GRAND VALLEY STATE UNIVERSITY
www.gvsu.edu

Cloud Computing Design Patterns

CIS437
Erik Fredericks // frederer@gvsu.edu

Adapted from Google Cloud Computing Foundations, Overview of Cloud Computing (Wufka & Canonico)

Design patterns for cloud apps!

https://github.com/mehdihadeli/awesome-software-architecture/blob/main/docs/cloud-design-patterns/cloud-design-patterns.md

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/introduction.html

https://www.techtarget.com/searchcloudcomputing/tip/5-cloud-design-patterns-to-create-resilient-applications https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/antipatterns/antipatterns-to-avoid

https://www.doit.com/cloud-landing-zone-anti-patterns-to-avoid/

First off...

What is a design pattern?

Next, some 350 slides to *remind us* what it is

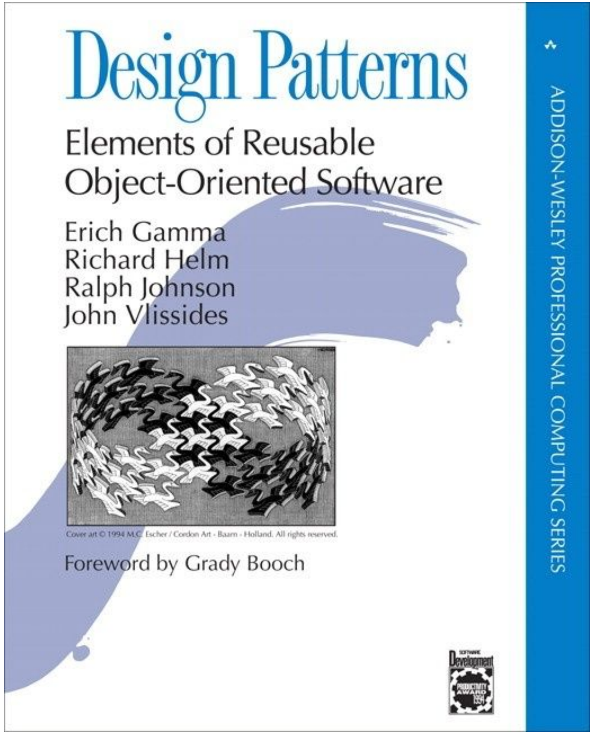
What *is* a design pattern?

A design pattern “...names, abstracts, and identifies the key aspects of a common design structural that make it useful for creating a reusable object-oriented design.”*

A design pattern is a **proven** solution to a recurrent problem in a context.

An effective, reusable, proven structure/**communication** solution for a given object-oriented design problem.

What do we mean by proven?
How does communication fit in?


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

*From the book pictured

Why study them (the design patterns)

Reuse existing, high-quality solutions to commonly recurring problems

Establish common terminology to improve communications within teams

- Shifts the level of thinking to a higher perspective.

Improve team communication and individual learning

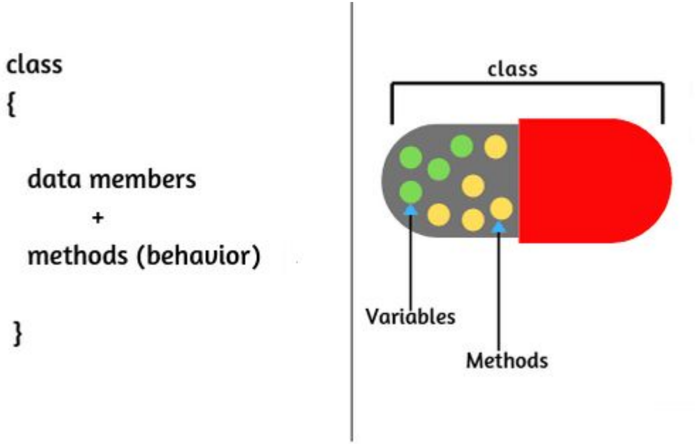
Improve modifiability and maintainability of code

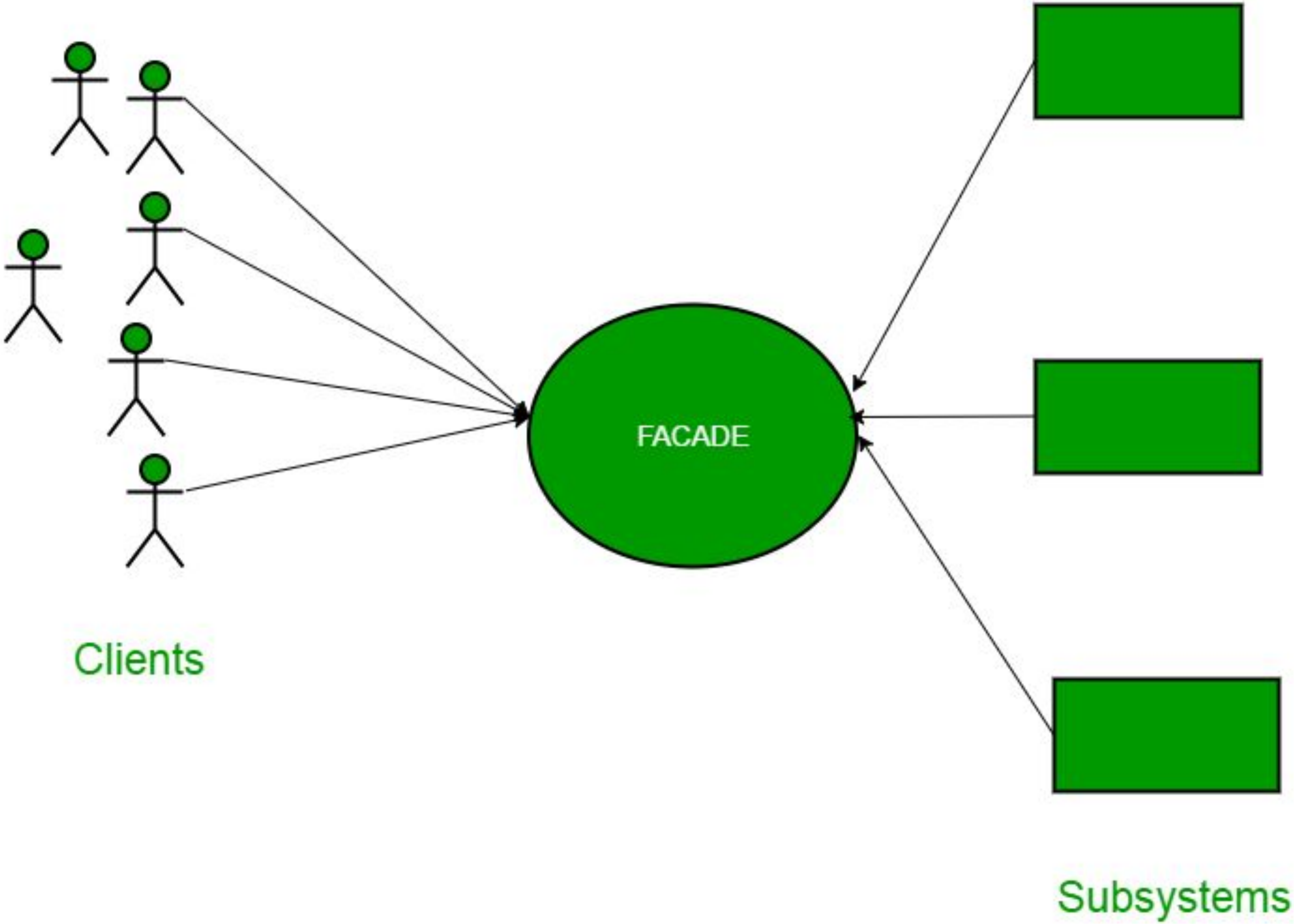
- Design patterns are time-tested solutions (i.e., “proven”)

Why study them (the design patterns)

Adoption of improved object-oriented design strategies

- Encapsulation and information hiding
- Design to interfaces
- Favor composition over inheritance




FACADE
Clients
Subsystems

Facade pattern

Pattern Category: **Structural**

Intent:

- Provide a unified interface to a set of interfaces in a subsystem.
- Facade defines a unified higher-level interface that makes the subsystems easier to use.

Problem addressed:

- Using design patterns often leads to a complex system of many small components which may be daunting for the casual user. It would be nice if there were a way to provide a simple interface for the basic functionality that is needed most often.

Solution:

- Create a Facade class that encapsulates the basic functionality of the system by bundling together common operations

When else would a *Facade* class be useful?

Cloud design patterns

Same as normal design patterns, but specific to cloud applications

- i.e., proven solutions to common problems

What are our concerns again?

- Normal application with:
  - Globally distributed userbase
  - Load spikes
  - … etc.

Now...

There are a *lot* of design patterns out there

- And there are ever-growing lists for the cloud
    - https://github.com/mehdihadeli/awesome-software-architecture/blob/main/docs/cloud-design-patterns/cloud-design-patterns.md

We're going to walk through a few of them

- Keep learning though!
- A good portion of them can be useful **for your future career**

Cloud fallacies (similar to distributed computing fallacies)

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Component versioning is simple
- Observability implementation can be delayed
    - i.e., monitoring and understanding what went wrong

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing
https://learn.microsoft.com/en-us/azure/architecture/patterns/

Publish/Subscribe

Asynchronous communication (decoupling transmission of data)

Publisher - sends out data on *topics*
Subscriber - receives data for *topics they've subscribed to*
(Broker) - middleman to disseminate traffic / store who gets what

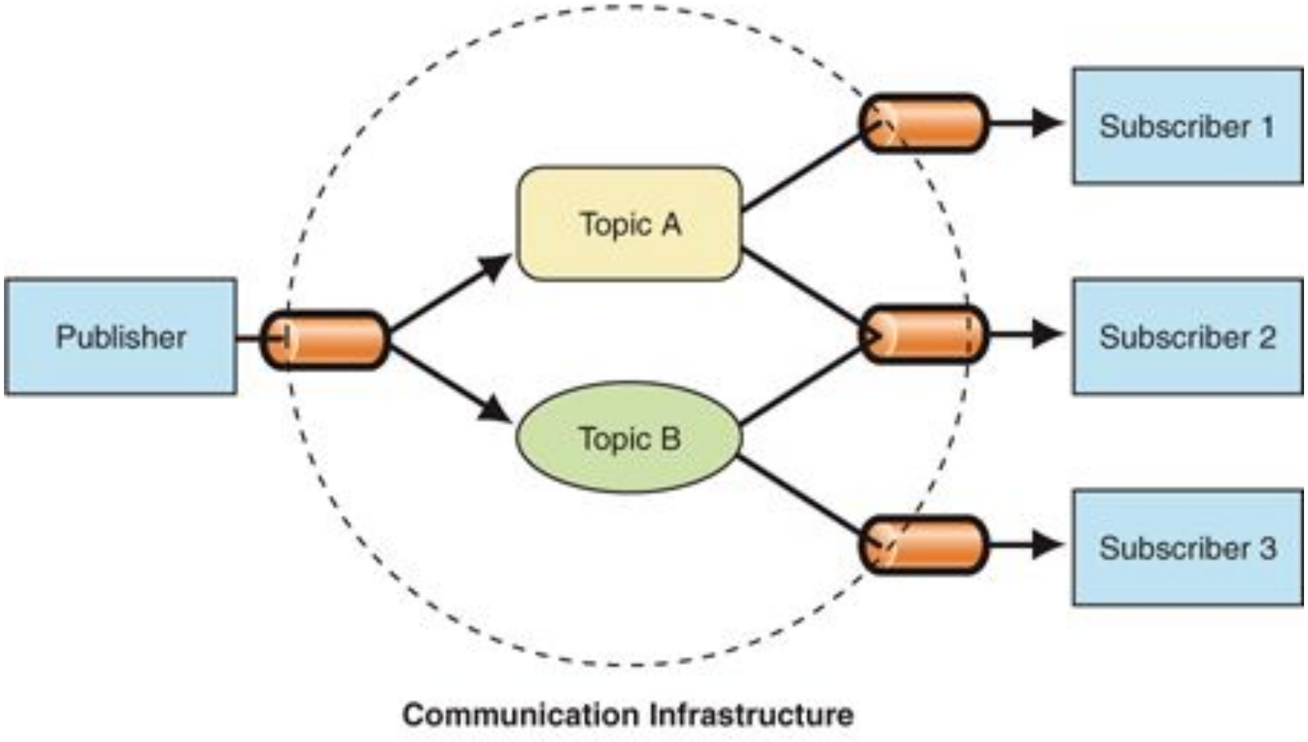
Multiple choices of quality of service

- i.e., do we care if the data is received or non-corrupted?
    - Why?
    -

Common in IoT applications!

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/publish-subscribe.html
https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber

Pub/sub


Publisher
Topic A
Topic B
Subscriber 1
Subscriber 2
Subscriber 3
Communication Infrastructure

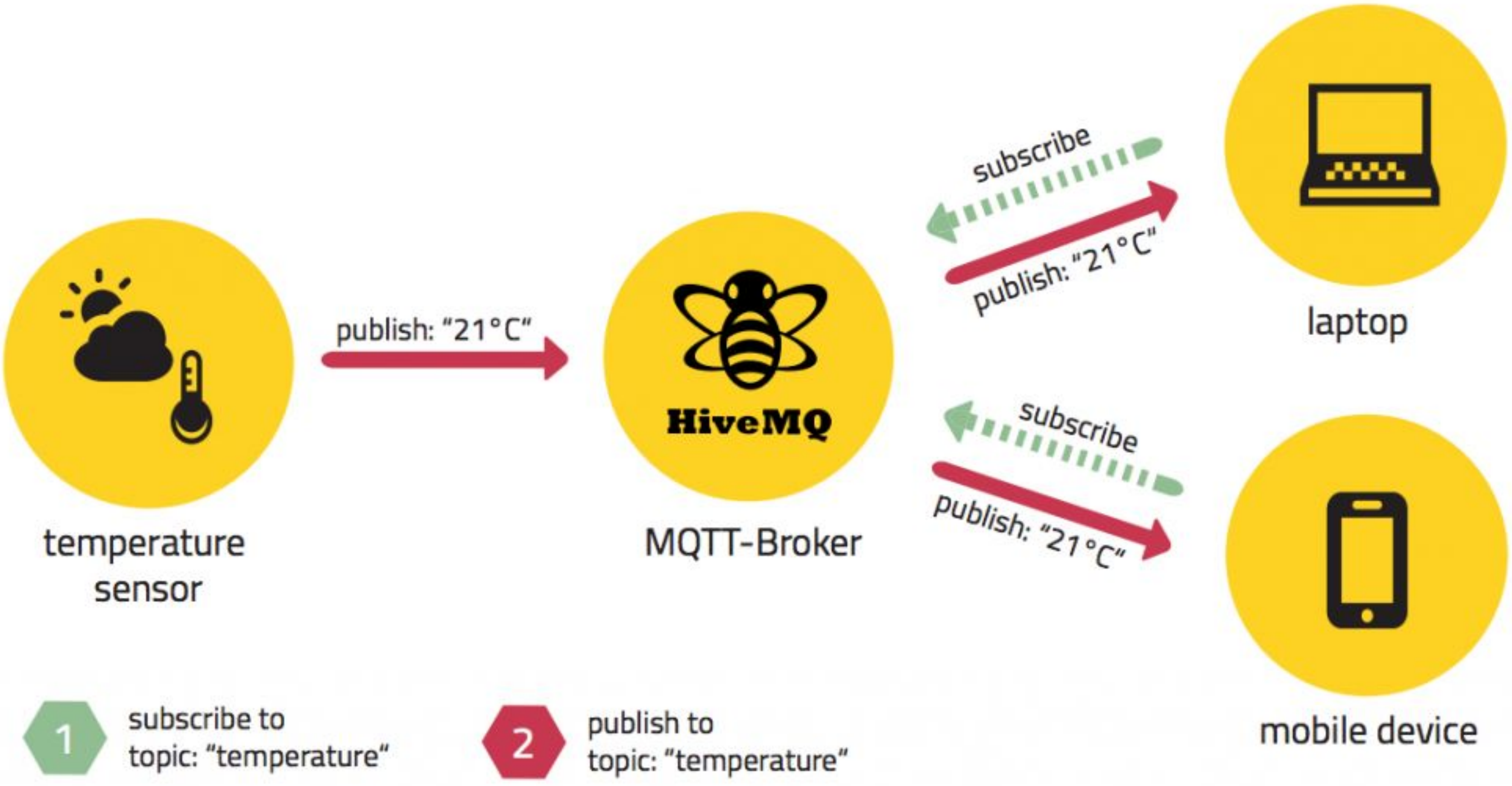
Clients and brokers

Client:

- Publisher or subscriber that connects to a **broker**
- Persistent (maintains connection) or transient (not tracked)

Broker (**central hub**):

- Receiving and filtering messages
- Understanding which clients are 'interested' in data
- Sending messages to subscribed clients
- Authenticating/authorizing clients



temperature sensor

publish: “21°C”

HiveMQ

MQTT-Broker

subscribe

publish: “21°C”

laptop

subscribe

publish: “21°C”

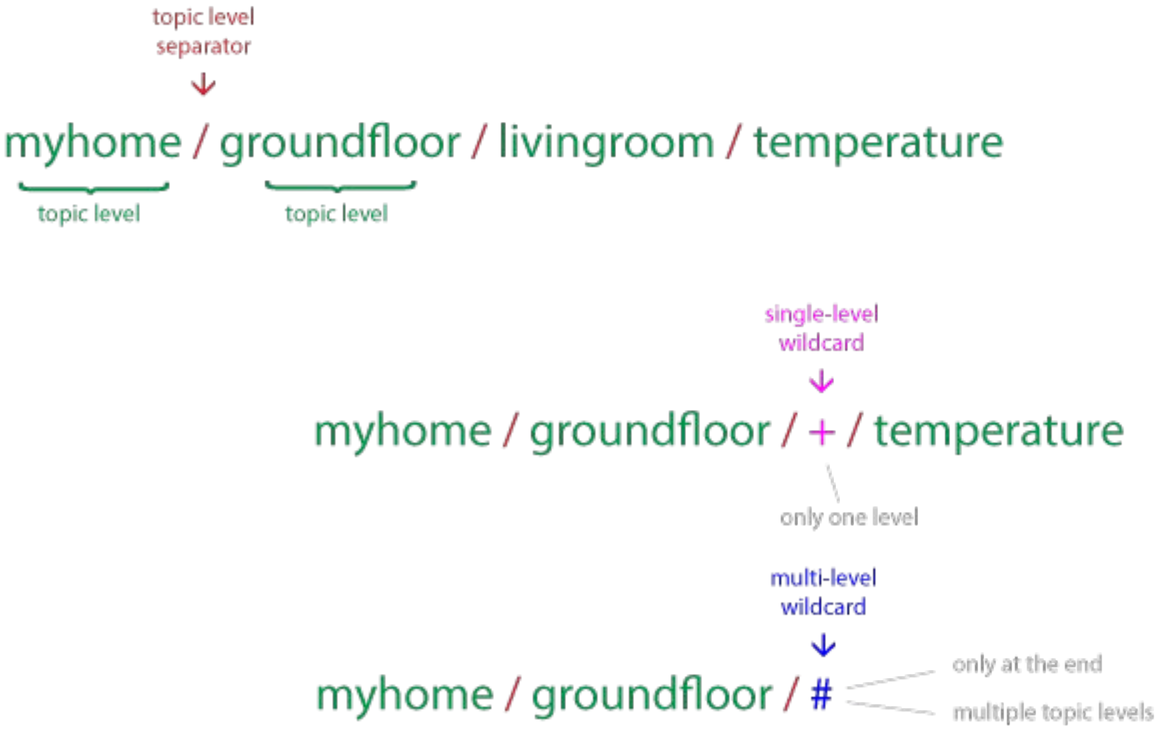
mobile device

1 subscribe to topic: “temperature”

2 publish to topic: “temperature”

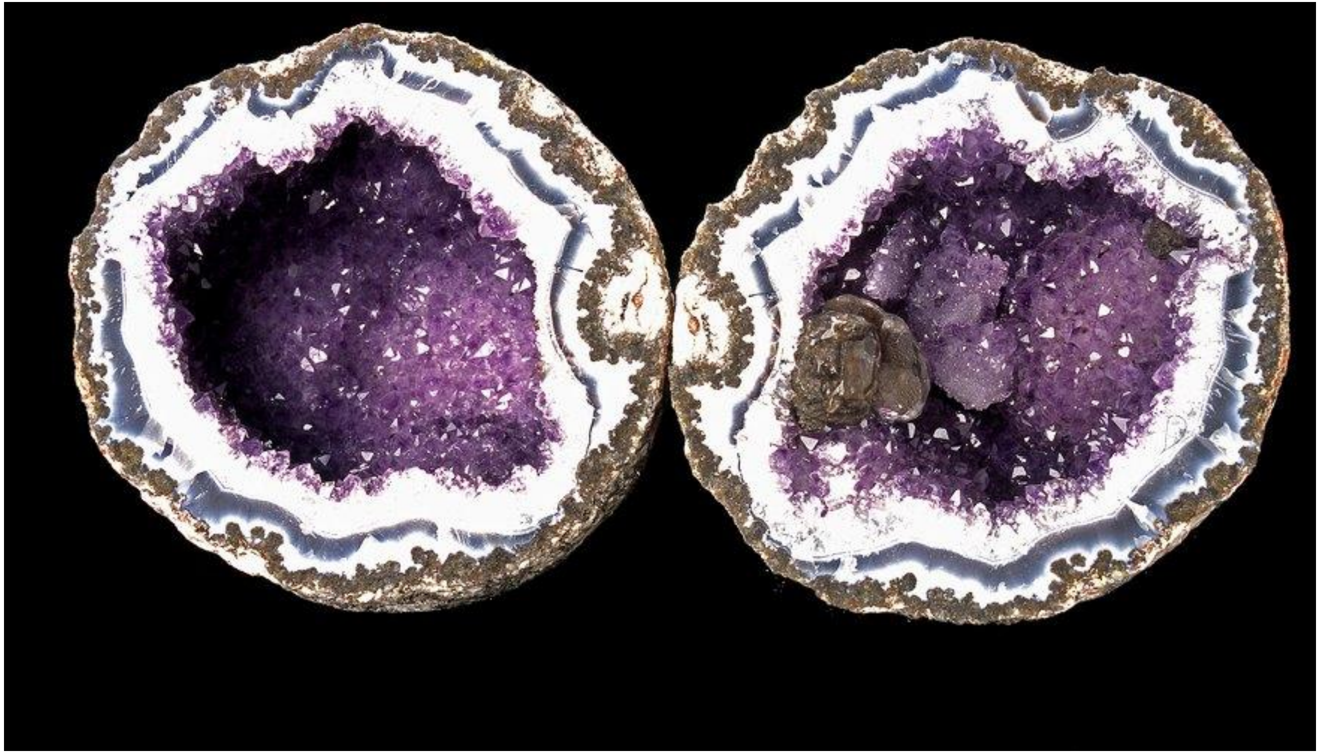
Topics

Hierarchical string that filters messages for clients



https://www.youtube.com/watch?v=f5o4tIz2Zzc

Geode



Geode

Why?

- Availability required worldwide (or, at least in multiple geographic regions)
- Scale required!

Concerns:

- Network latency + traffic management
- Worldwide deployment
- Data geo-distributed

Geode

How can we solve here?

Geode

How can we solve here?

- Create a bunch of geographic nodes (geodes...)
- "Satellite" deployments

Essentially, have a good devops approach (CI/CD)

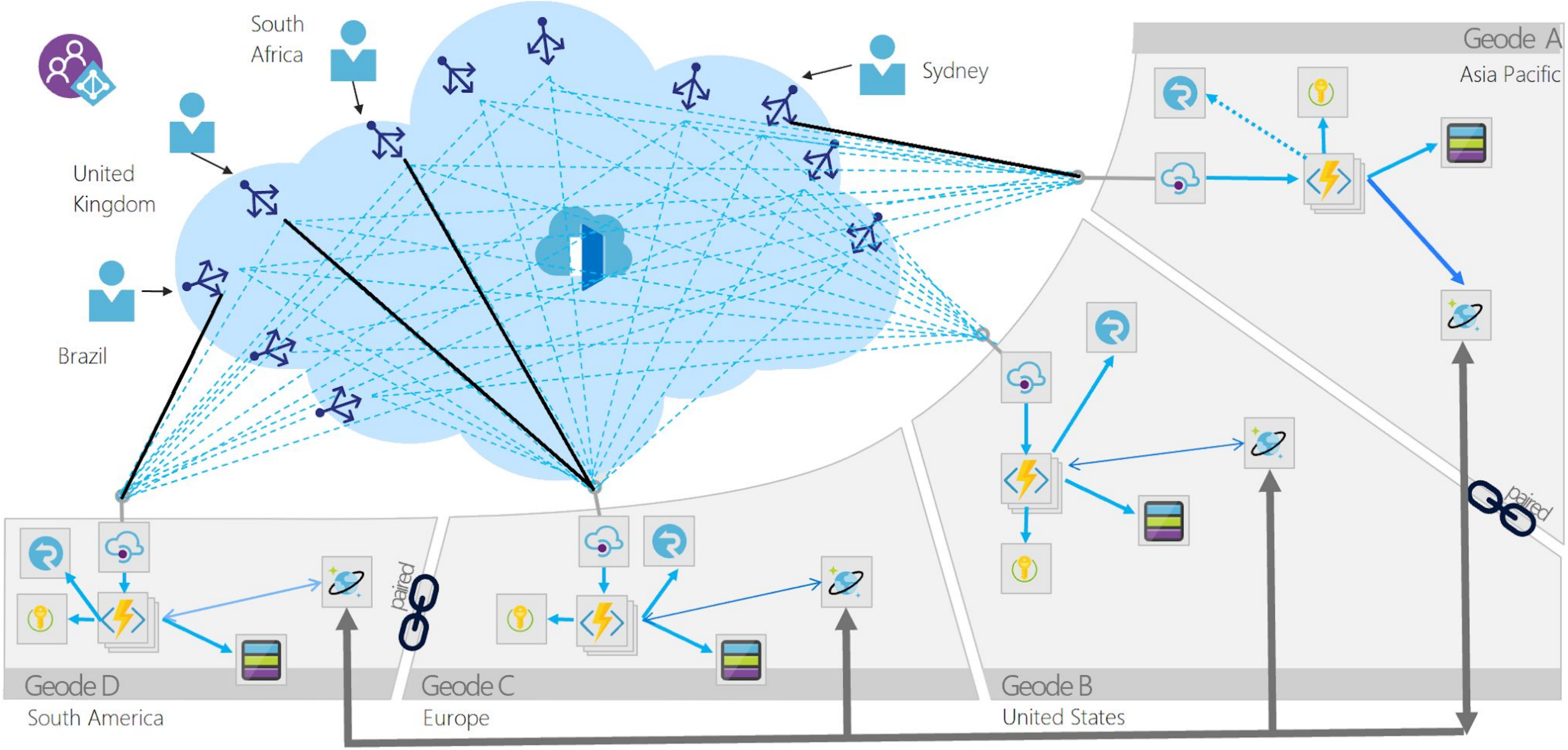
- Deploy your app (ideally, templated)
- Reflect it to multiple regions automatically (via CI/CD)
- Load balance to direct traffic

Should never be used by itself (> 1 geode required for the pattern)

Updates to app reflect automatically to all geodes!

https://learn.microsoft.com/en-us/azure/architecture/patterns/geodes

Geode



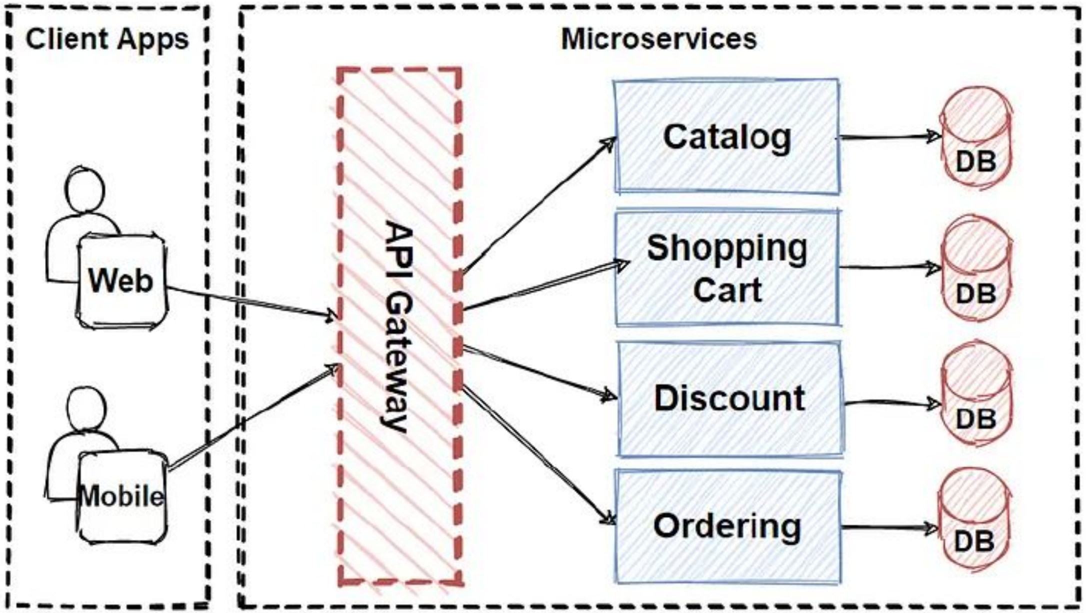
https://medium.com/design-microservices-architecture-with-patterns/api-gateway-pattern-8ed0ddfce9df

API Gateway

Target application:

- Collection of microservices
- Multiple client frontends

Similar to facade pattern!



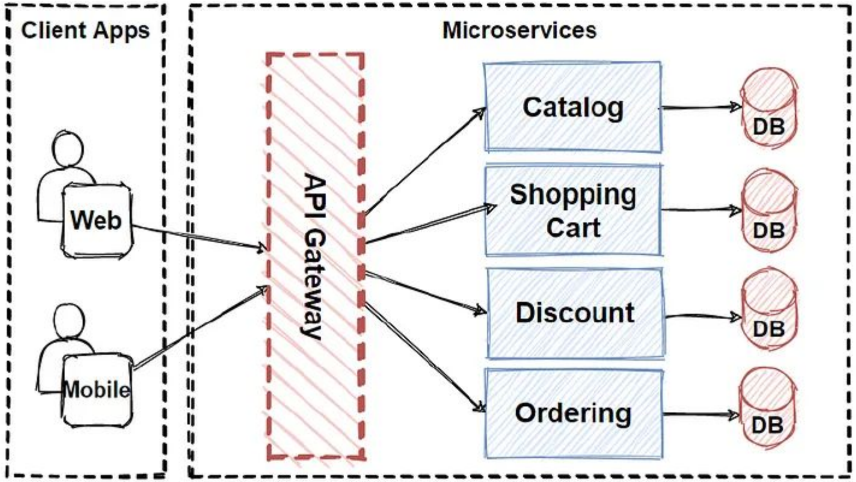
API Gateway

Single point-of-entry from clients to backend

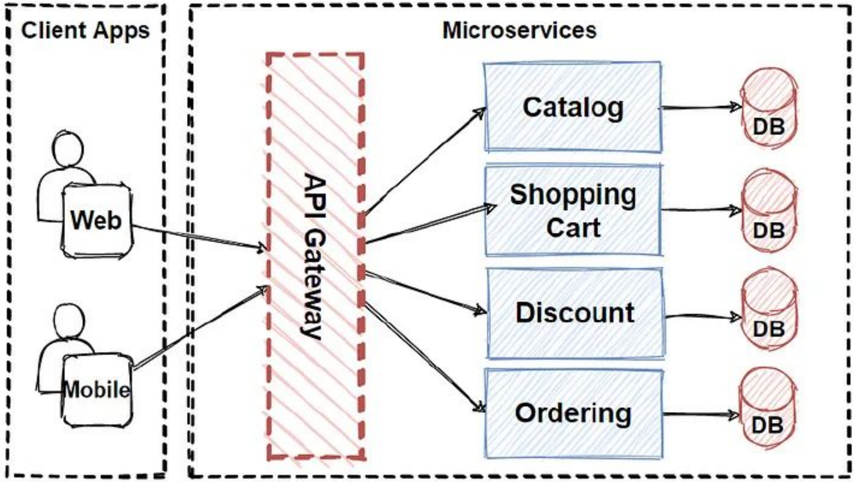
- Complexity behind the scenes hidden/abstracted

Difference to facade?

- Uses reverse proxy / gateway routing for communication
- i.e., requests from client are routed appropriately to microservice needed



How could we implement this?



How could we implement this?

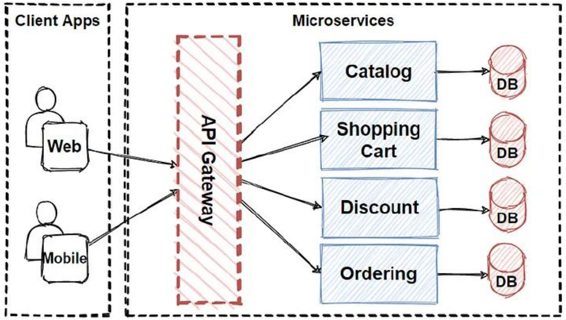
Microservices with databases

- Naturally

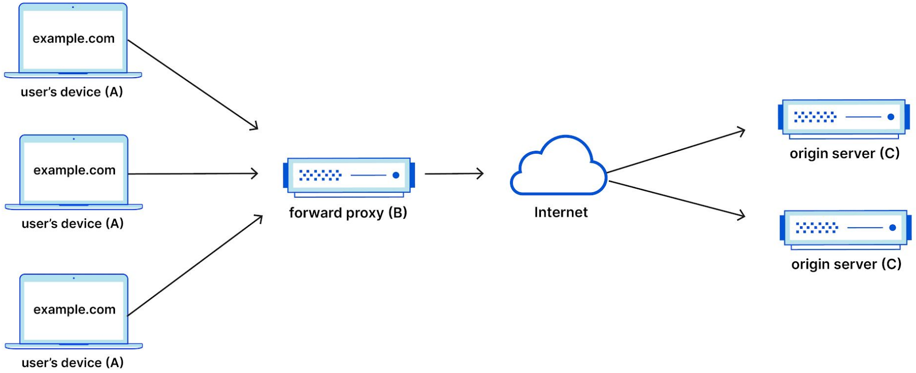
Some form of routing application

- Combination of serverless functions that know where the microservices are?

- VM that handles reverse proxy (or a reverse proxy server itself)



Forward Proxy Flow



API Gateway

Advantages?

Disadvantages?

API Gateway

Advantages?

- Can aggregate client requests into single response
  - i.e., Multiple microservices queried and lumped together
- Load balancing possible
- Authorization/Authentication handled by networking layer

Disadvantages?

- Single(-ish) point of failure
- Extra complexity
- Possible anti-pattern
  - Bad design!
  - Could be giving the gateway "too much to do"

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/circuit-breaker.html

Circuit breaker

Prevents caller service from retrying after multiple timeouts/failures

- Detects when callee is available again

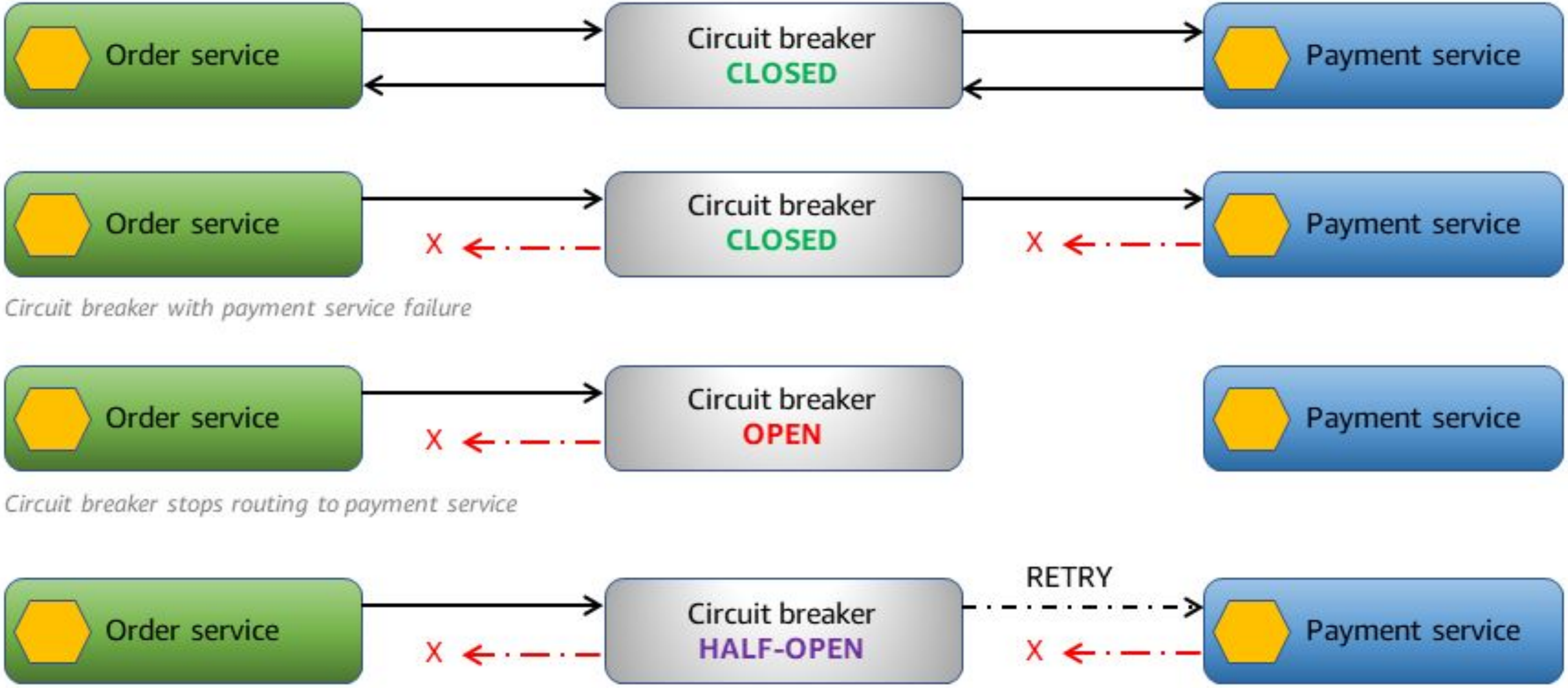
Possible causes:

- Network disruption
- Callee overloaded
- etc.

Issue avoiding?

- Consuming resources from numerous retried calls
  - Could impact cost and/or performance!

Circuit breaker



Order service → Circuit breaker CLOSED → Payment service

Order service → Circuit breaker CLOSED → Payment service

X X

Circuit breaker with payment service failure

Order service → Circuit breaker OPEN Payment service

X

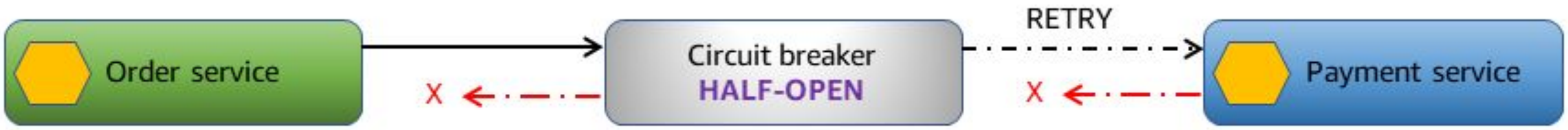
Circuit breaker stops routing to payment service

Order service → Circuit breaker HALF-OPEN RETRY Payment service

X X

Circuit breaker periodically retries payment service

Circuit breaker



Order service

Circuit breaker

HALF-OPEN

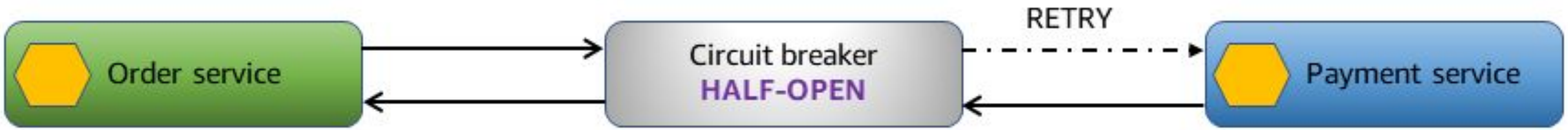
RETRY

X

X

Payment service

Circuit breaker periodically retries payment service



Order service

Circuit breaker

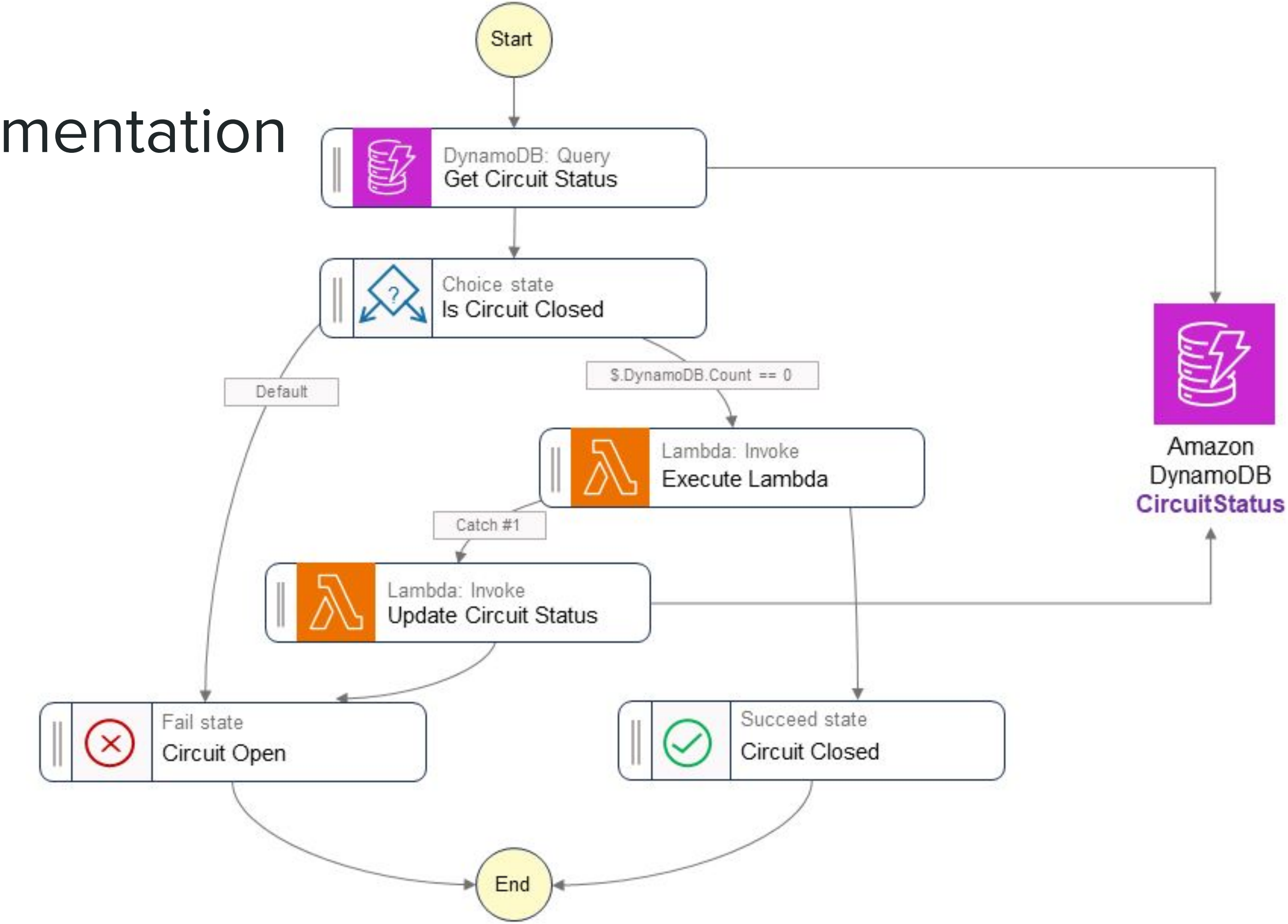
HALF-OPEN

RETRY

Payment service

Circuit breaker with working payment service

AWS Implementation



And?

Advantages?

Disadvantages?

And?

Advantages?

- Reduction in unnecessary retry calls
- Possible reduction in 'stale' or duplicate calls
    - Perhaps a credit card auth. got stuck in the system?

Disadvantages?

- Complexity!
    - Requires multiple services (database, serverless, state machine, etc.)
    - Extra $$ for extra services!

- Multiple points of failure
    - What if you have an issue with your database now? Or one of your lambdas?

In-class work!

Break up into teams of 2 or 3

What kind of design pattern would you apply to the following situations? Why?

1) A system deployed to a factory comprising hundreds of sensor nodes reporting on environment readings, conveyor belt status, etc., that transmits the data to a central application for analysis

2) A global company using an ERP system (enterprise resource planning, used from managing HR topics to project status/tasks) across its offices worldwide (think: lots of data to deal with, reports to generate, backups to make)

3) An application deployed to POS (point of sale) systems (think - cash registers) that communicate with a company's backend servers to update products in/out and cash in/out