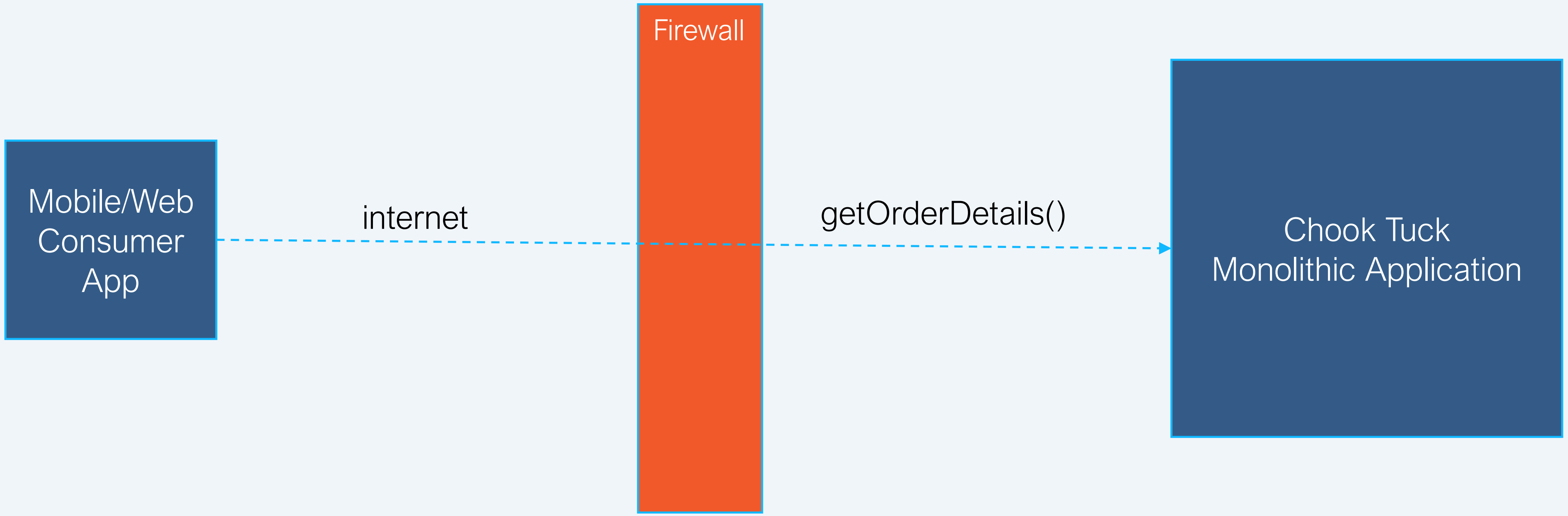
  Security - Access Token Pattern
  Externalized Configuration Pattern
  Observability Patterns
  Microservice Chassis Pattern
  Service Mesh



External API Patterns

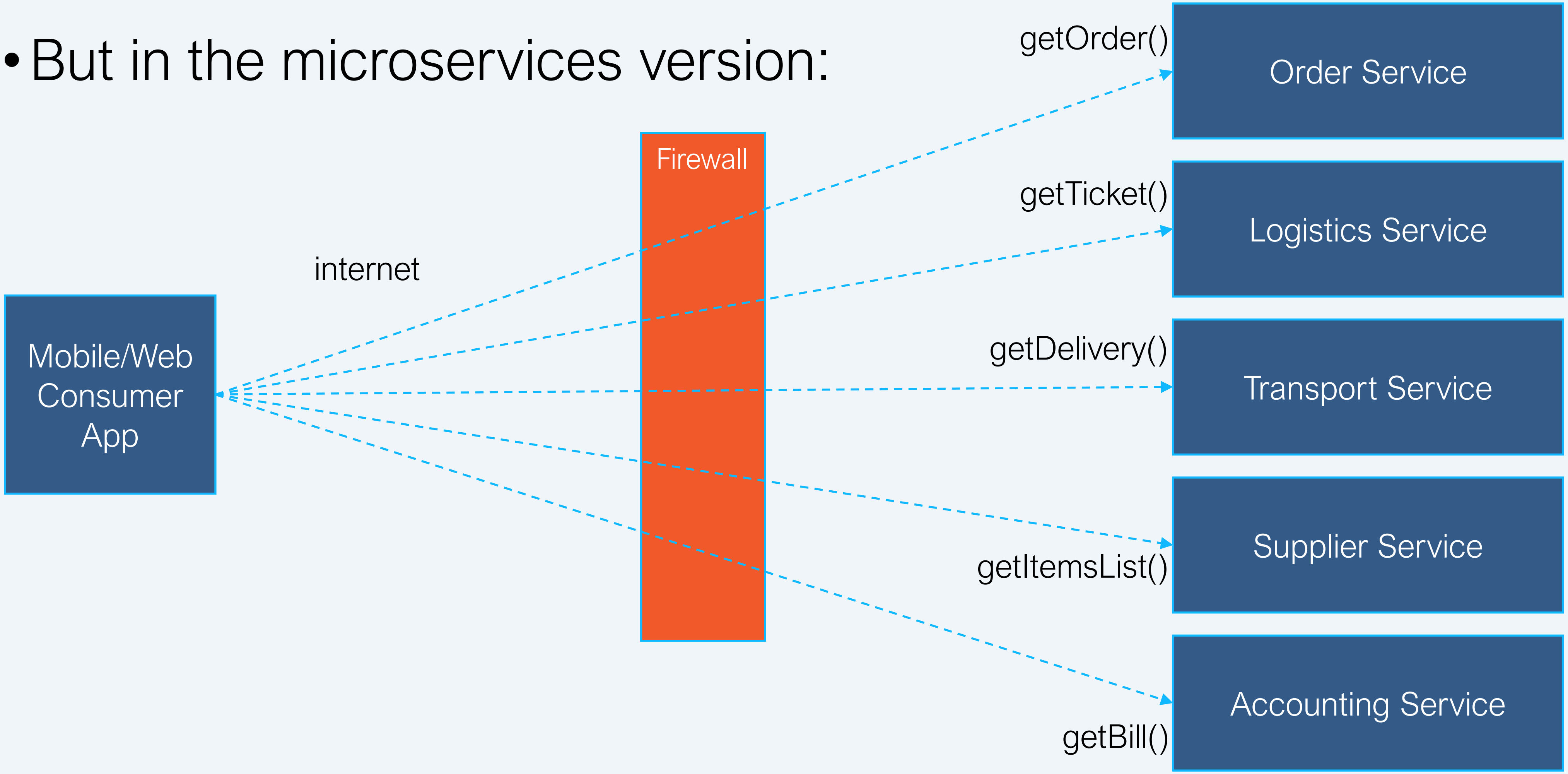
External APIs for Microservices

- Consider the monolithic Chook Tuck application
- A query for order details requires a single API call from the consumer application to the backend services



External API call in a microservice

- But in the microservices version:



Owerya RESOURCING

Problem and Forces

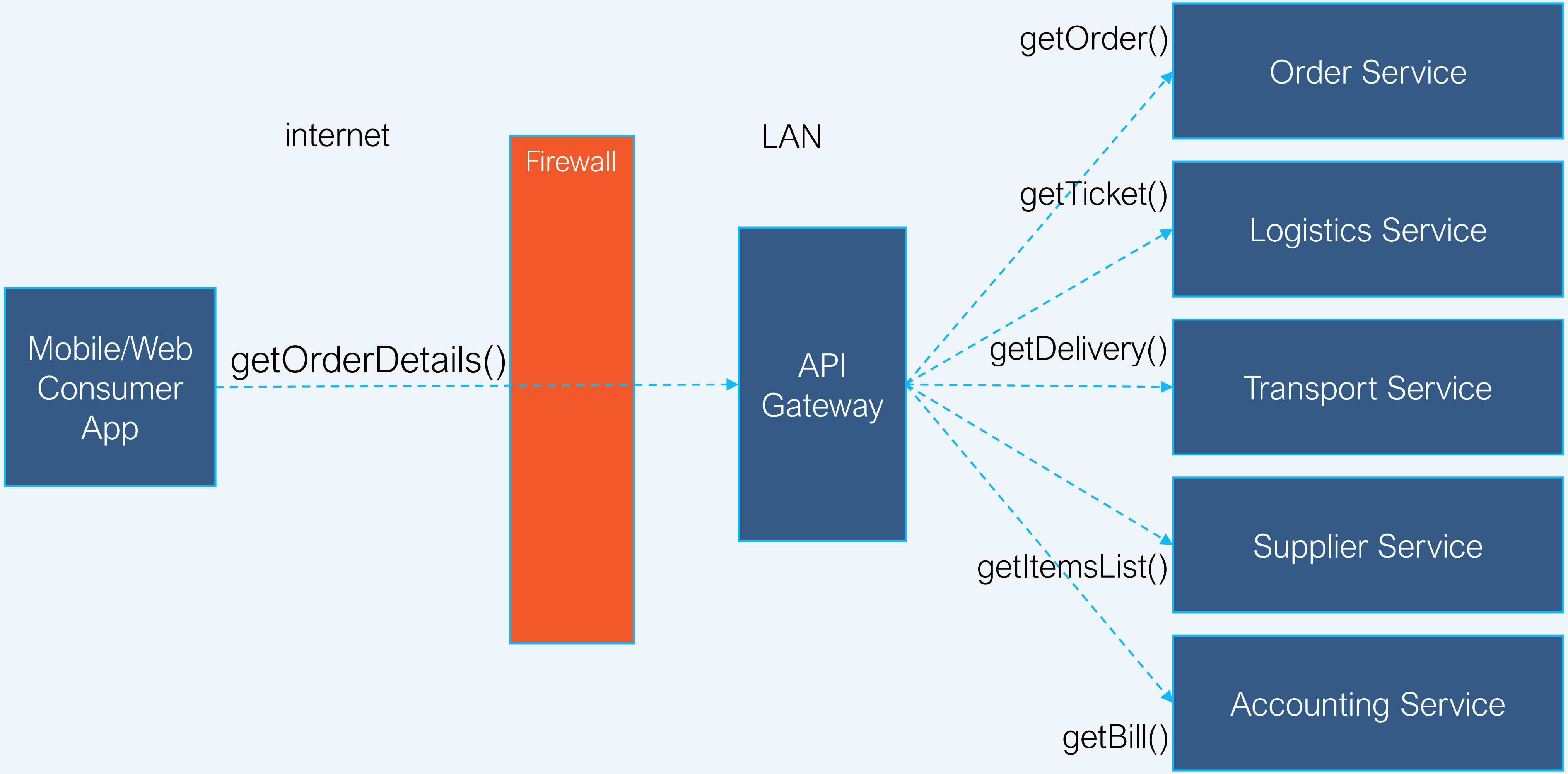
- Problem
  - How do the clients of a microservices-based application access the individual services?

- Forces
  - The granularity of the microservices APIs are often too fine-grained for what the client needs.
  - Different clients need different data (e.g. desktop browser and mobile version)
  - Different clients have different network performance
  - The number of service instances and their host+port configuration changes dynamically
  - Partitioning into services can change over time and should be hidden from clients
  - Services might use a diverse set of protocols (gRPC, REST, etc), some of which might not be web/firewall friendly.

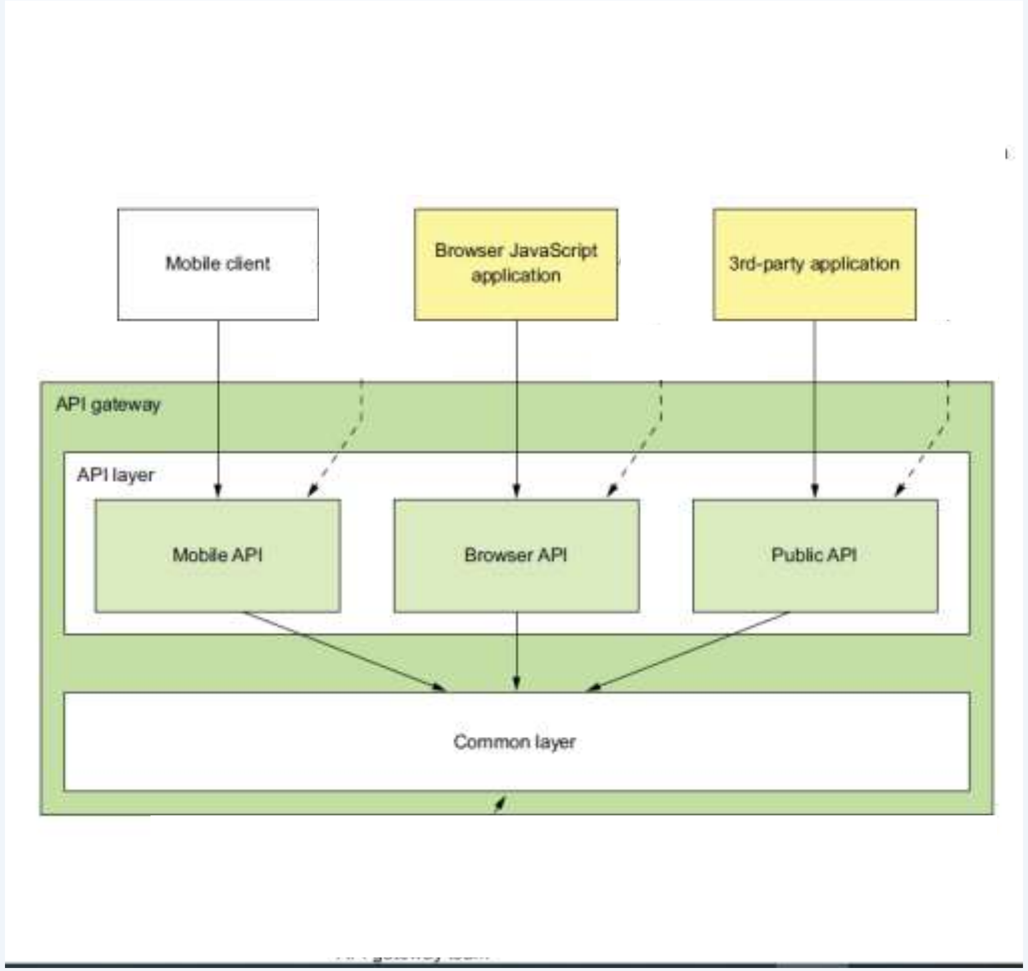
Solution 1: API Gateway Pattern

- Implement an API Gateway that is a single point of entry for all clients.
  - Some requests are just passed onto the appropriate service
  - Others are handled by conscripting multiple services and combining the results using the API Composer pattern.
  - Can supply each client with a client-specific API
  - Can implement *edge functions*
    - Authentication
    - Authorization
    - Rate Limiting
    - Caching – cache responses to reduce number of API calls
    - Metrics Collection – for billing and analytics purposes
    - Requests Logging

API Gateway


internet
Firewall
LAN
Mobile/Web Consumer App
getOrderDetails()
API Gateway
getOrder()
Order Service
getTicket()
Logistics Service
getDelivery()
Transport Service
Supplier Service
getItemsList()
Accounting Service
getBill()

API Gateway Architecture



- Layered, modular architecture
- Common layer implements functionality common to all APIs.
- Depending on your team structure, you can implement this by determining ownership
  - A team for each API and the API Gateway team for the common layer.
- The Backend from Frontend pattern is very similar, it specifically mandates the APIs to be owned by each individual client team and can include splitting the Common Layer.

Resulting Context: Benefits

- Insulates the clients from how the application is partitioned into microservices
- Insulates the clients from the problem of determining the locations of service instances
- Provides the optimal API for each client
- Reduces the number of requests/roundtrips.
- Simplifies the client by moving logic for calling multiple services from the client to API gateway
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally

Resulting Context: Drawbacks and Issues

- Drawbacks
  - Increased complexity - another moving part to be developed, deployed and managed
  - Increased response time due to the additional network hop through the API gateway
    - For most applications the cost of an extra roundtrip is insignificant.

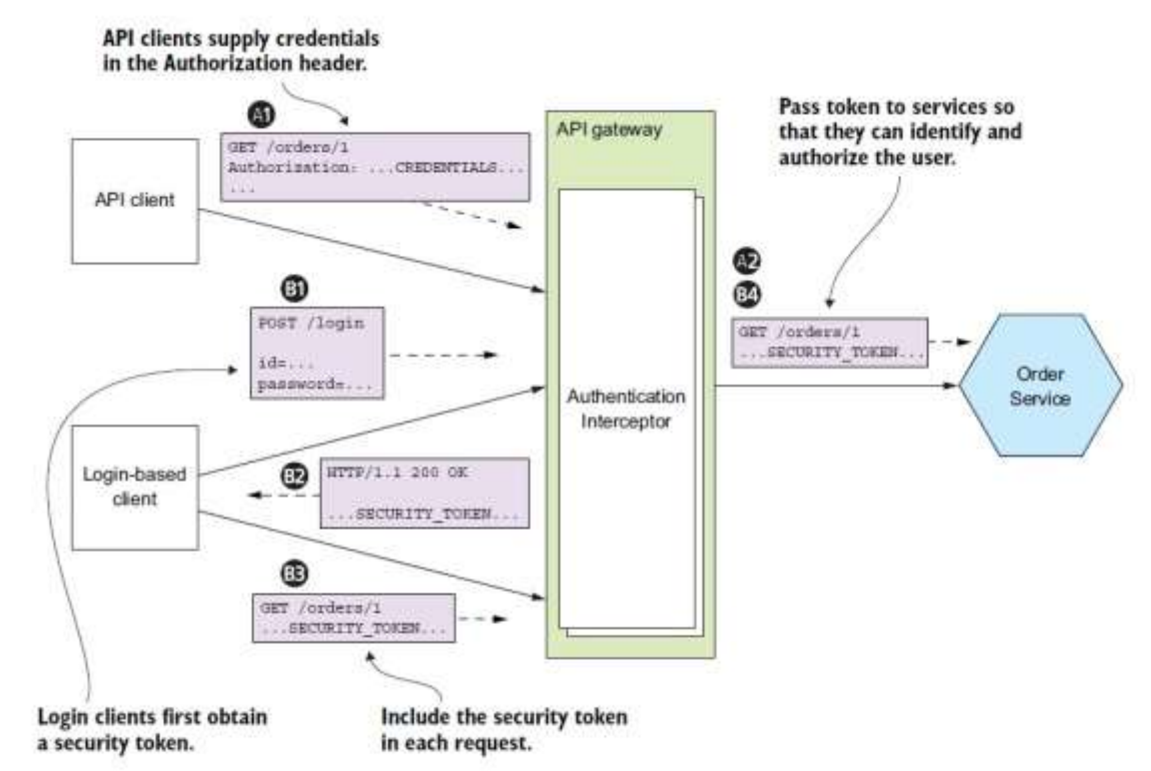
- Issues:
  - How implement the API gateway?
    - An event-driven/reactive approach is best if it must scale to scale to handle high loads.

Other Deployment Patterns

Security: Access Token Pattern

- You have a microservices application and have applied the API Gateway pattern.
- Problem:
  - How to communicate the identity of the requestor to the services that handle the request?
- Forces:
  - Services often need to verify that a user is authorized to perform an operation.
- Solution:
  - The API Gateway authenticates the request and passes an access token (e.g. JSON Web Token) that securely identifies the requestor in each request to the services.
  - The access token can be passed from one service to another as internal requests are made.

Access Token Pattern



API clients supply credentials in the Authorization header.

Pass token to services so that they can identify and authorize the user.

Login clients first obtain a security token.

Include the security token in each request.

```
GET /orders/1
Authorization: ...CREDENTIALS...
...
```

```
POST /login

id=...
password=...
```

```
HTTP/1.1 200 OK

...SECURITY_TOKEN...
```

```
GET /orders/1
...SECURITY_TOKEN...
```

```
GET /orders/1
...SECURITY_TOKEN...
```

- API clients and login clients are treated differently
  - API clients need to supply their credentials as part of their request
  - Login clients can first obtain a security token and then include that in their request

- Benefits
  - The identity of the requestor is securely passed around the system
  - Services can verify that a requestor is authorized for a given operation.

Externalized Configuration Pattern

- Context
  - Current microservices applications will often require infrastructure and 3rd party services
    - E.g. message broker, payment processing

- Problem
  - In that context, how do we enable a service to run in multiple environments without modification?

- We would like to be able to apply configuration variables with minimal intervention
  - Run different versions for different clients
  - Allow configuration changes with a simple service restart

Externalized Configuration Pattern: Forces

- A service must be provided with configuration data that tells it how to connect to external/3rd party services.
  - E.g. database network location and credentials.
- A service must run in multiple environments – dev, test, qa, staging, production – without modification and/or recompilation
- Different environments have different instances of the external/3rd party services
  - E.g. QA database vs production, test credit card processing vs production.

Solution

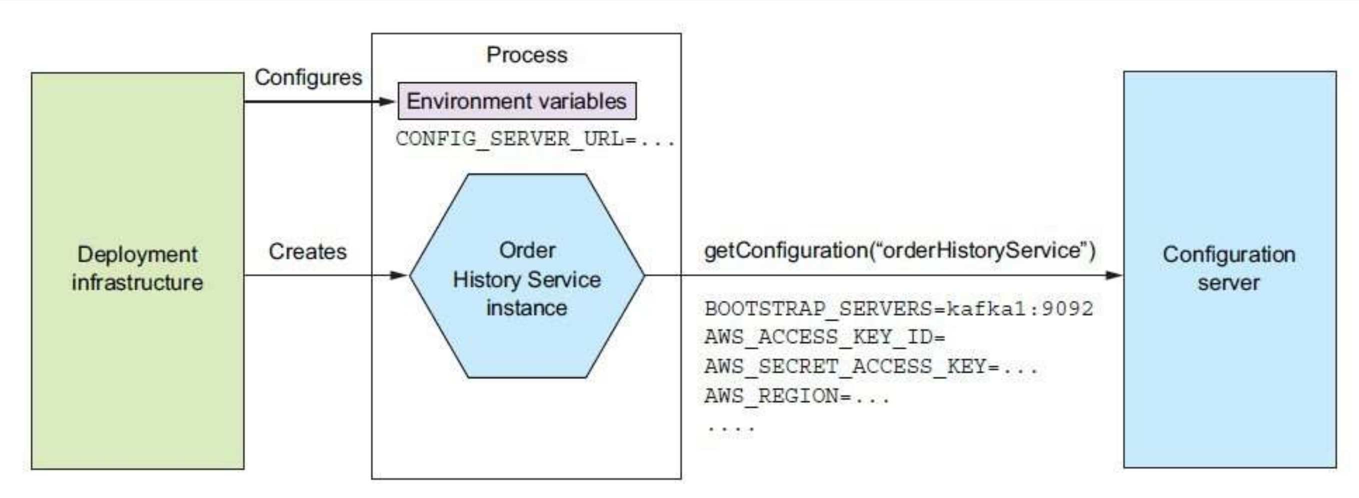
- Externalize all application configuration including the database credentials and network location.
- On startup a service reads the configuration from an external source.

- Push-based:
  - The deployment infrastructure creates an instance of the required service and writes the necessary configuration information to accessible locations such as
    - Command line arguments
    - Environment variables
    - A configuration file
  - The service then reads the configuration info on startup.

- Example: Spring Boot Externalized Configuration.

Solution: Pull based externalized configuration

- Pull-Based
  - A service instance reads its configuration properties from a configuration server.
  - On startup, the service instance queries the server for its configuration.
  - This can be implemented in a variety of ways
    - Through a version control system, like Git
    - SQL or NoSQL databases
    - Specialized configuration servers (E.g. Spring Cloud Config Server, AWS Parameter Store)
- Example: Spring Cloud Config Server


Deployment infrastructure
Configures
Creates
Process
Environment variables
Order History Service instance
getConfiguration("orderHistoryService")
Configuration server

```
CONFIG_SERVER_URL=...
```

```
BOOTSTRAP_SERVERS=kafka1:9092
AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=...
AWS_REGION=...
....
```

Resulting Context

- Benefit
  - The application runs in multiple environments without modification and/or recompilation

- Drawback
  - Yet another piece of infrastructure to be setup and maintained
    - Various open-source frameworks such as Spring Cloud Config can mitigate this.

- Issues
  - How to ensure that when an application is deployed the supplied configuration matches what is expected?

Observability Patterns

- Q: When your application is deployed in production, what information would you like to know about it?

- Requests per second?
- Resource utilization?
- Service instance failures?
- Triggers of defensive patterns like circuit breaker?
- Timing of requests to responses?

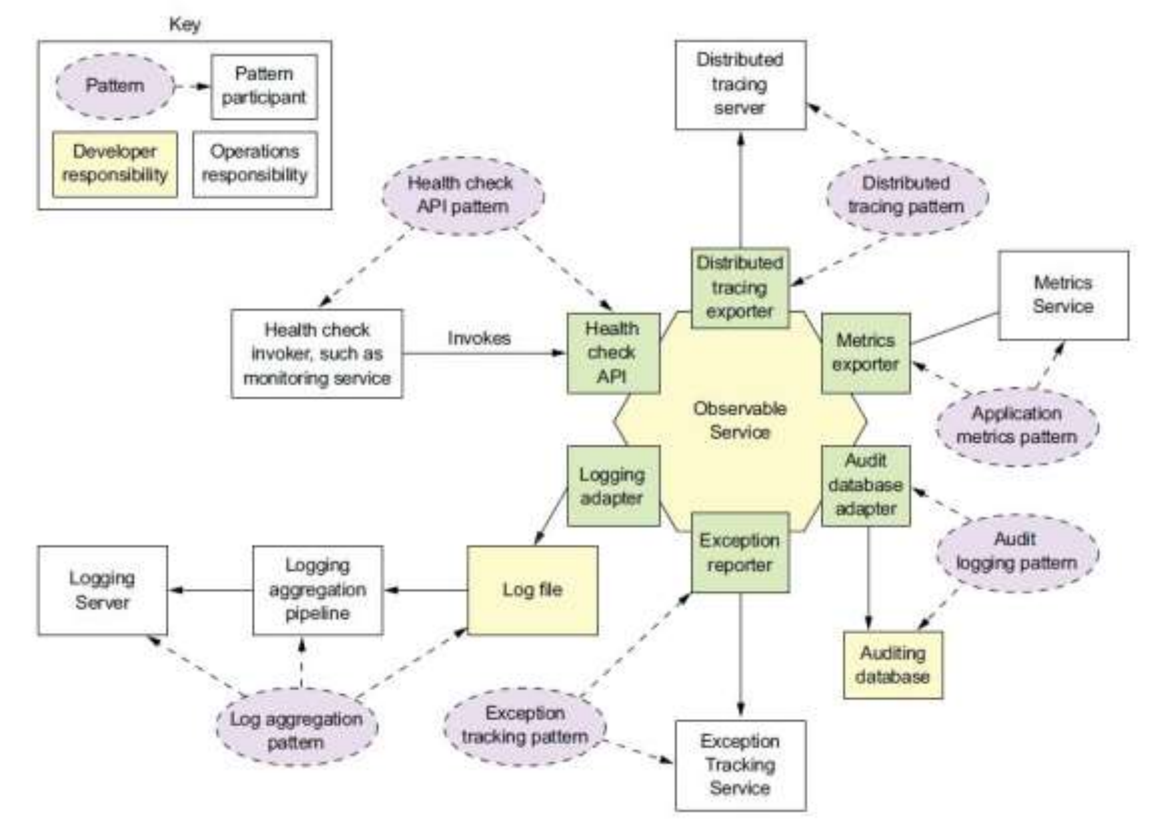
Observability Patterns

- Some of these things are pure production support (monitoring hardware and availability)

- However you should implement some patterns to expose the behaviour of your services’ instances to track and visualize the system state, and monitor for problems.
  - Health Check API: an endpoint that exposes the health of the service
  - Log Aggregation: Log service activity and write logs to a centralized server
  - Distributed tracing: Trace external requests as they flow between services
  - Exception tracking: Report all exceptions to an exception tracking service
  - Application Metrics: Expose all maintenance metrics (counters and gauges) to a server.
  - Audit Logging: Log all user actions

Observability Patterns

- Examples:
  - Health Check API: Spring Boot Actuator
  - Log Aggregation: Log4j + ELK
  - Distributed Tracing: Spring Cloud Sleuth + Open Zipkin



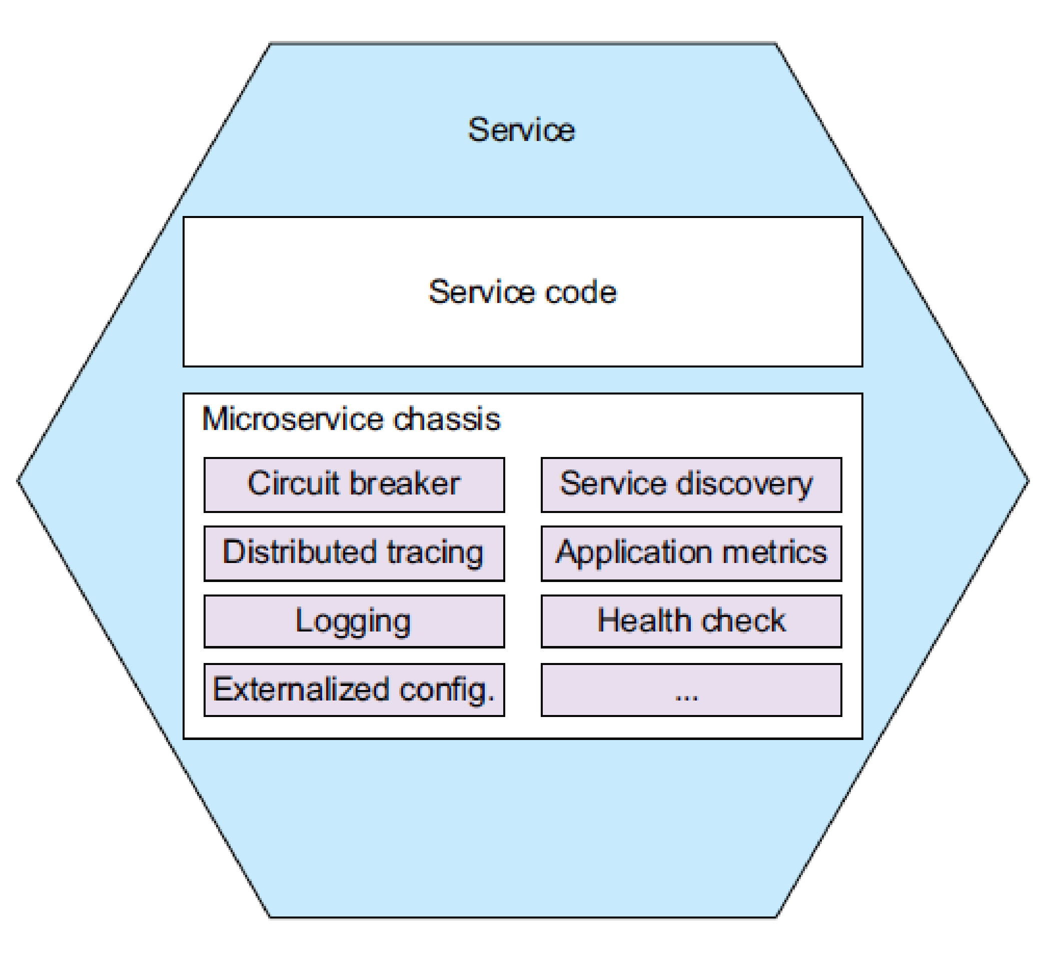
Microservice Chassis Pattern

- We have seen lots of concerns a service must implement
  - Externalized Configuration
  - Security Access
  - Metrics, Health Check, Exceptions etc

- Lots of these are standard implementations you would not want to reimplement each time.

- You can build your services upon a microservices chassis
  - A framework or a set of frameworks that handle these concerns.

Microservice Chassis



- Cross-cutting concerns are available for service code “out-of-the-box”.
- Speed up your delivery time and reliability.
- E.g. In Java: Spring Cloud, Spring Boot, in GoLang: Go Kit, Micro.

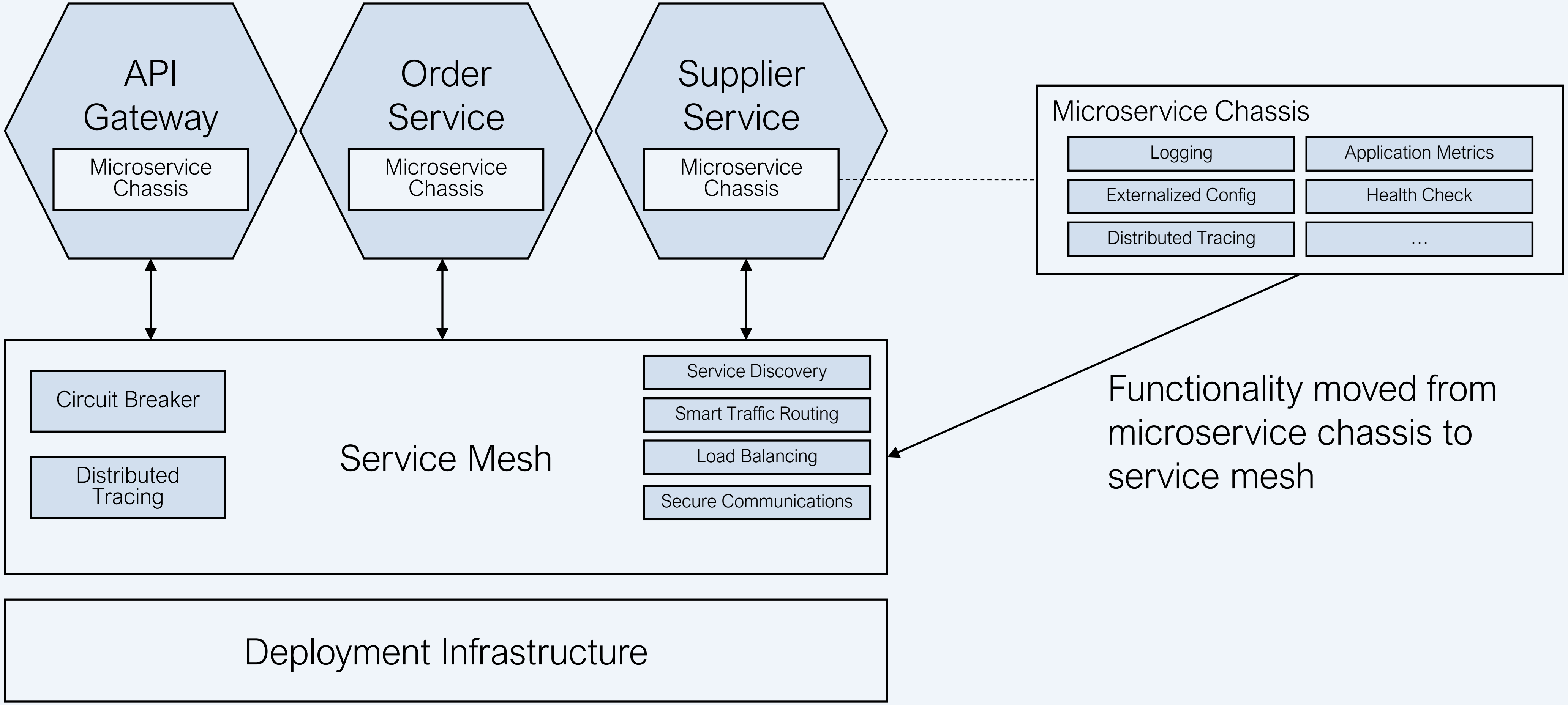
- Many chassis functions may be provided by your infrastructure (e.g. service discovery on OpenShift)

Service Mesh Pattern

- A microservice chassis is, by nature, implemented in one programming language
  - That means you need a different one for each language your application uses.

- An alternative is a Service Mesh – a networking infrastructure that mediates the communication between a service and other services and external applications.
  - All network traffic into and our of a service goes through the service mesh.
  - The service mesh implements concerns
    - Circuit breakers
    - Distributed tracing
    - Service discovery
    - Load balancing
    - Traffic routing
    - Secure inter-process communication
    - Etc.

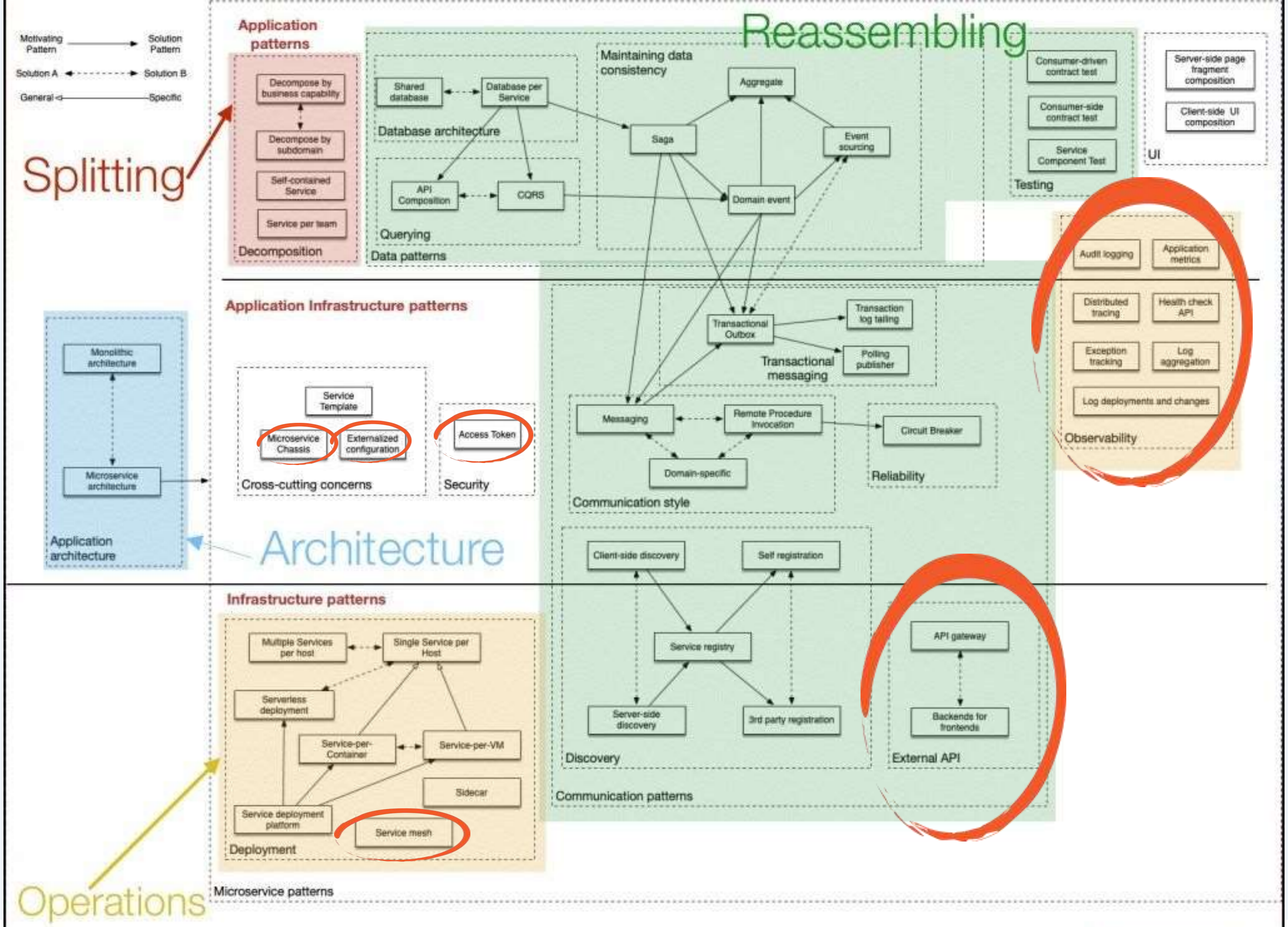
Service Mesh



Functionality moved from microservice chassis to service mesh

Service Mesh Examples

- Istio https://istio.io
- Linkerd https://linkerd.io
- Conduit https://conduit.io



Motivating Pattern → Solution Pattern

Solution A ← - - → Solution B

General ◁— Specific

Splitting

Reassembling

Architecture

Operations

Application patterns

- Decompose by business capability
- Decompose by subdomain
- Self-contained Service
- Service per team

Decomposition

Database architecture

- Shared database
- Database per Service

Querying

- API Composition
- CQRS

Maintaining data consistency

- Aggregate
- Saga
- Event sourcing
- Domain event

Data patterns

Testing

- Consumer-driven contract test
- Consumer-side contract test
- Service Component Test

UI

- Server-side page fragment composition
- Client-side UI composition

Application Infrastructure patterns

Cross-cutting concerns

- Service Template
- Microservice Chassis
- Externalized configuration

Security

- Access Token

Application architecture

- Monolithic architecture
- Microservice architecture

Transactional messaging

- Transactional Outbox
- Transaction log tailing
- Polling publisher

Communication style

- Messaging
- Remote Procedure Invocation
- Domain-specific

Reliability

- Circuit Breaker

Observability

- Audit logging
- Application metrics
- Distributed tracing
- Health check API
- Exception tracking
- Log aggregation
- Log deployments and changes

Discovery

- Client-side discovery
- Self registration
- Service registry
- Server-side discovery
- 3rd party registration

External API

- API gateway
- Backends for frontends

Communication patterns

Infrastructure patterns

Deployment

- Multiple Services per host
- Single Service per Host
- Serverless deployment
- Service-per-Container
- Service-per-VM
- Sidecar
- Service deployment platform
- Service mesh

Microservice patterns

Learn-Build-Assess Microservices http://adopt.microservices.io

Owerya RESOURCING

Summary

- External API Patterns

  Issues with External APIs for microservices
  The API Gateway and Backend for Frontend Patterns

- Service Deployment Patterns

  Security - Access Token Pattern
  Externalized Configuration Pattern
  Observability Patterns
  Microservice Chassis Pattern
  Service Mesh

Questions or Comments?

Appendix: Managing a Move to Microservices

- The following presentation is quite long (around 35mins) that shows how this company used SwaggerHub to manage a move to Microservices.
- The speaker brings up some general good points about doing the move, and specific tips about APIs and comms between services.

Webinar ON DEMAND

Modernizing the Enterprise API Development Process

Travis Koenig, RealPage