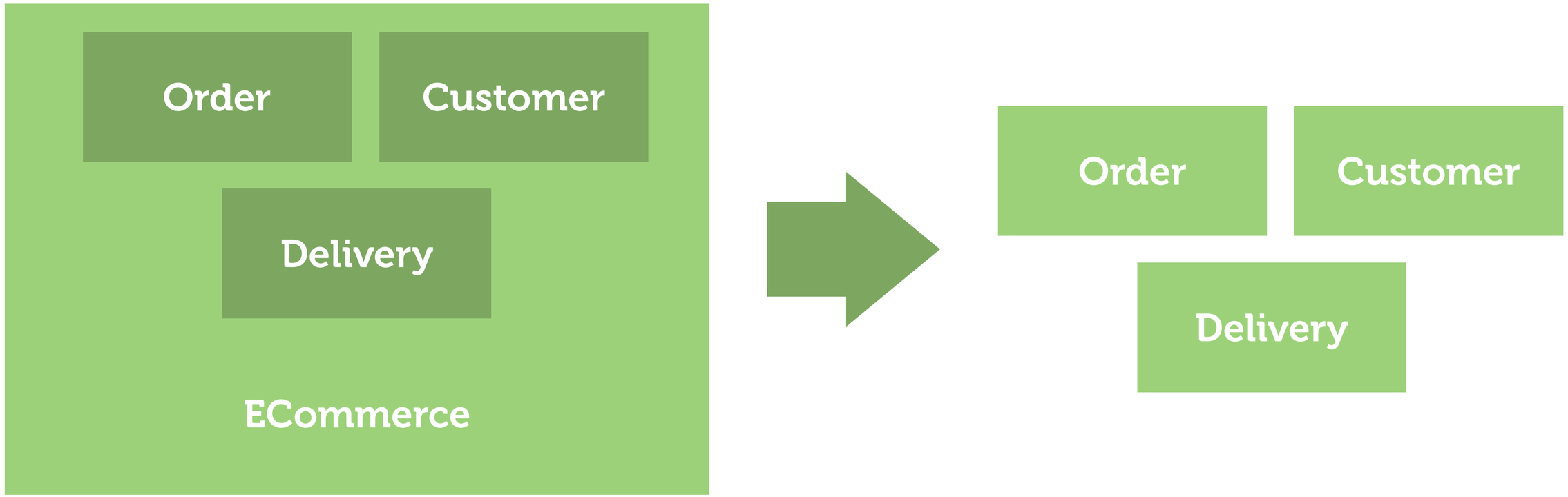
REST vs. Messaging

Integration Approaches for Microservices

Eberhard Wolff
/ ewolff
innoQ

Oliver Gierke
/ olivergierke
Pivotal

Microservices



Package

WAR

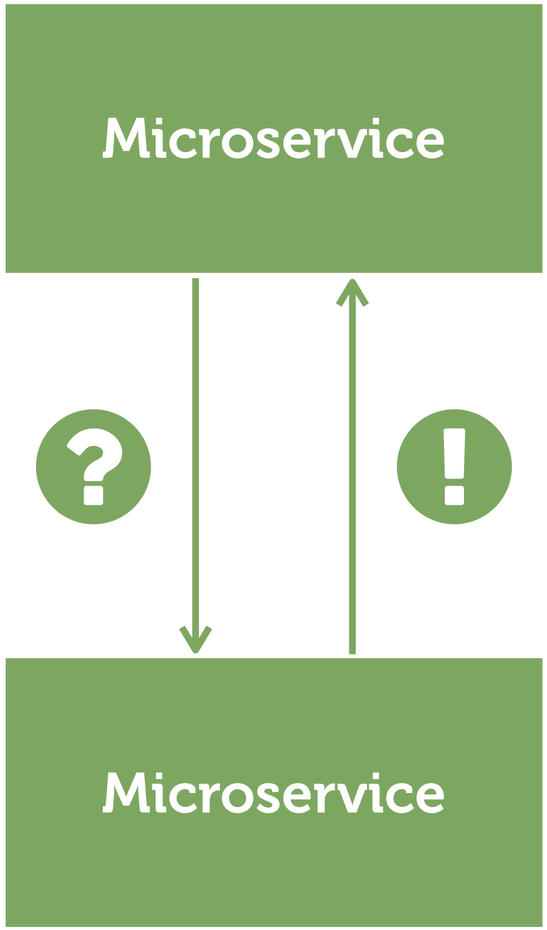
Integration

UI	Links	Modular UI
Logic	REST	Messaging
Data	Replication	No Common Schema!

Overview

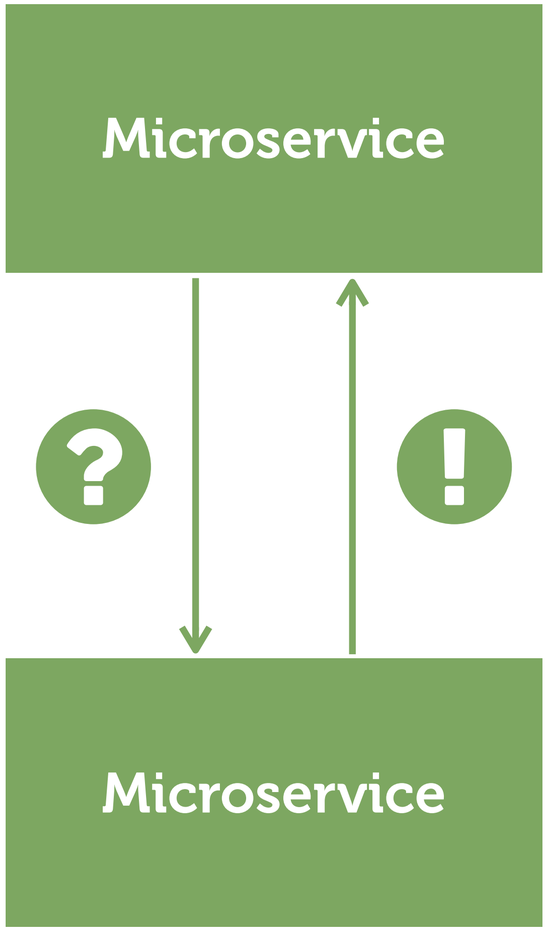
REST

- Architectural style
- Constraints in architecture result in traits of the system
- Identifiable resources, uniform interface, representations, hypermedia
- Synchronous by default



REST

- Service discovery
  - DNS or registry & hypermedia
- Load balancing
  - Dedicated infrastructure
  - Software load-balancer (Ribbon)

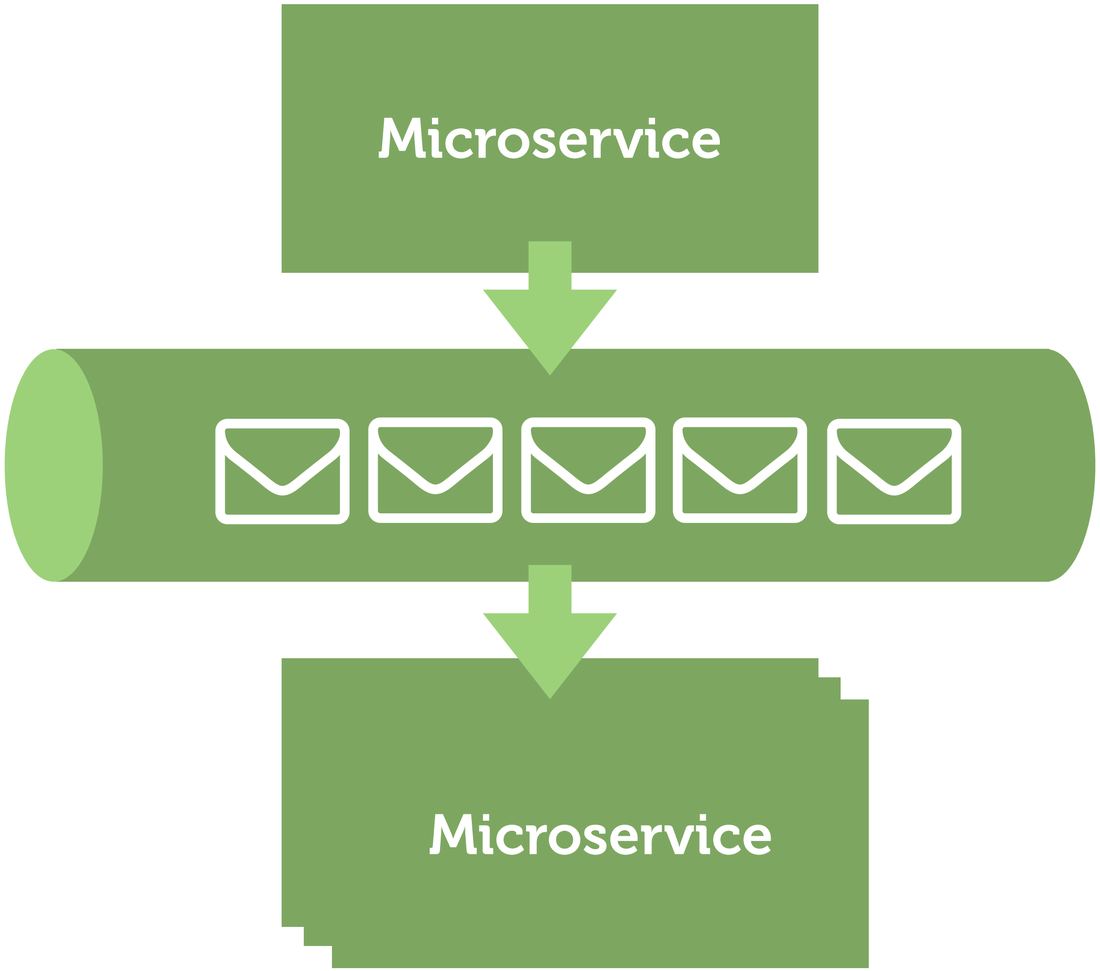


Messaging

- Microservices send message
- Asynchronously

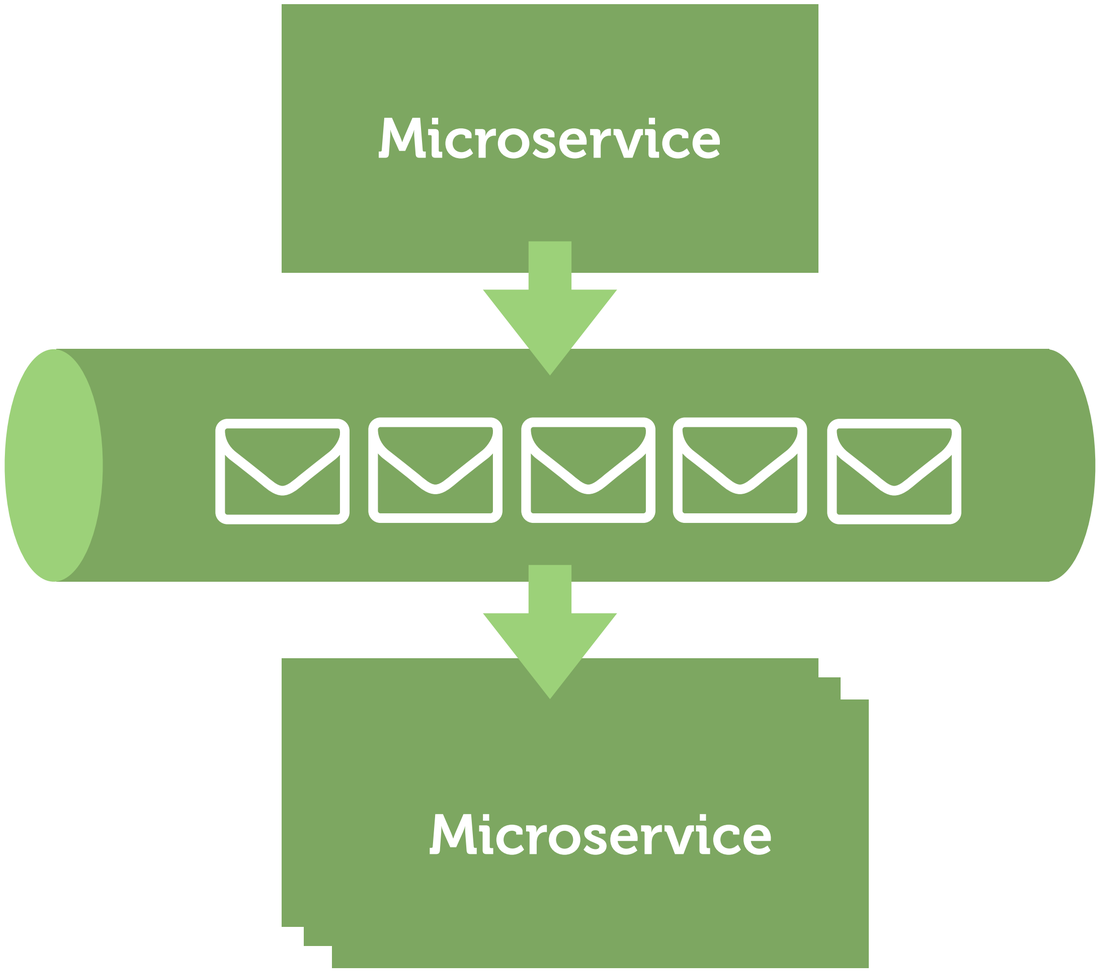
Load Balancing

- Can have any numbers of instances of a receiver
- Load Balancing very easy



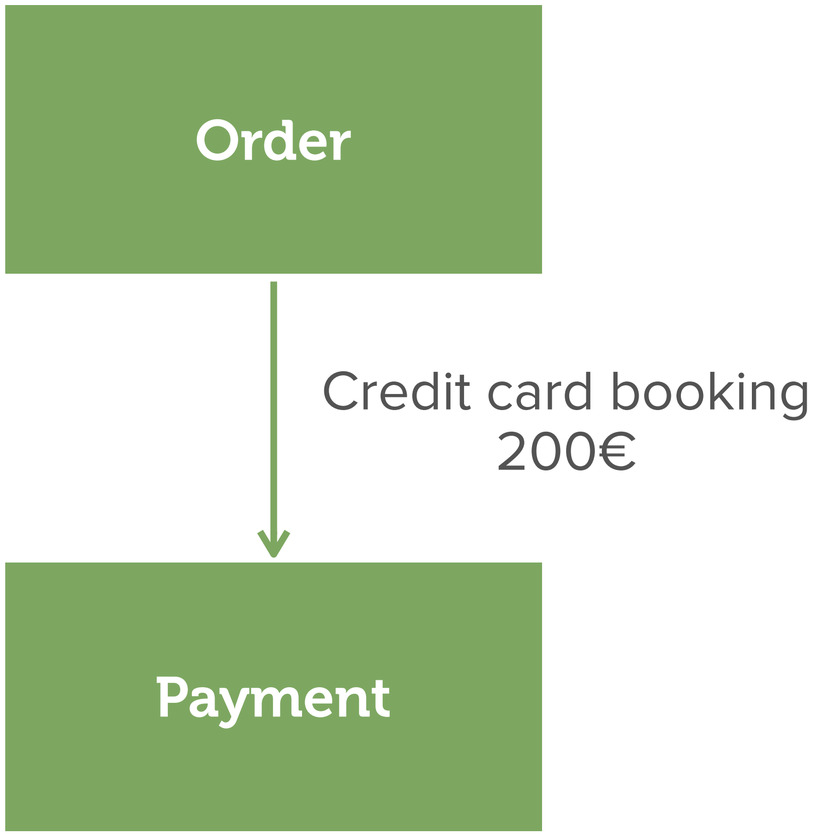
Service Discovery

- Microservices send messages to queues / topics
- Receiver(s) read messages
- Decoupled by queues & messages
- No need for service discovery



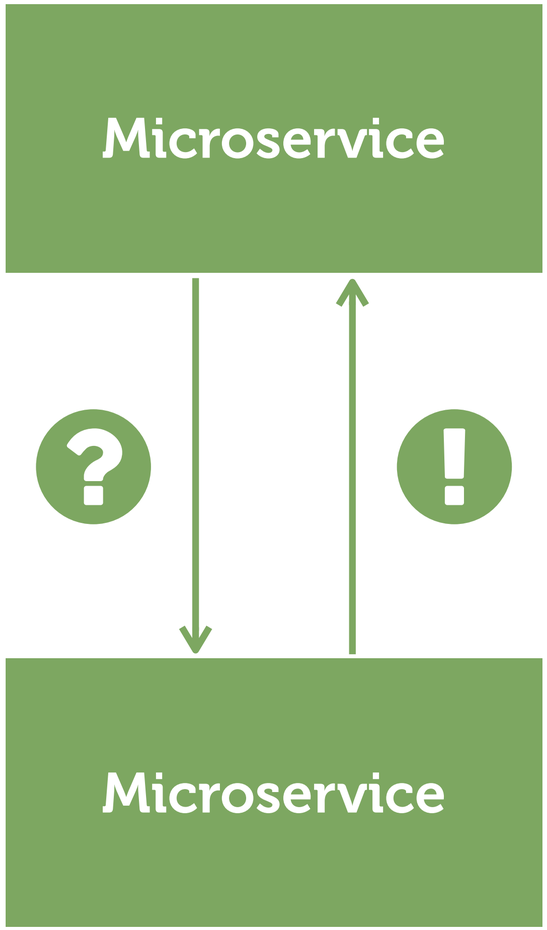
Fire & Forget

Example



REST

- F&F doesn’t fit naturally
- Which HTTP method to use?
- Requests to create side effects
  - DELETE, PUT, POST

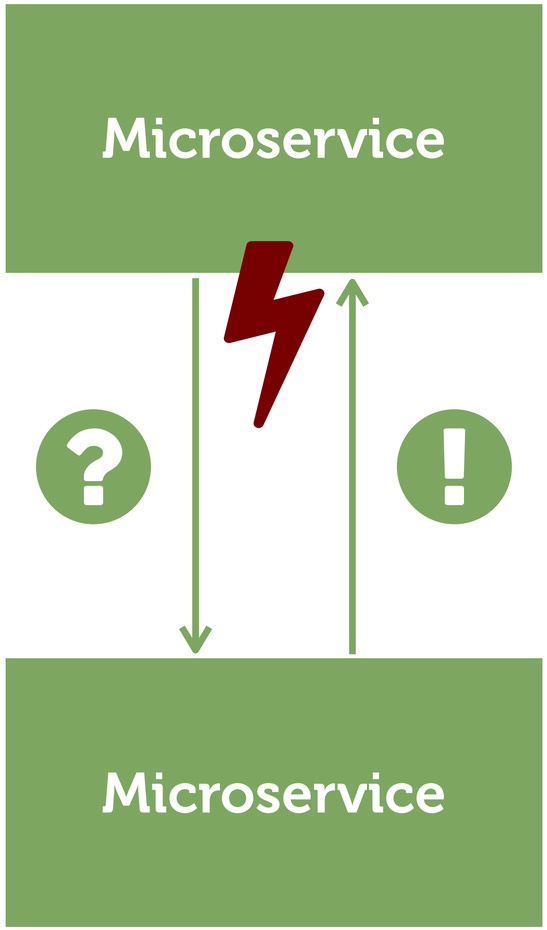


Safety / Idempotency

HTTP Method	Safe	Itempotent
GET	✓	✓
PUT	✗	✓
DELETE	✗	✓
POST	✗	✗
PATCH	✗	✗

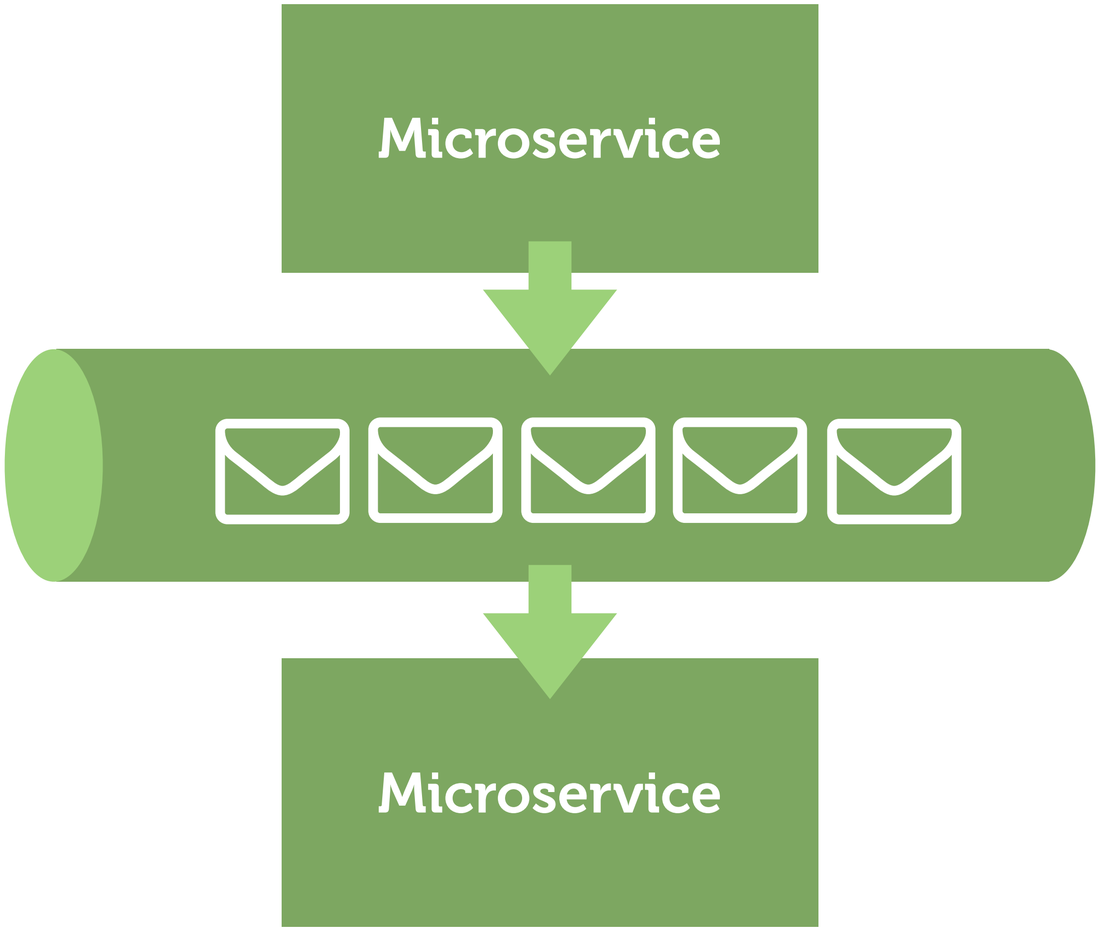
REST: Failure

- Remote system unavailable
  - Can’t easily retry because of non-itempotency
- Status codes to communicating semantic problems



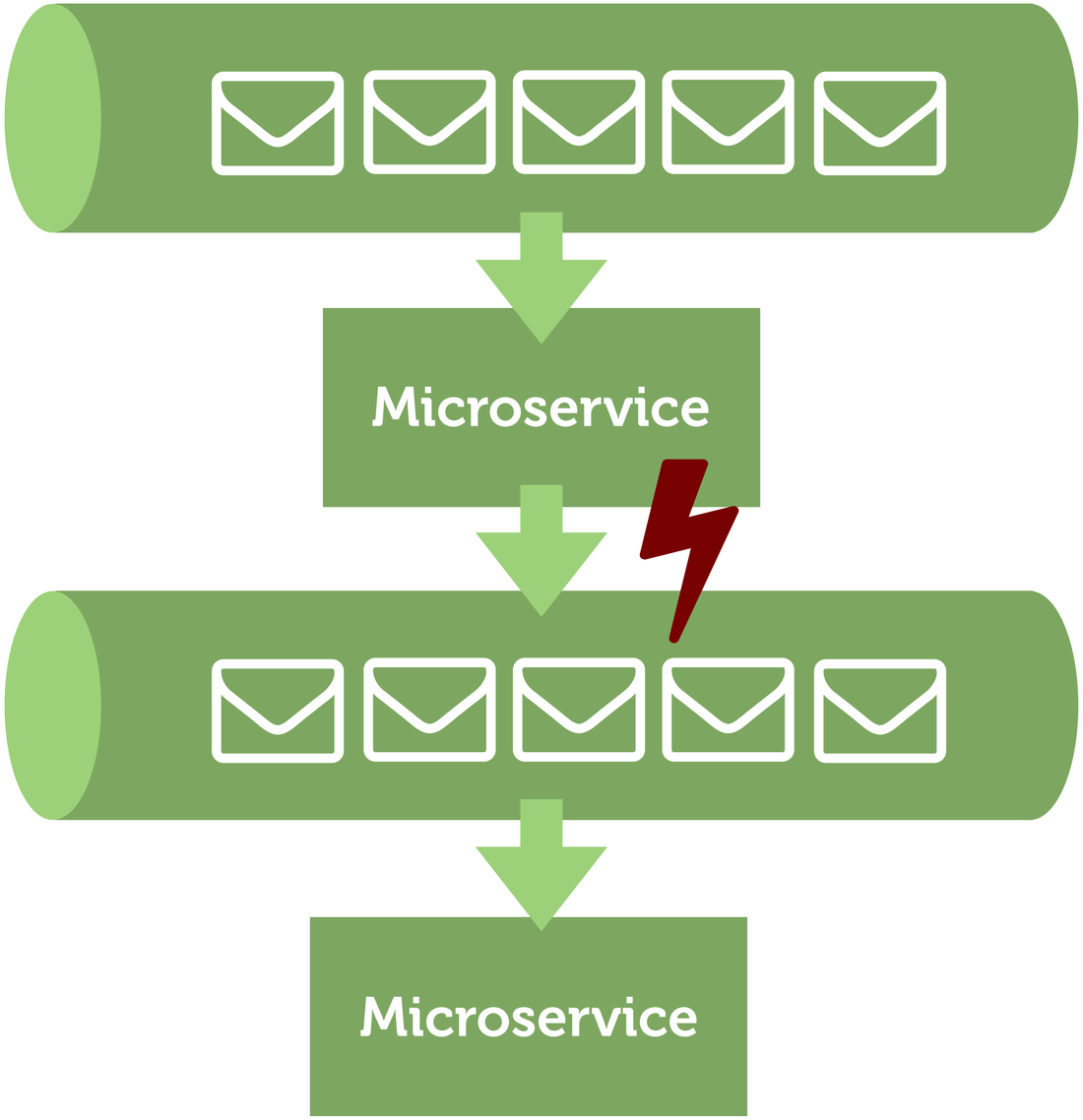
Messaging

- Hand message over to messaging system
- Messaging system guarantees delivery
- Stores message
- Acknowledgement
- Might have duplicated messages



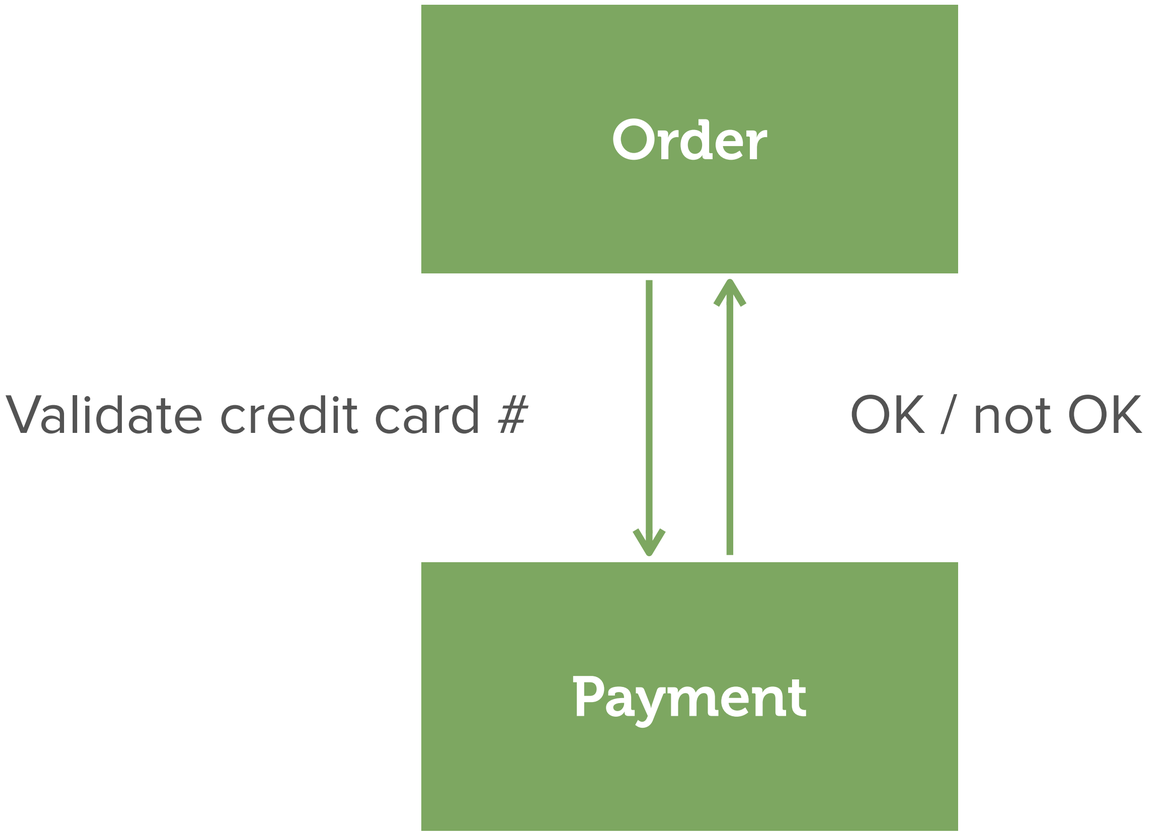
Messaging: Failures

- Message doesn’t make it into the message broker
- e.g. Timeout / TCP problem
- Retry
- Rely on re-transmission of incoming message



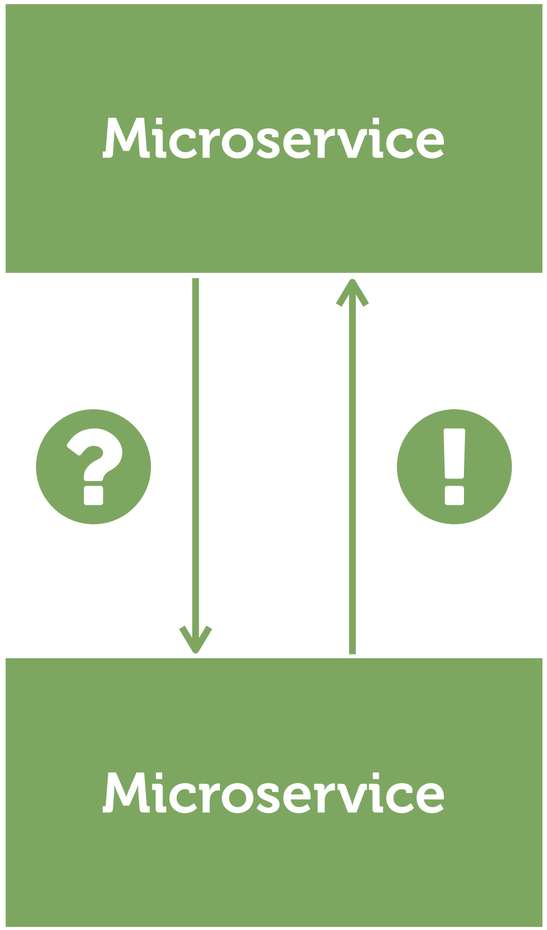
Request & Reply

Example



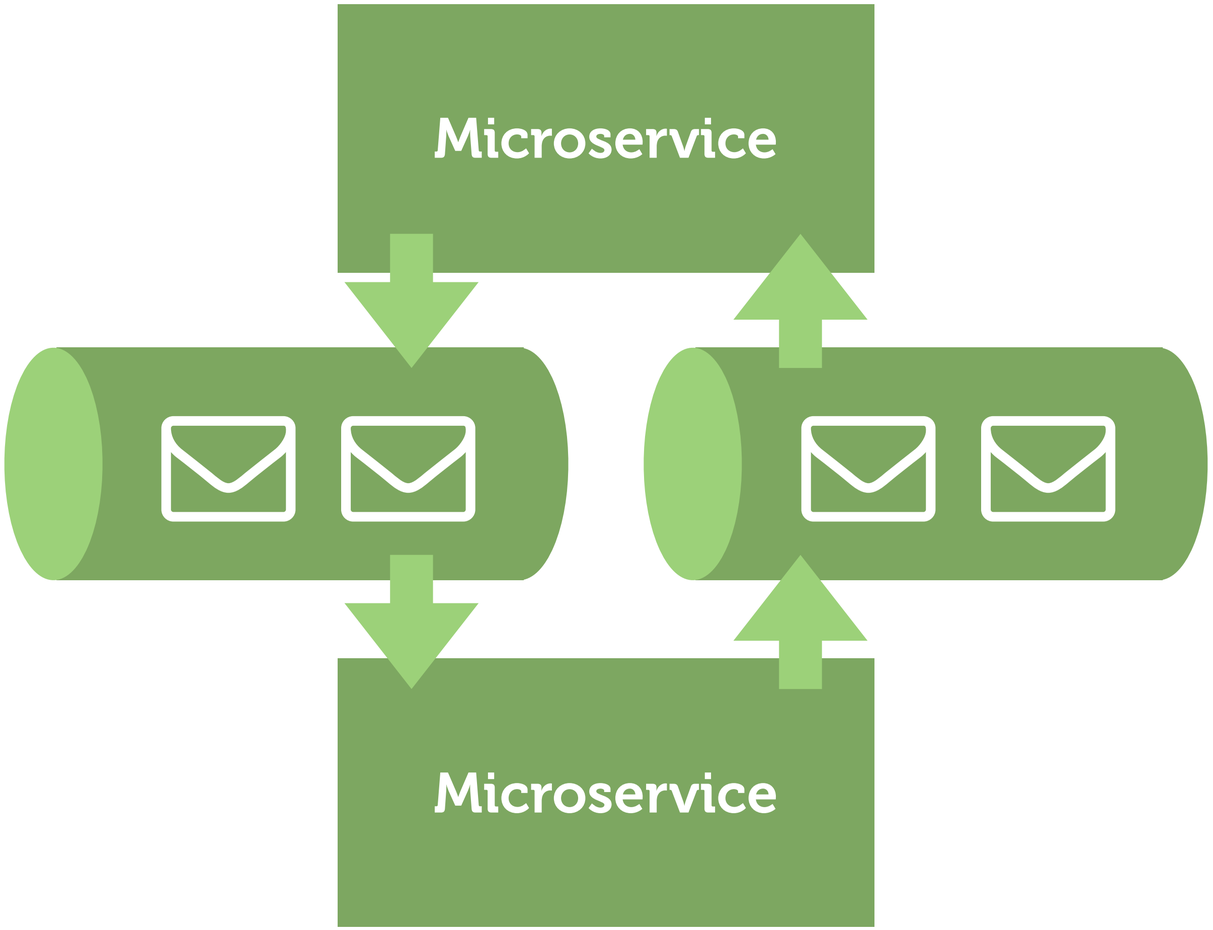
REST

- Natural model
- GET request
- Support for caching built in
  - ETags, Last-Modified, conditional GET / PUT
- Still needs care
  - Timeouts, resilience



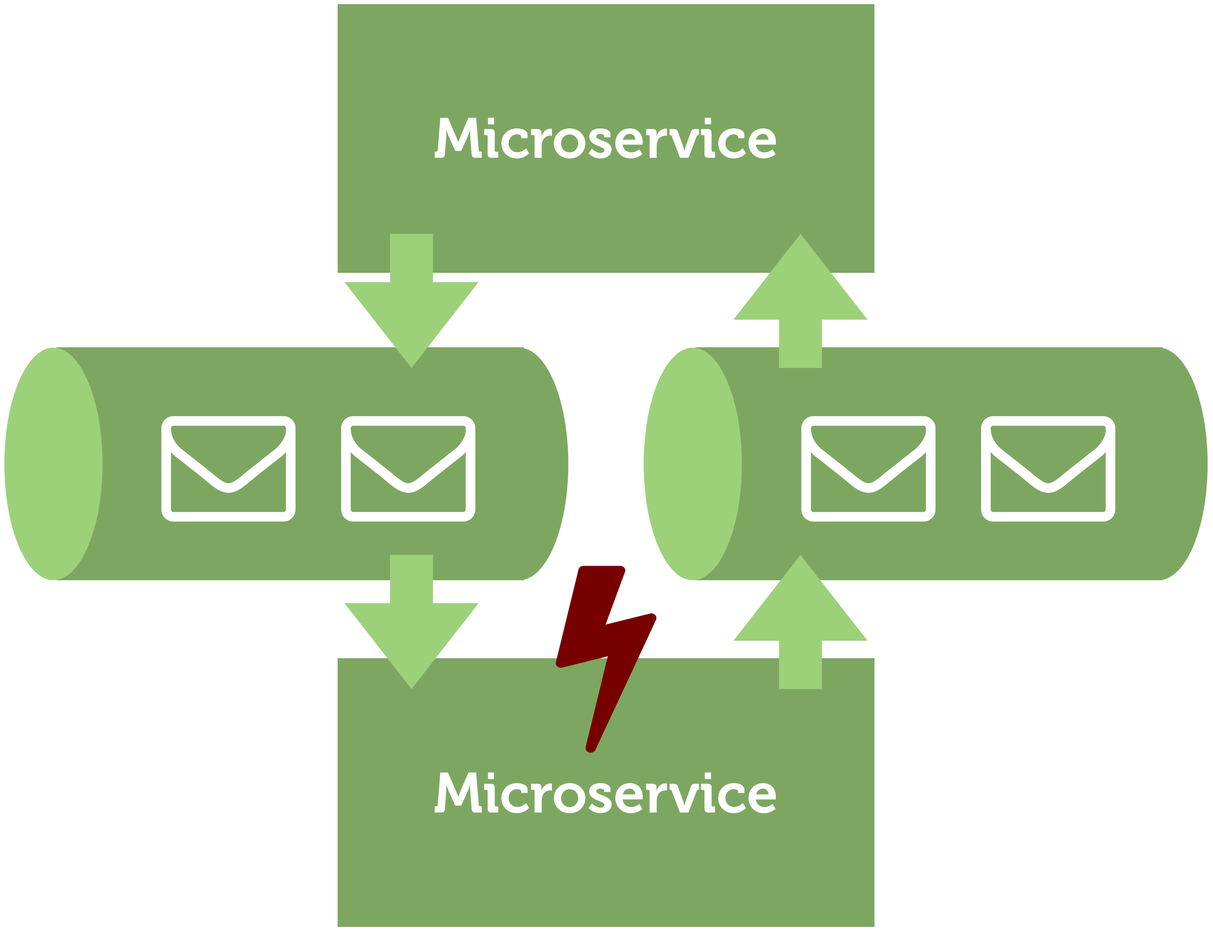
Messaging

- Send request
- Expect response
- Correlation
  - …or temporary queue
- Asynchronous by design



Resilience

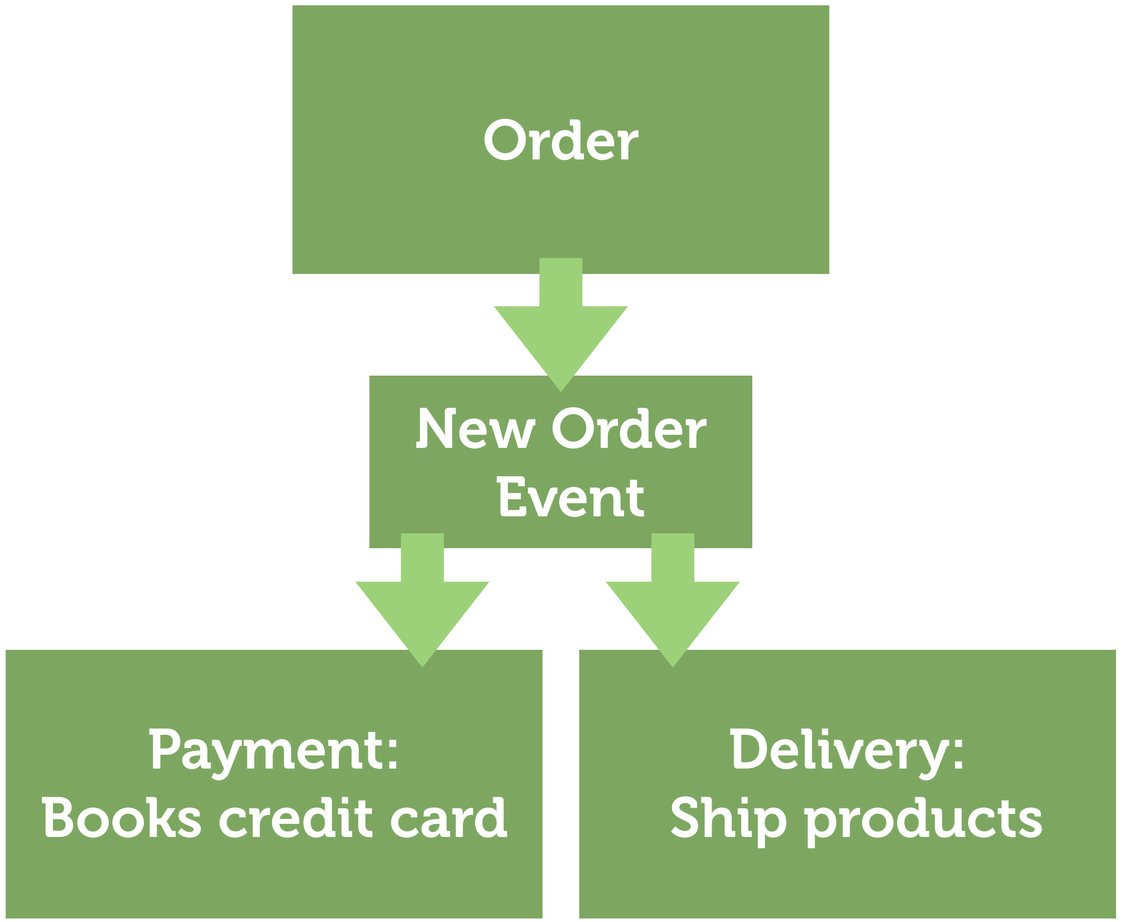
- Messaging can guarantee delivery
- Failure just increases latency
- System must deal with latency anyway



Events

Event Driven Architecture

- Order sends events
- Decoupled: no call but events
- Receiver handle events as they please



Event Driven Architecture

- System are built around publishing domain events
- Multiple event listeners
- Event listener decides what to do
- Can easily add new event listener with additional business logic
- Challenges
  - Delivery hard to guarantee
  - What about old events?

Events + REST = Feed

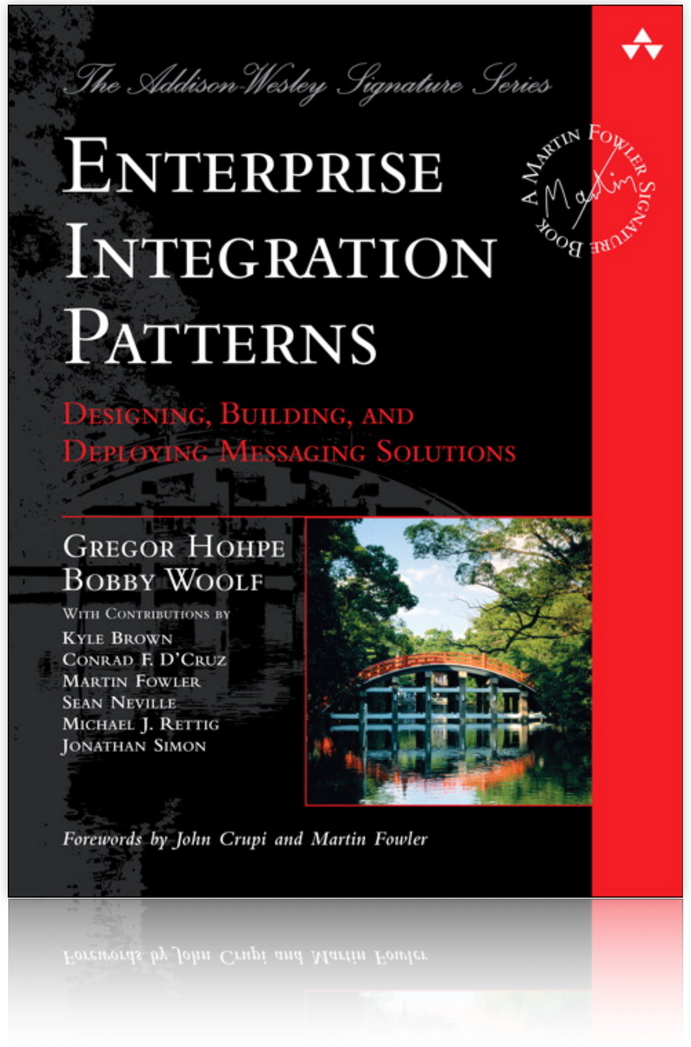
- System stores domain events and publishes feed (e.g. Atom)
  - Strong consistency within the service
  - No additional infrastructure required
  - Getting closer to Event Sourcing
- Clients subscribe to feed
  - Clients in charge of polling frequency
- Server side optimizations: caching, ETags, pagination, links
- Client side optimizations: conditional requests

Messaging

- Publish / Subscribe e.g. JMS Topics
- History of events limited
- Guaranteed delivery somewhat harder

More Decoupling

- Enterprise Integration Patterns (Hohpe, Woolf)
- www.eaipatterns.com
- Contains patterns like Router, Translator or Adapter
- Create flexible messaging architectures



Code

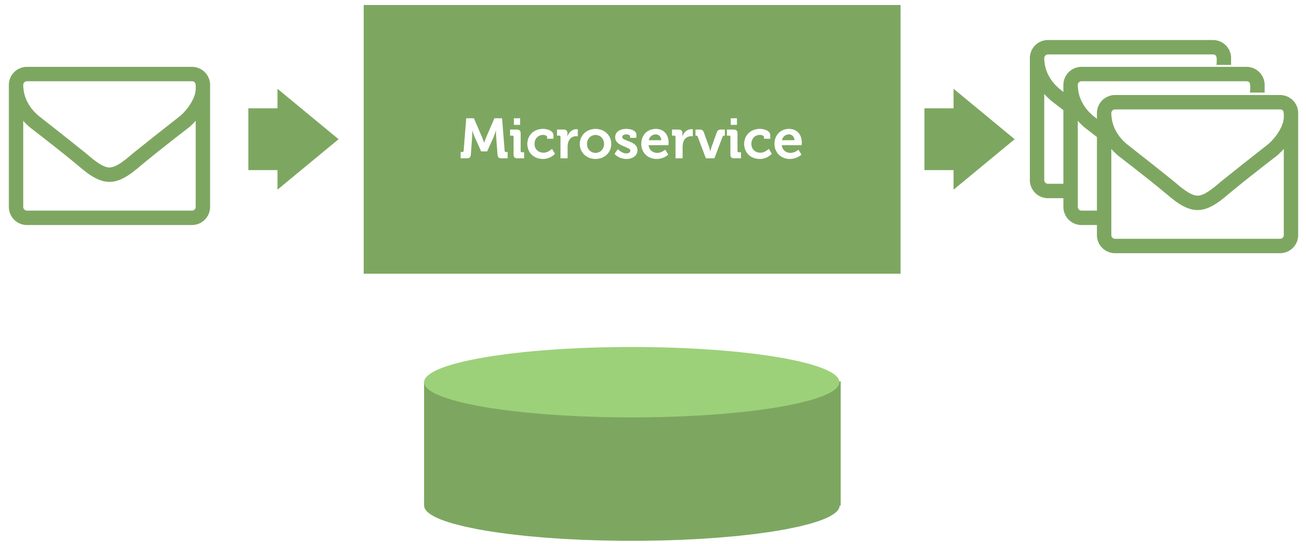
```
@Inject OrderRepository repository;

@Transactional
public void order(Order order) {
  repository.save(order.deliver());
  doCreditCardBooking(order.getCcNumber());
}
```

Transactions

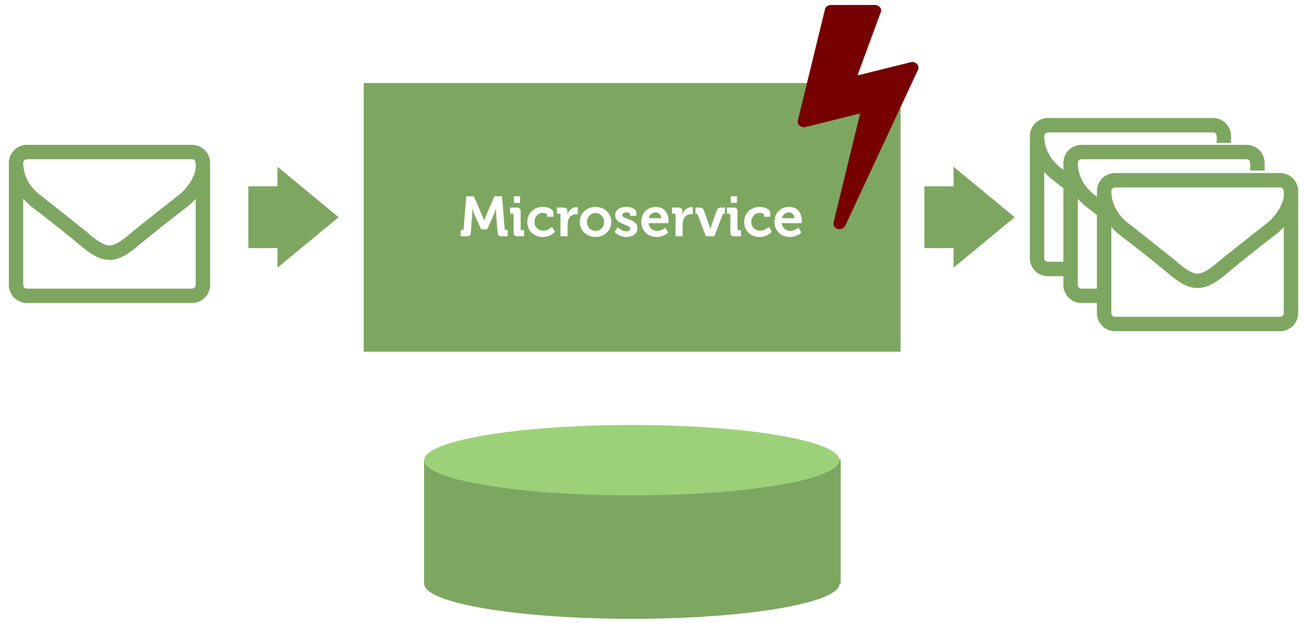
Messaging & Transactions (Commit)

- Database commit
- Incoming messages acknowledged
- Commit success: outgoing messages sent
- Outgoing messages hopefully handled successfully.
- Inconsistencies: Outgoing messages not yet processed



Messaging & Transactions (Rollback)

- Database rollback
- Outgoing message not sent
- Incoming message retransmitted



REST & Transactions

- No implicit infrastructure support
- But can be built manually

REST & Transactions

```
@Inject OrderRepository repository;
@Inject ApplicationEventPublisher publisher;

@Transactional
public void order(Order order) {
  repository.save(order.deliver());
  publisher.publish(new OrderDeliveredEvent(order));
}

@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
public void onOrder(Order order) {
  doCreditCardBooking(order.getCcNumber());
}
```

Evolvability

Evolvability

- Core aspect of Microservices: independent deployability
- Means: decoupling
- Change in one system must not break downstream systems

REST

- Core concepts built into the protocol
- Representations
  - Content negotiation
  - Media types
- Hypermedia
  - Discoverability

Messaging

- Data format: Your choice
  - i.e. easy to evolve if changes backwards-compatible
- But: no support for content negotation

Summary

	REST	Messaging
Communication style	synchronous	asynchronous
Service Discovery	DNS, Service Registry Resource Discovery	Message Broker Queues / Topics
Strengths	Content negotiation, Hypermedia More control over direct interaction	Messages in Re-submission of messages

Your next project: Messaging or REST?

You'll probably use both :-)