LearnQuest

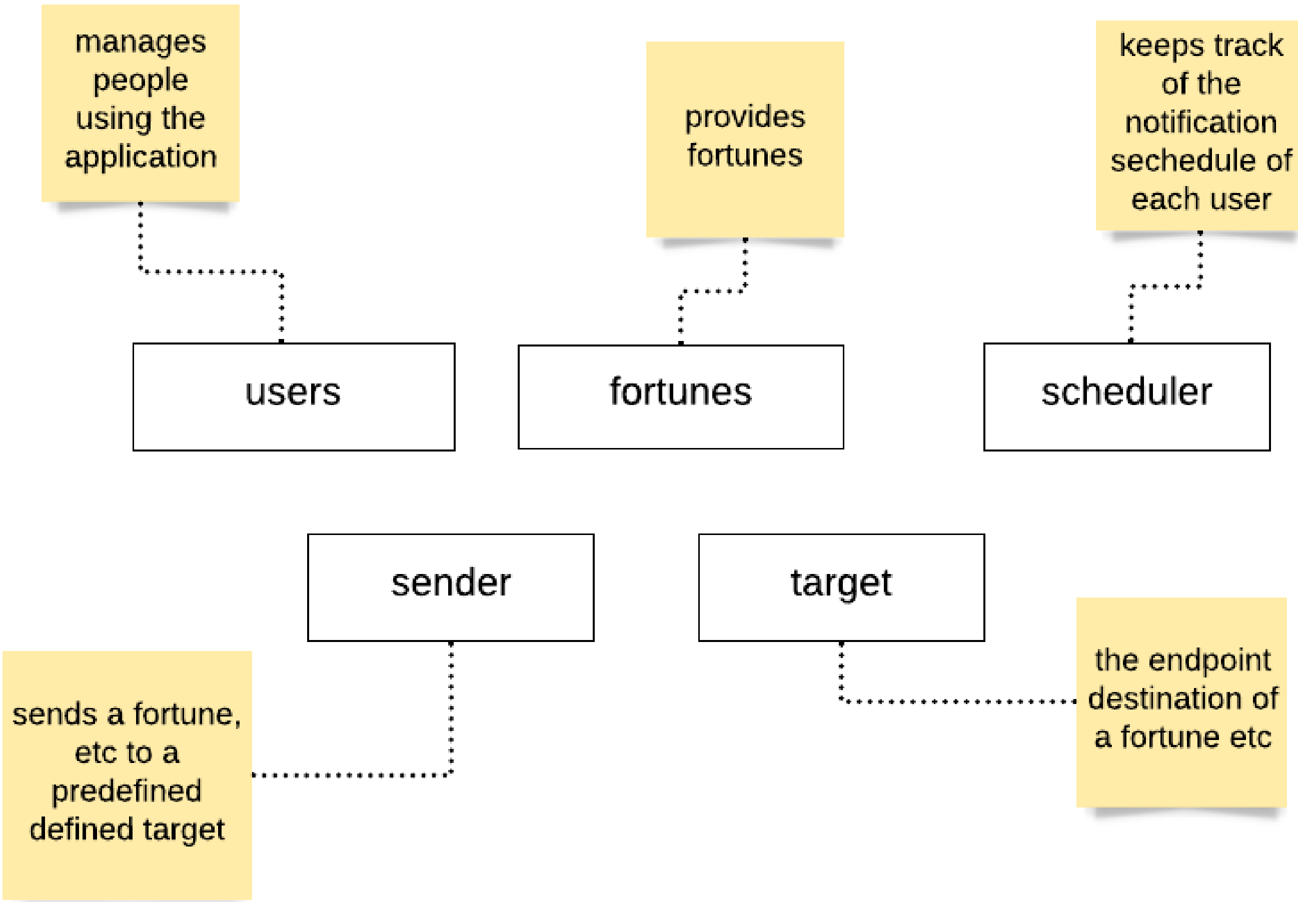
Presents

Microservice Patterns

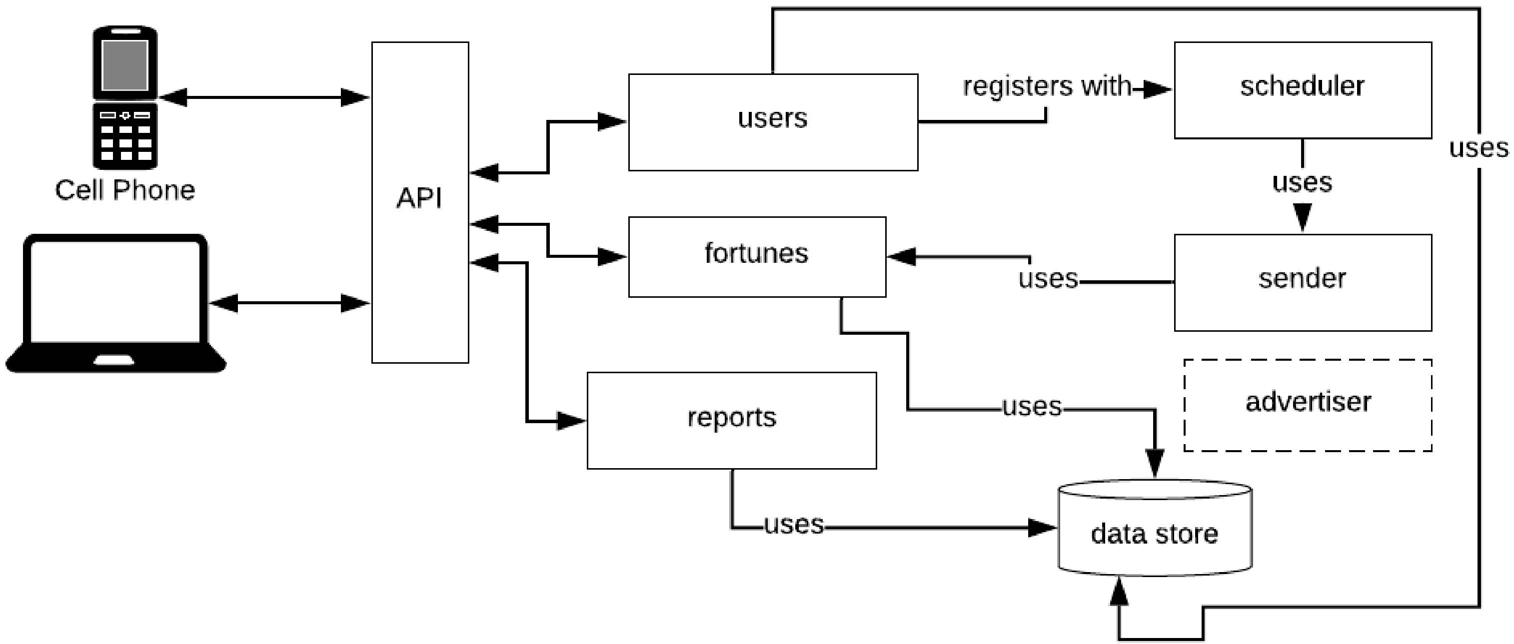
Microservice Styles

- Synchronous
- Asynchronous
- Hybrid

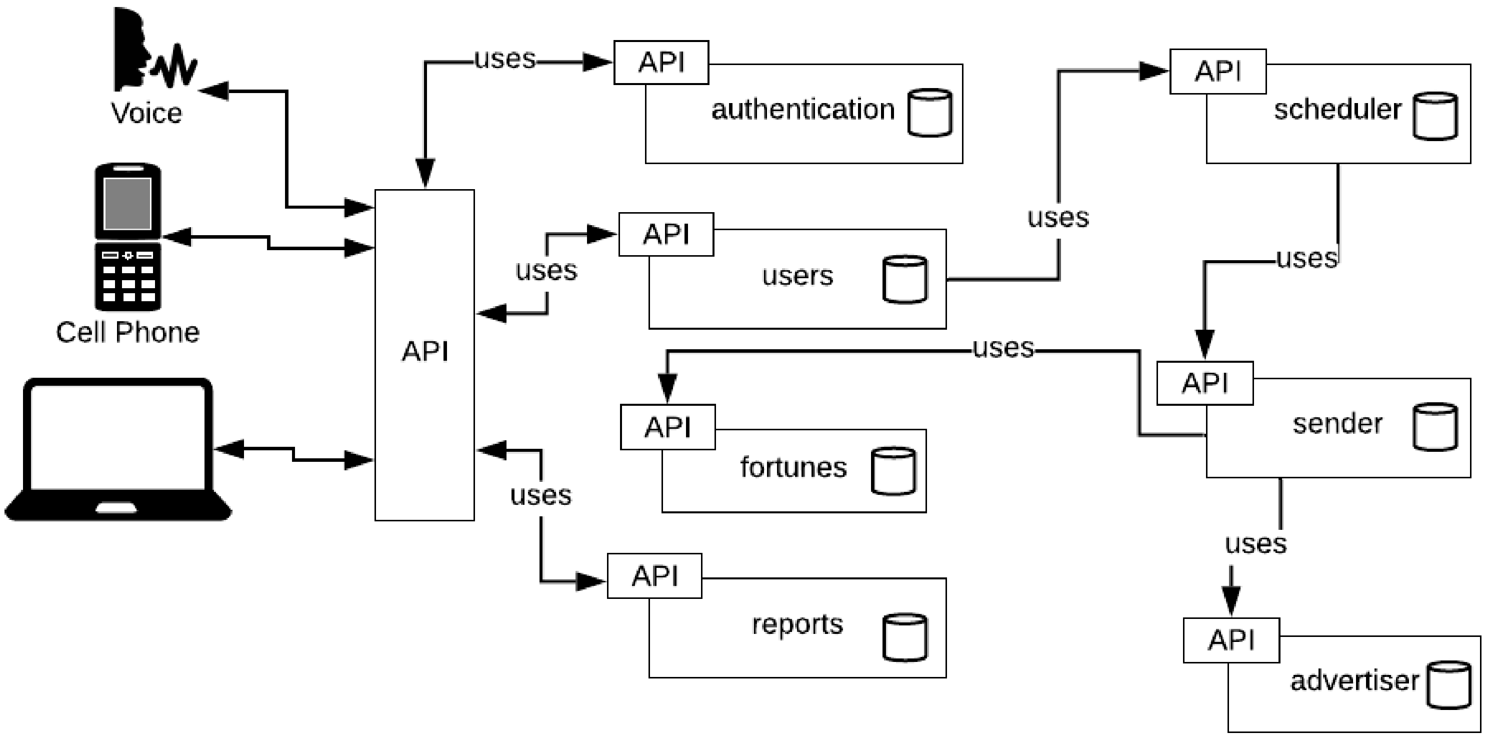
The Application Components



Monolithic Application



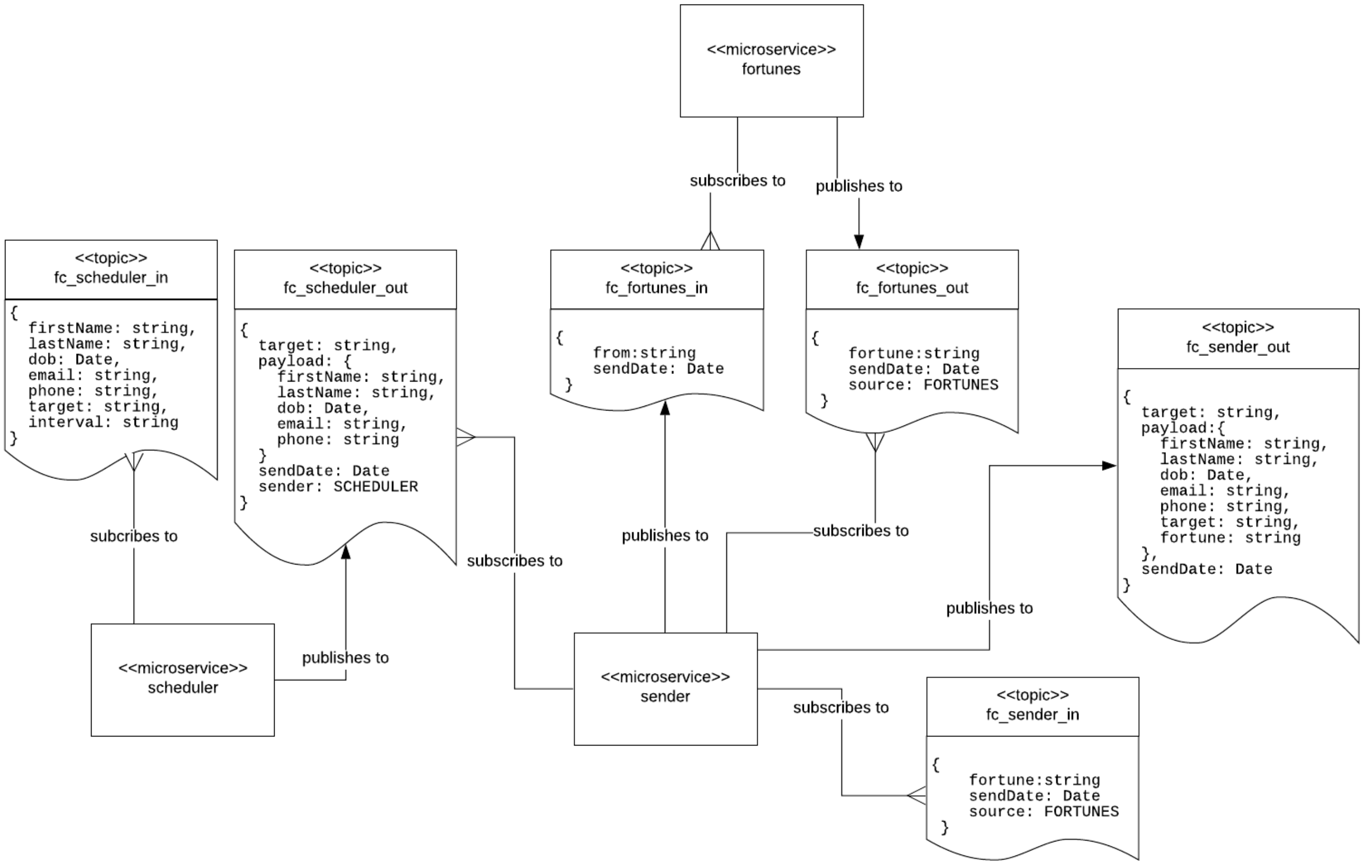
Microservices: Synchronous



Microservices: Synchronous

- Advantages:
  - ✓ Service is stateless (relative to caller)
  - ✓ Allows many active instances
  - ✓ Lower management overhead since no message server
  - ✓ Immediate and direct feedback (success or failure)
- Disadvantages:
  - ✓ Caller must wait, potentially limiting caller scalability
  - ✓ Can add hard dependencies between services if chained

Microservices: Asynchronous

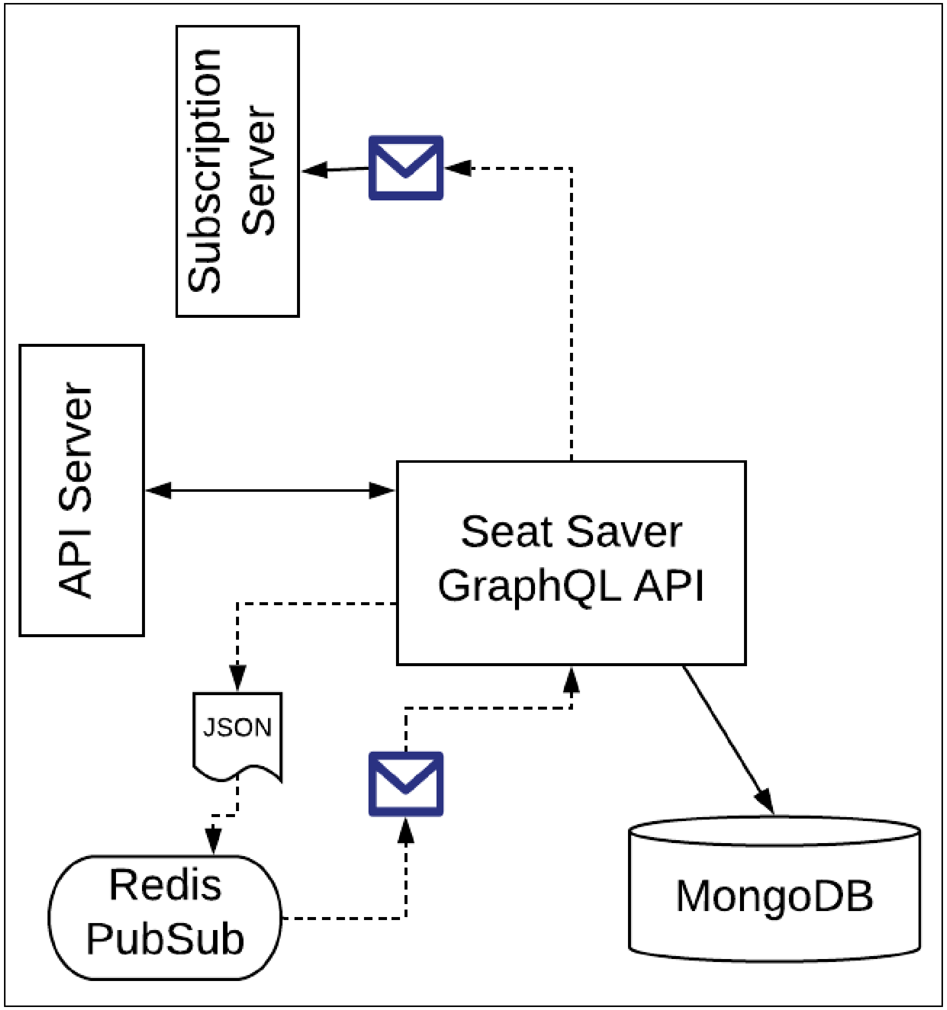


LearnQuest

Microservices: Asynchronous

- Advantages:
  - Improves scalability since no coupling between caller and service instance
  - Queueing allows for degree of inherent load balancing
- Disadvantages:
  - Dependent on external message server
  - Additional tuning and monitoring is required

Microservices: Hybrid



Microservices: Hybrid

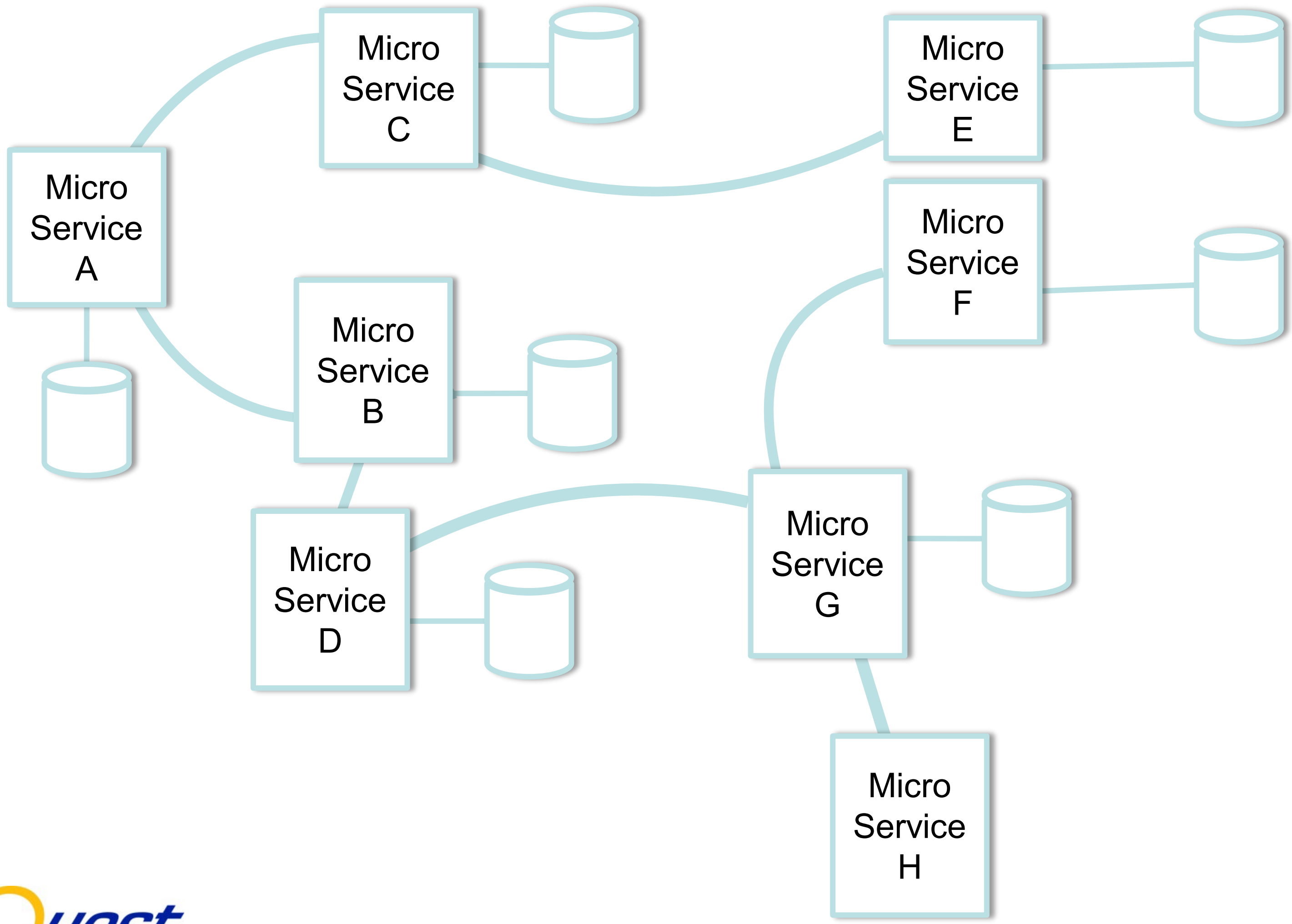
- Advantages:
  - ✓ Best of both worlds
  - ✓ Flexibility in terms of developer experience
  - ✓ Easy to implement as an API
- Disadvantages:
  - ✓ Hard to implement as a microservices
  - ✓ There’s no free lunch, you must contend with more latency issues

Challenges Building Microservices

- Even the simple applications become very complex
- Requires teams that are autonomous and cross functional
- Requires support for the principles of DevOps throughout the organization, e.g., Infrastructure as code
- Infrastructure management is paramount

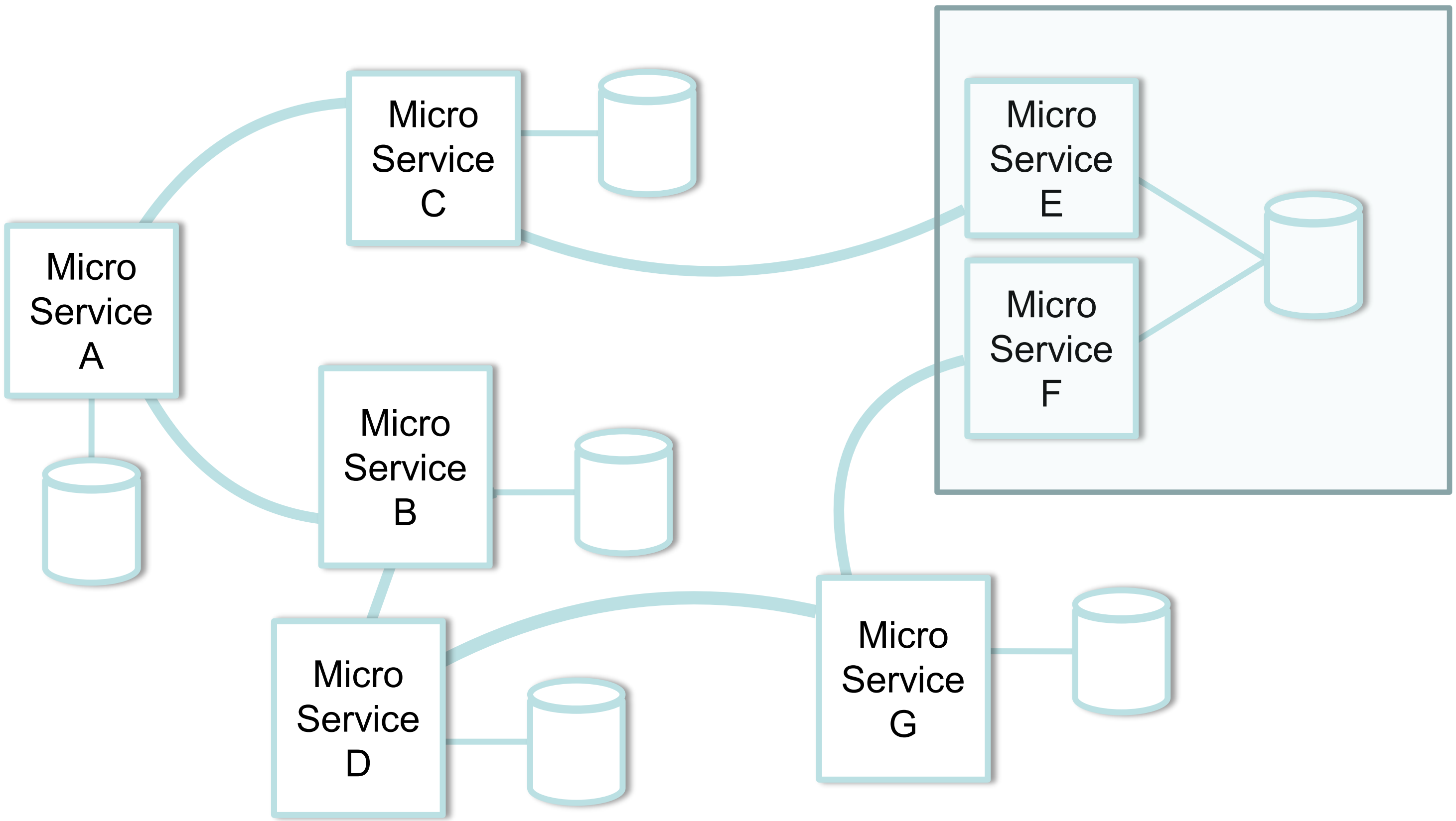
Data Patterns

Database Per Service



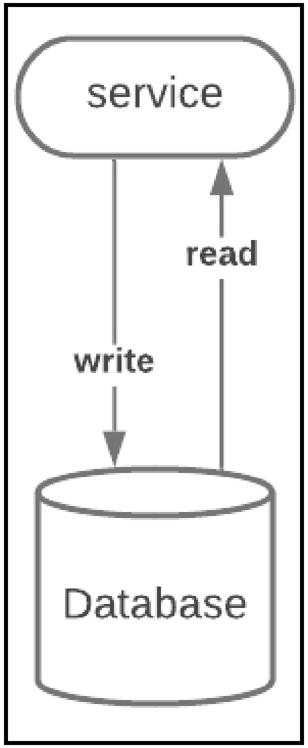
LearnQuest

Database Per Service



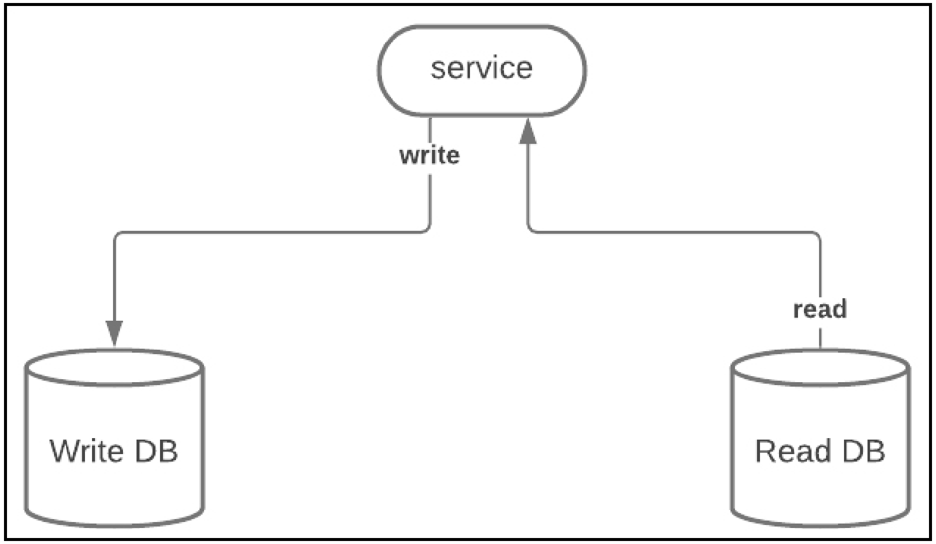
CQRS

- Command-Query Responsibility Segregation
- CQRS pattern separates read and update operations for a data store
  - ✓ "Asking a question should not change the answer"
- Addresses issues with reading and writing to a single DB
  - ✓ Typically, there are more reads than writes
  - ✓ Optimization techniques are different for reads and writes
  - ✓ Writes will cause side effects in read behavior



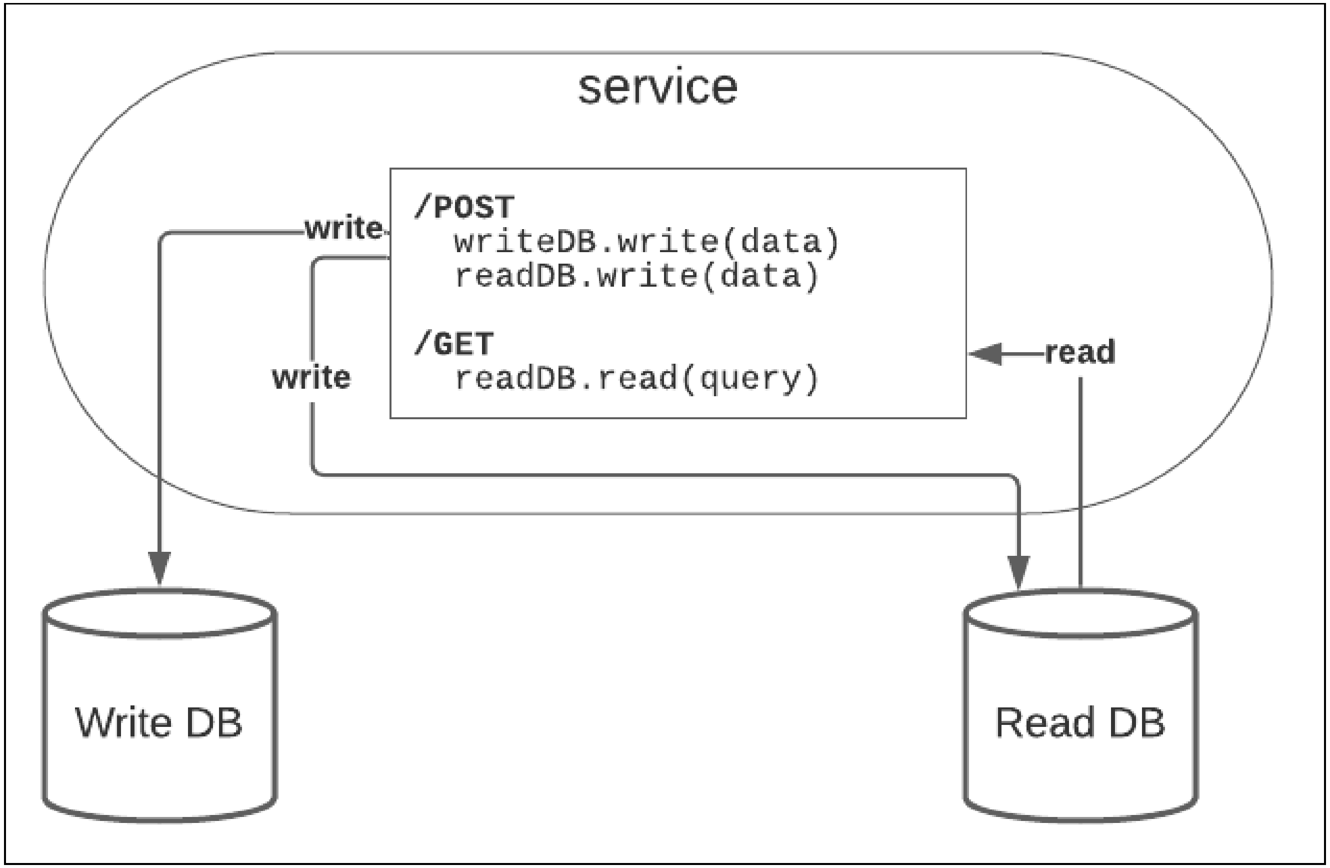
CQRS

- Separating reads and writes improves performance but
  - ✓ The problem of synchronization now needs to be addressed



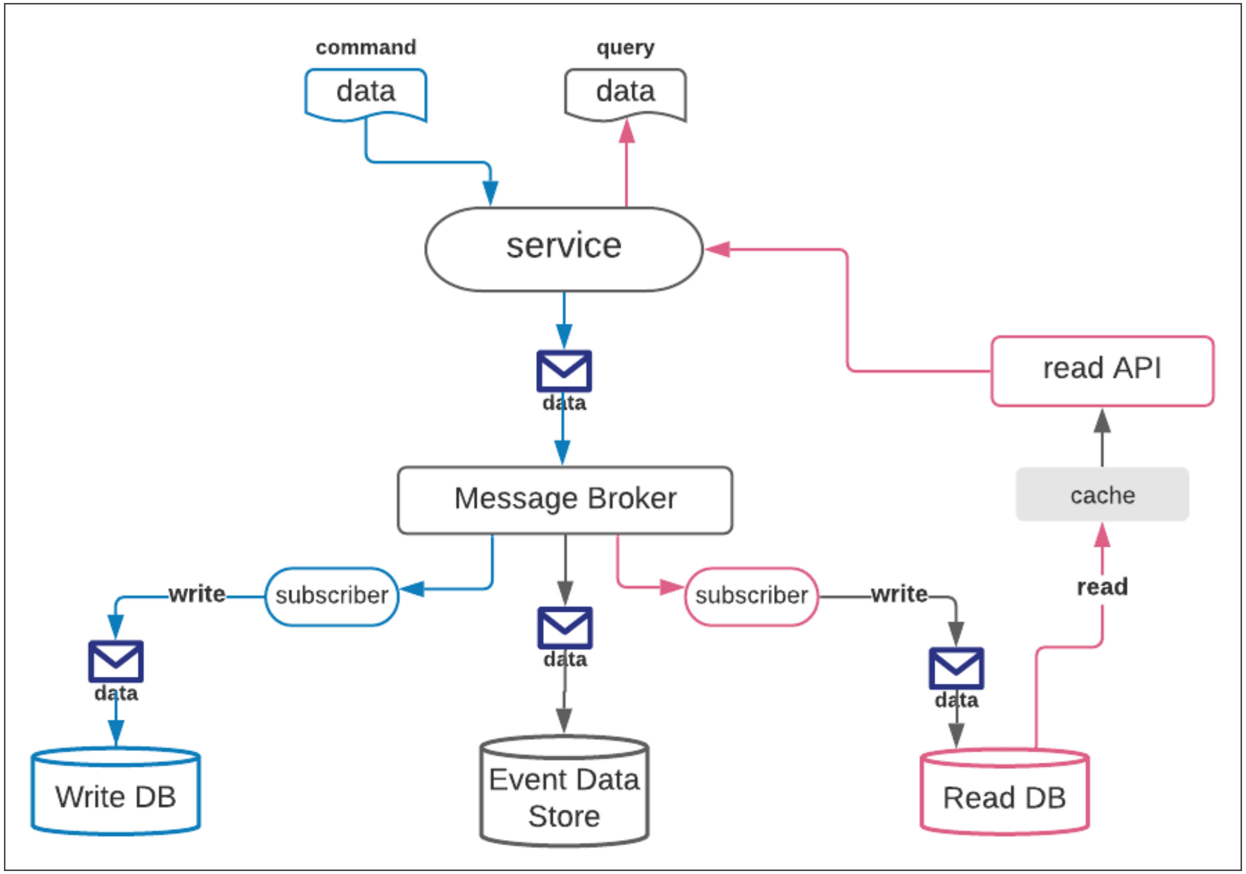
CQRS

- Different read and write services can use different technologies



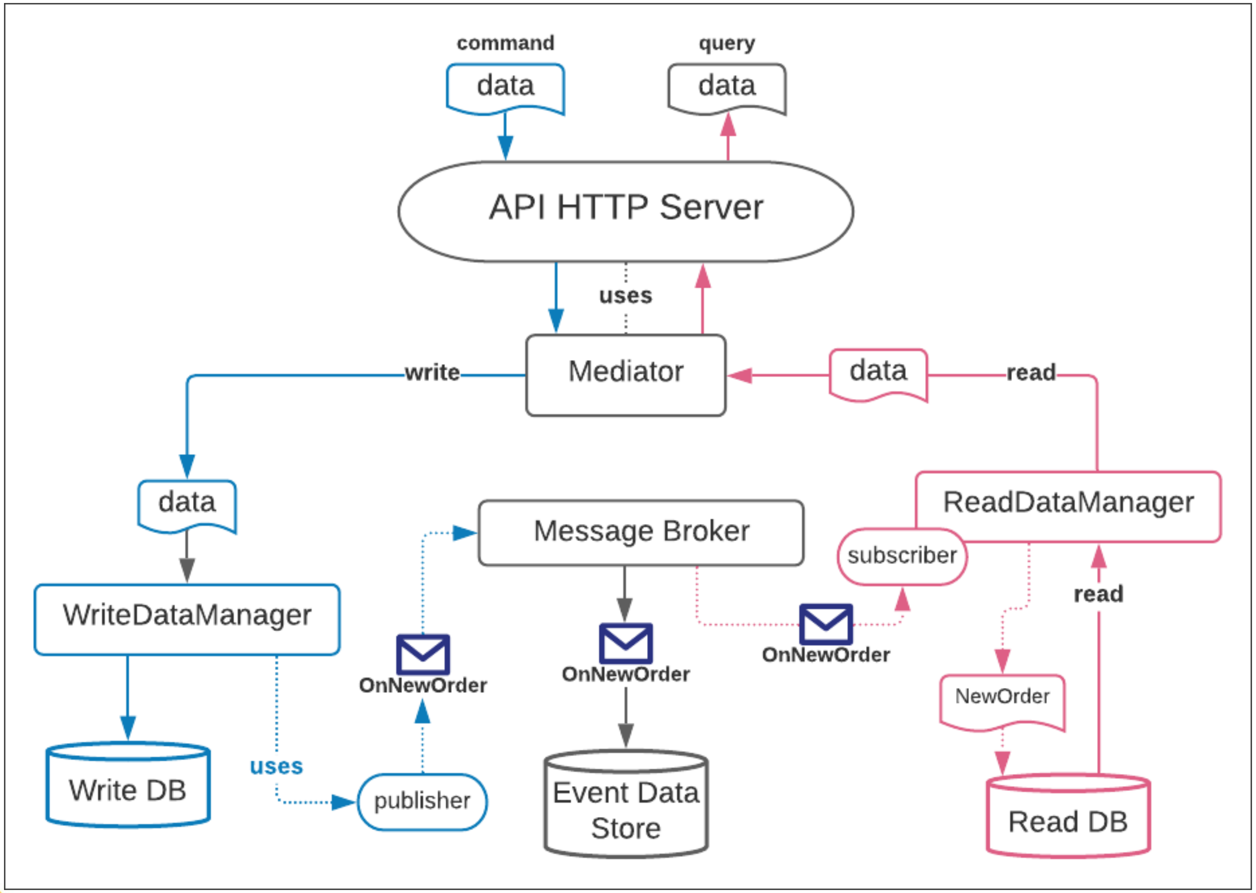
Working with CQRS

- Using a message driven architecture addresses many of the risks

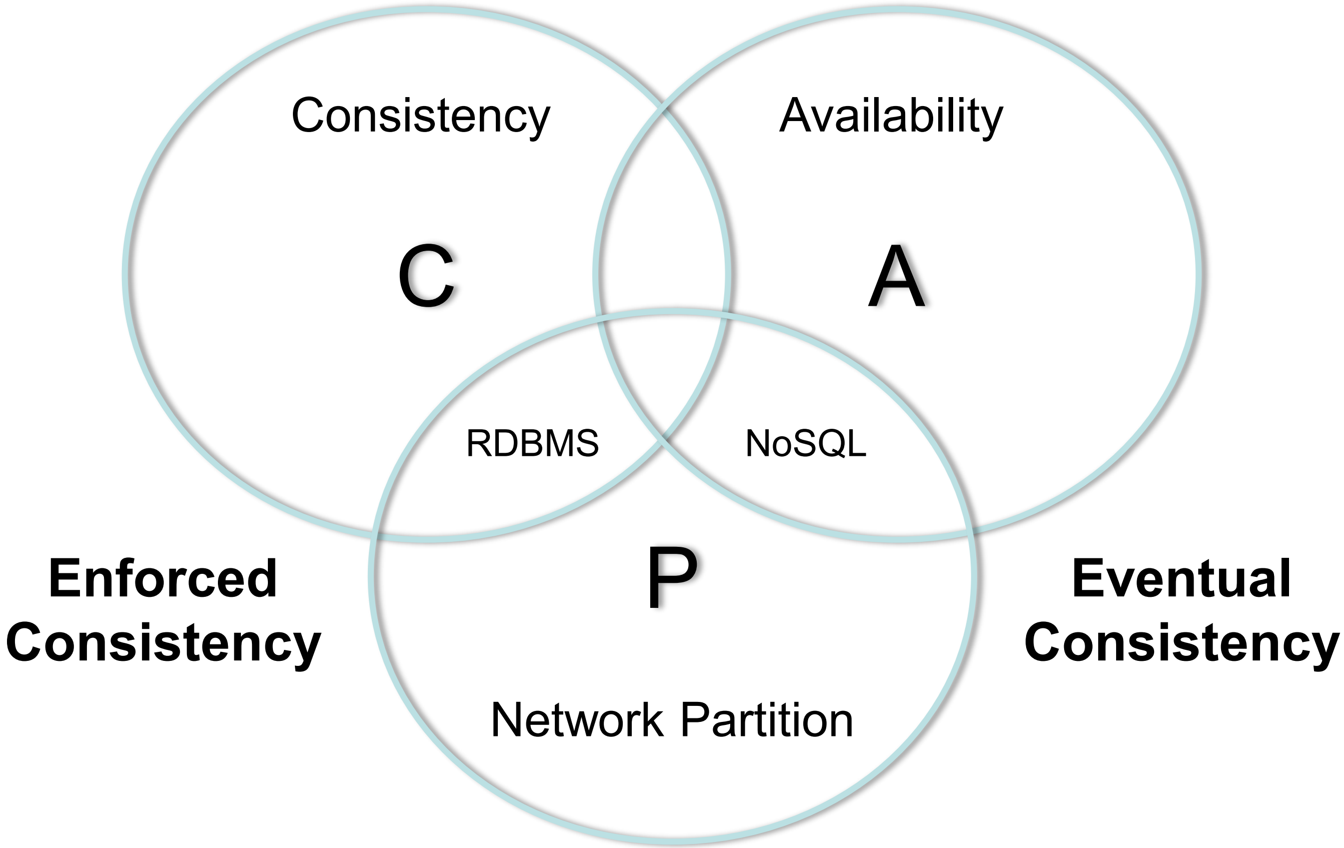


Using a Mediator

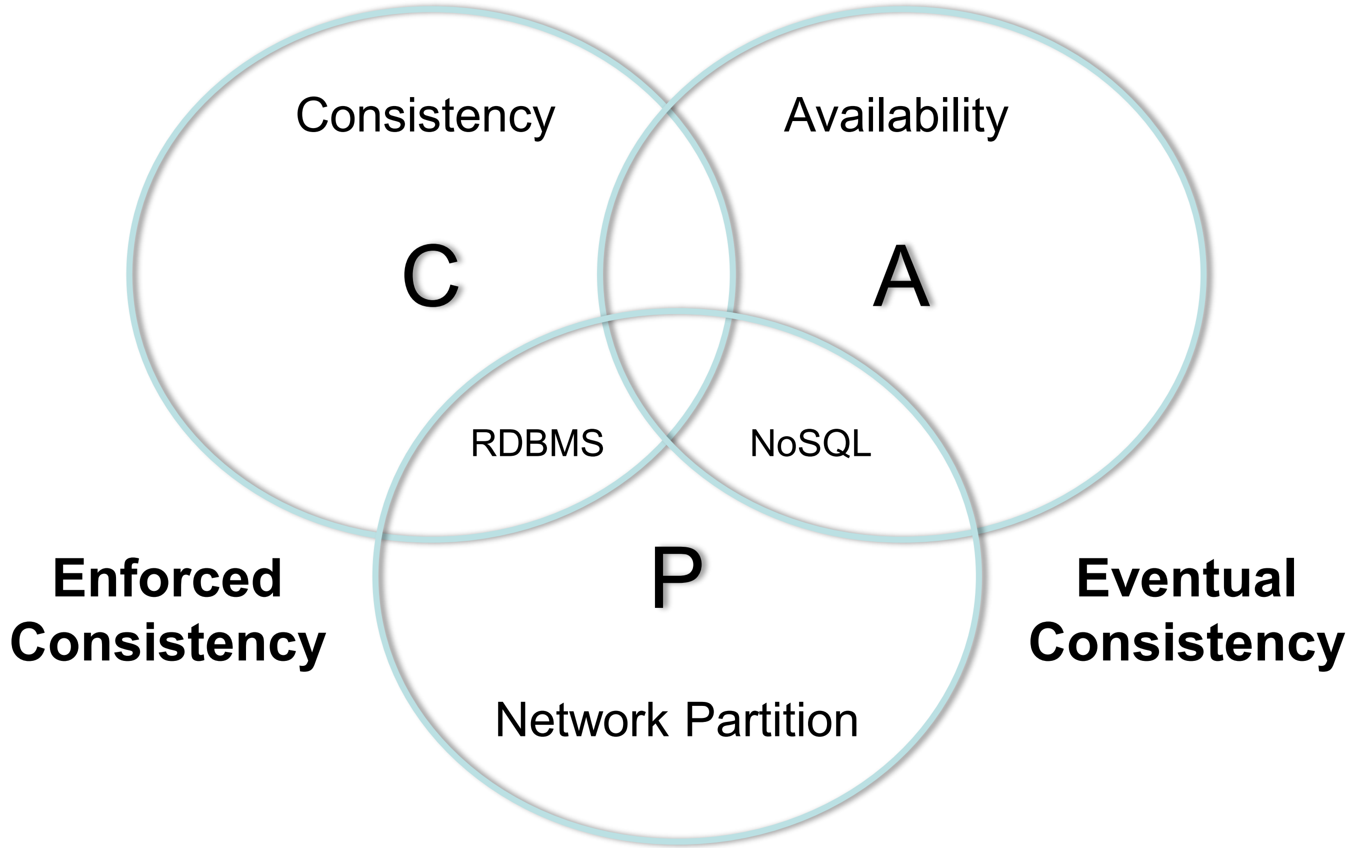
- Using the Mediator pattern reduces the risk of tight coupling



The Cap Theorem



The Cap Theorem



LearnQuest

Understanding the CAP Theorem

- When data is distributed among many data sources then:
  - ✓ You can have inaccurate date available immediately; or
  - ✓ You can have accurate data available eventually
  - ✓ You cannot have accurate data available always

Consistency Models

		Consistency	Performance	Availability
Strong	See all previous writes.	excellent	poor	poor
Consistent Prefix	See initial sequence of writes.	okay	good	excellent
Bounded Staleness	See all “old” writes	good	okay	poor
Monotonic Read	See increasing subset of writes	okay	good	good
Read Your Writes	See all writes performed by reader	okay	okay	okay
Eventual	See subset of previous writes.	poor	excellent	excellent

Consistency Models

- Strong Consistency
  - ✓ The data must be consistent at all times
  - ✓ All nodes everywhere should contain the same values at all times
  - ✓ Implemented by locking down the nodes when being updated
- Consistent Prefix
  - ✓ reads never see out of order writes
  - ✓ If writes were performed in the order A, B, C, then a client sees either A, A,B, or A,B,C, but never out of order like A,C or B,A,C
- Bounded Staleness
  - ✓ All observers have the same data at the same time
  - ✓ Writes may only be ahead of reads by a set number of ops or time lag

LearnQuest

Consistency Models

- Monotonic Read
  - ✓ Read operations do not return results that correspond to an earlier state of the data than a preceding read operation
  - ✓ For example, if in a session:
    - write1 precedes write2,
    - read1 precedes read2, and
    - read1 returns results that reflect write2
    - then read2 cannot return results of write1
- Read Your Writes
  - ✓ Guarantees that once a record has been updated, any attempt to read the record will return the updated value.

Consistency Models

- Eventual Consistency
  - ✓ Ensures high availability to all users
  - ✓ If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value

Transaction Patterns

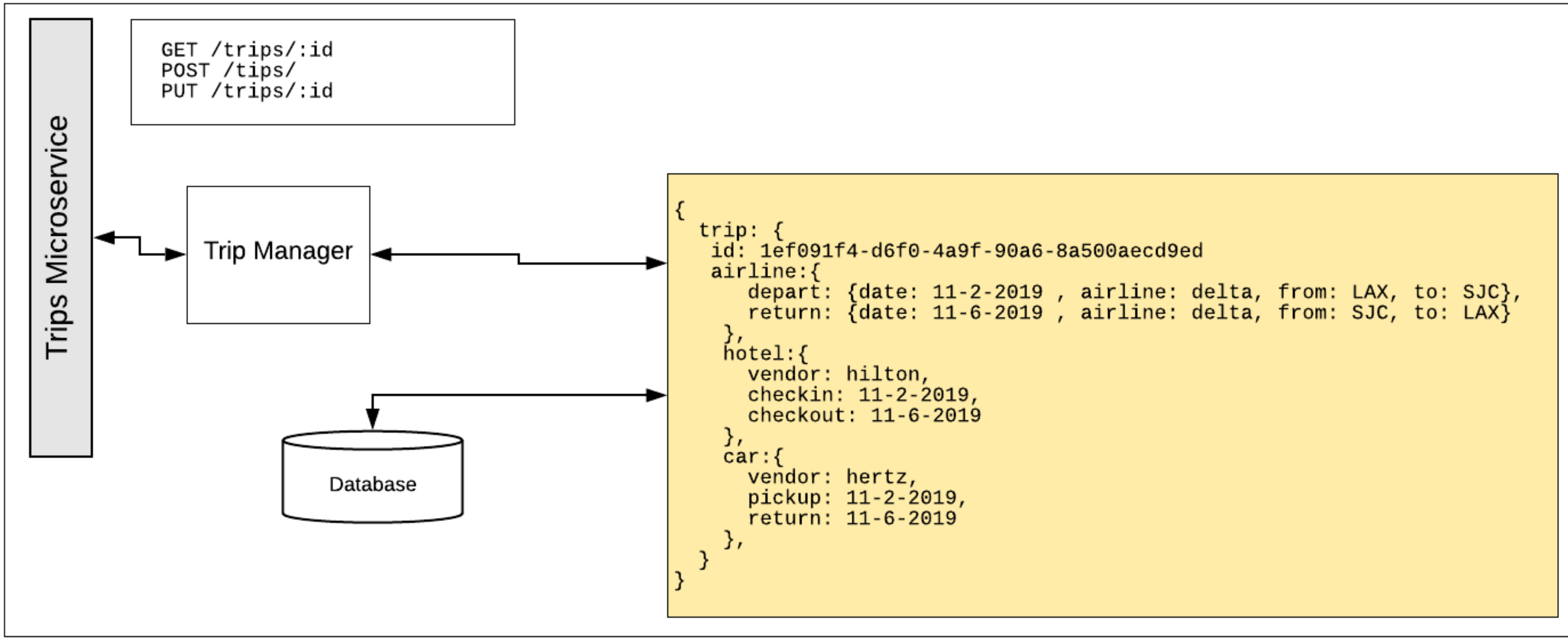
Data Management: Transactions

- Transaction design patterns
  - ✓ Same Bounded Context
  - ✓ State Store
  - ✓ 2 Phase Commit Protocol (2PC)
  - ✓ Routing Slip

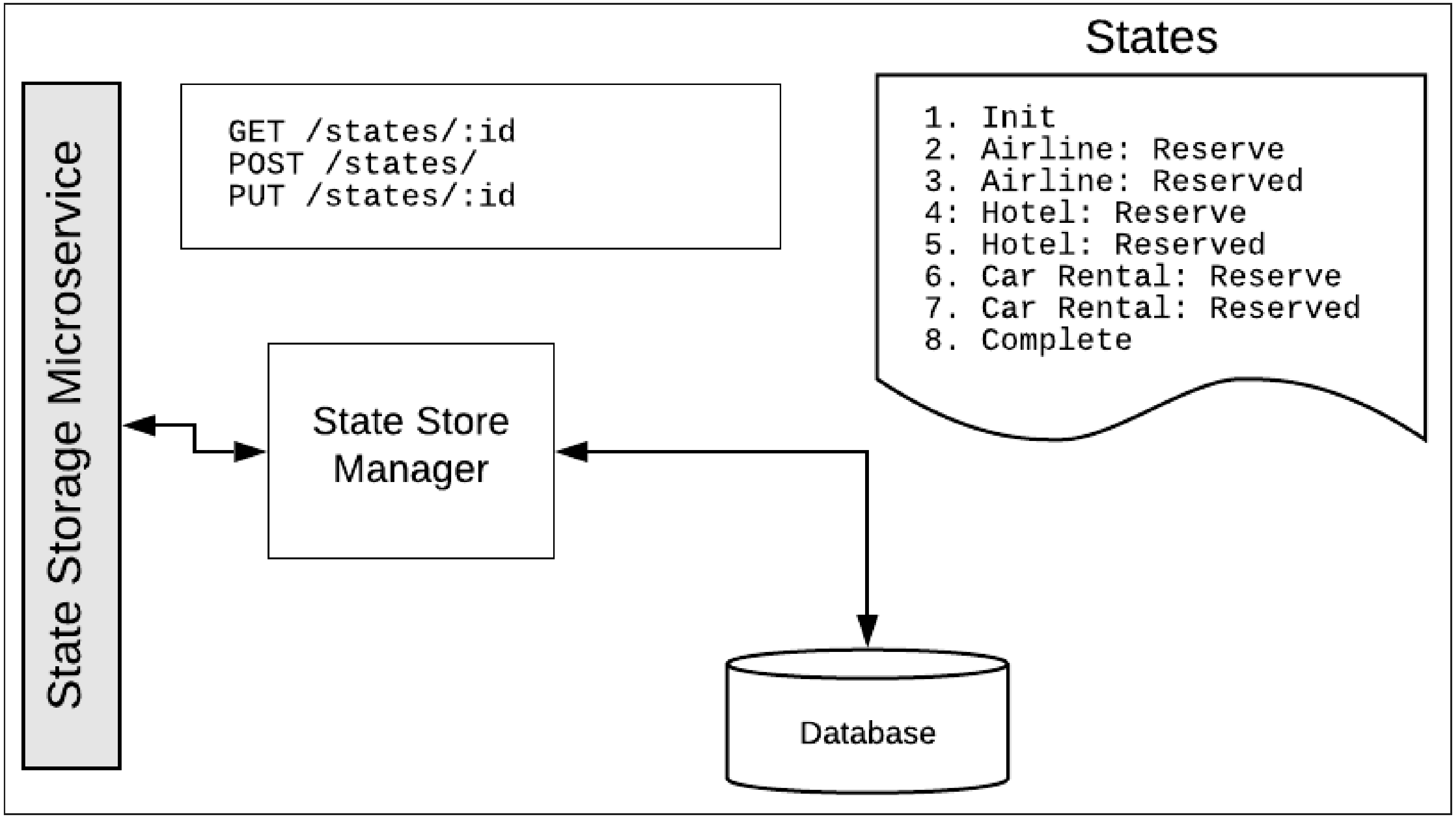
Implementing a transactions that occurs across several microservices residing in a variety of datacenters is always a challenge. The easiest thing to do is to avoid inter-service transactions completely.

Sometimes circumstances in the real world require supporting transactions. So, we need to do what the need demands. However, be advised supporting inter-service transactions is rarely an easy undertaking.

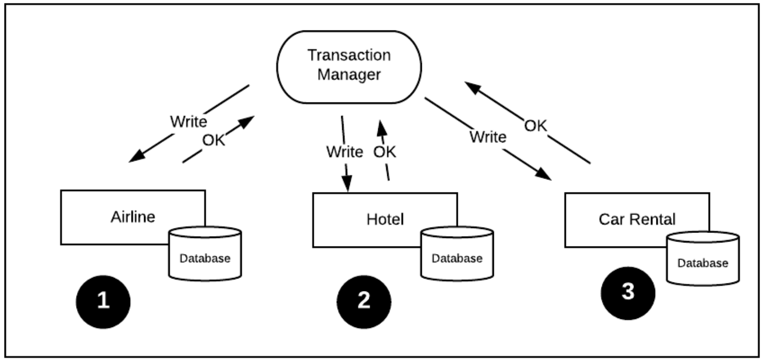
Same Bounded Context

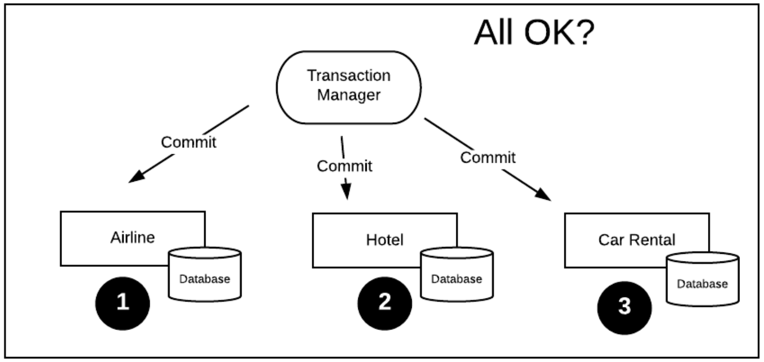


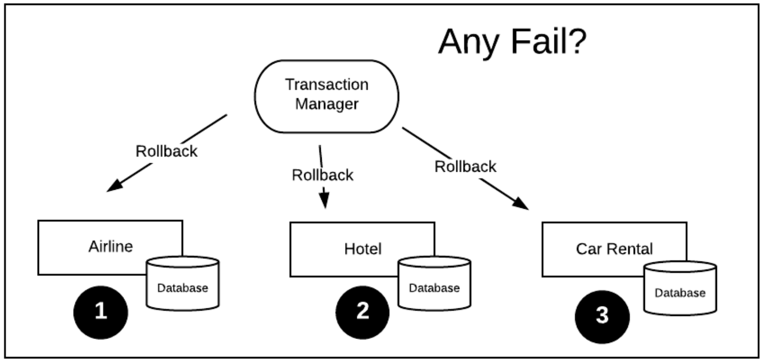
State Store


States
State Storage Microservice
GET /states/:id
POST /states/
PUT /states/:id
1. Init
2. Airline: Reserve
3. Airline: Reserved
4: Hotel: Reserve
5. Hotel: Reserved
6. Car Rental: Reserve
7. Car Rental: Reserved
8. Complete
State Store Manager
Database

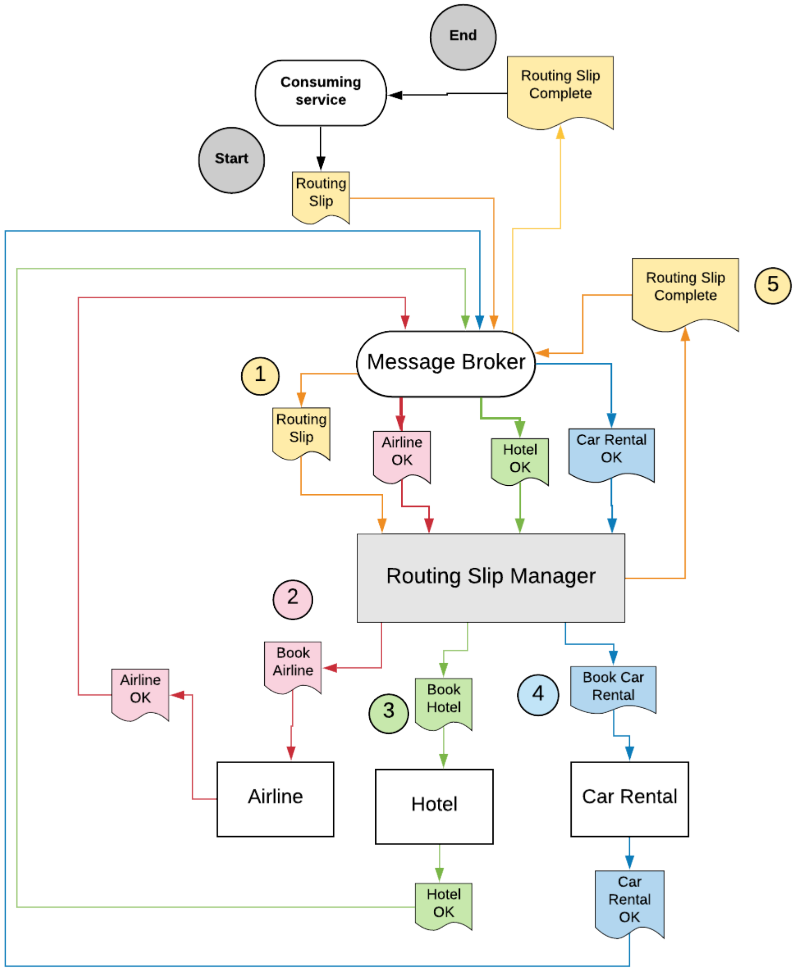
Two Phase Commit (2PC)







Routing Slip



LearnQuest

Contexts and Interfaces

Working in Domain: Context Boundary

- Context boundary is the concept of organizing data structures according to the semantics of the context in which the application or services operates
- Typically, you want to have a data structures that are consistent within the domain.
- However, sometimes data structures get become incongruent due to subtle differences in the way the service operates
- Keeping things consistent within the boundary of a context can become challenging particularly around the way data is structured within an event payload

LearnQuest

The Importance of Context

Documentation becomes essential for translation

Accounting is in USA

```
{
  person: {
    firstName: string
    lastName: string
    dob: date
   },
  address: {
    address_1: string
    address_2: string
    city: string
    state: string
    zip: string
  }
}
```

Shipping is by Region

```
{
  person: {
    givenName: string
    surName: string
    birthday: date
   },
  address: {
    address_1: string
    address_2: string
    city: string
    province: string
    postal_code: string
  }
}
```

LearnQuest

The Value of Schema Definition

A well-known schema makes exchanging data between services easier

One way to manage schemas is to use a Schema Registry

```
message Customer {
    string firstName = 1;
    string lastName = 2;
    string email = 3;
    string created = 4;
    string message = 5;
}
```

Protocol buffer IDL

```
"seatsaver.Customer": {
    "title": "Customer",
    "type": "object",
    "properties": {
        "firstName": {
            "type": "string"
        },
        "lastName": {
            "type": "string"
        },
        "email": {
            "type": "string"
        },
        "created": {
            "type": "string"
        },
        "message": {
            "type": "string"
        }
    }
},
```

JSON Schema

The Value of Schema Definition

Client

```
{
  person: {
    firstName: string
    lastName: string
    dob: date
  },
  address: {
    address_1: string
    address_2: string
    city: string
    state: string
    zip: string
  }
}
```

Schema Registry

Server

```
{
  person: {
    givenName: string
    surName: string
    birthday: date
  },
  address: {
    address_1: string
    address_2: string
    city: string
    province: string
    postal_code: string
  }
}
```

What is a Schema Registry?

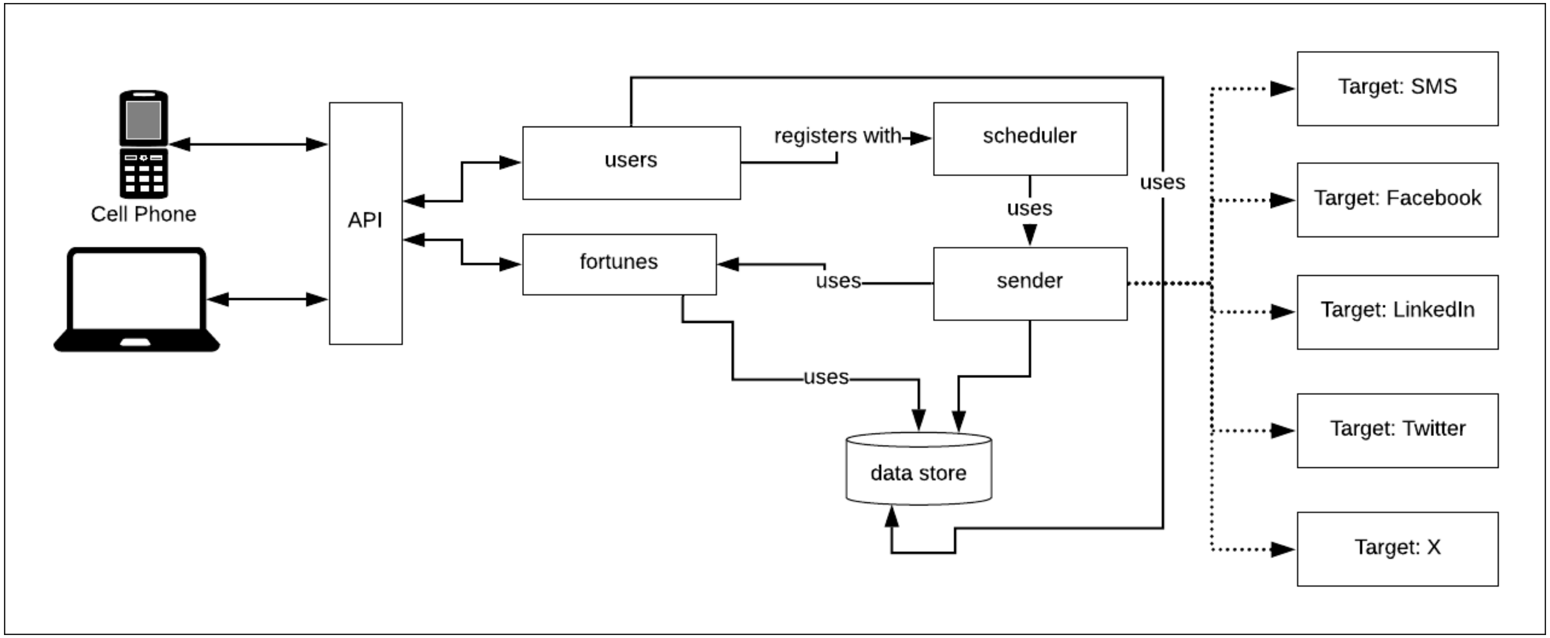
- Provides a centralized repository for schemas and metadata
  - ✓ Allowing services to flexibly interact and exchange data with each other without the challenge of managing and sharing schemas between them.
- Has support for multiple underlying schema representations (Avro, JSON, etc.)
  - ✓ Able to store a schema's corresponding serializer and deserializer.

Decomposition

Decomposition: 3 Questions to Ask

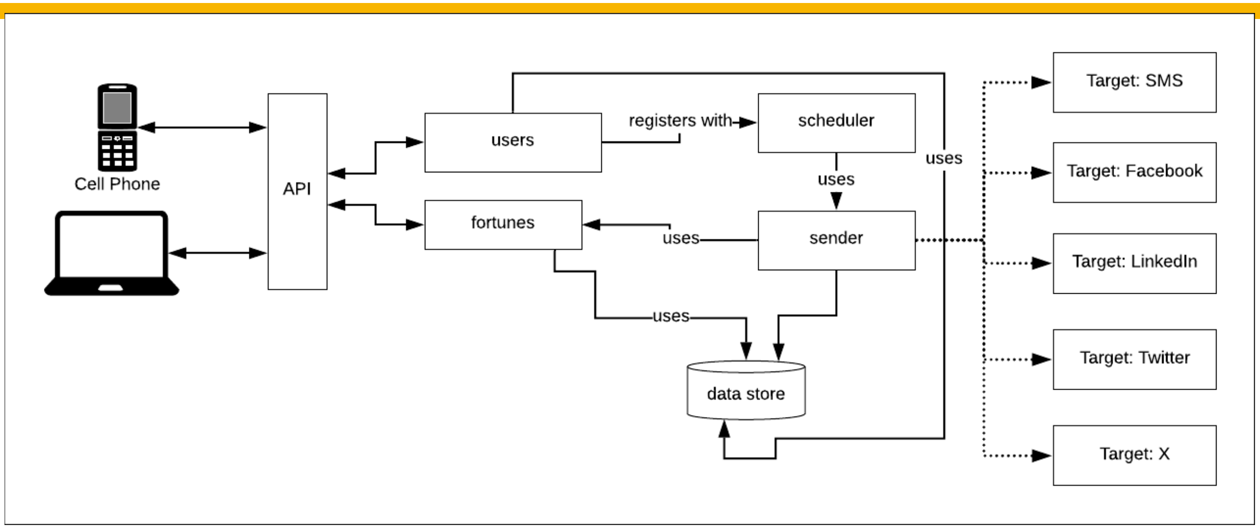
- Decomposition is not a practice of black and white
- Any design can be viable provided three basic questions are addressed
  - ✓ What is the justification for the design?
  - ✓ What are the risks?
  - ✓ What is the risk mitigation plan?

The Challenge of Decomposition

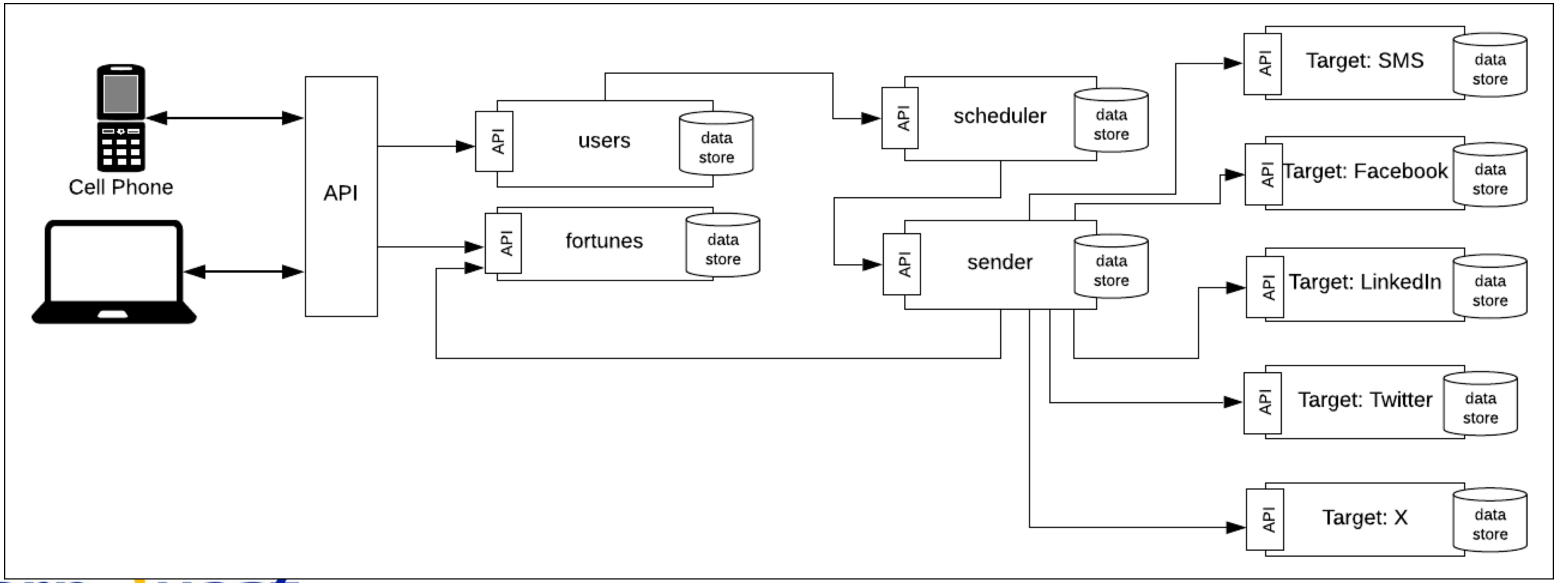


How do you transform this monolithic application into a set of microservices?

All at Once



One way is to create a whole new system and then do the transformation at once

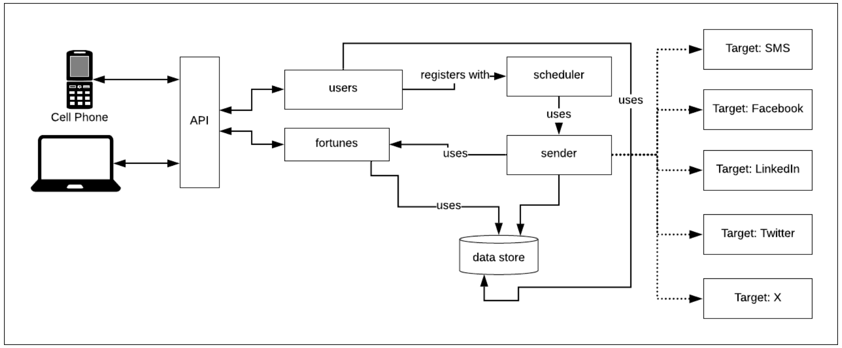


LearnQuest

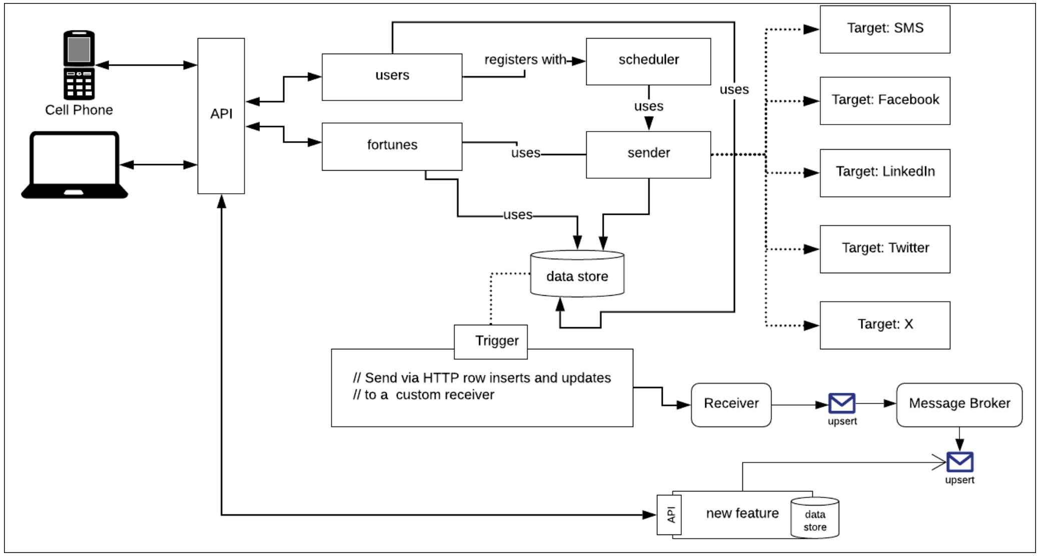
The Strangler Pattern

- New code is applied to new or existing features or services in an isolated manner
- Once the new code is applied, the existing service is disconnected
- Eventually all the services/features are replaced with well encapsulated substitutes
- The legacy application is strangled.

The Strangler Pattern



Applying the Strangler Pattern to a new feature



LearnQuest

Decomposition: All At Once

- Benefits
  - ✓ High degree of control of all the technology
  - ✓ Requires nothing more than '"flicking the switch"
  - ✓ Works well with green field development
- Risks
  - ✓ Expensive and can take a long time
    - The business finds it can't tolerate the costs and timeline
    - Longer projects go on the more risk is incurred
    - The longer projects go one, the potential for cancellation increases
  - ✓ Requires a large, organized workforce with each team focused on its service

LearnQuest

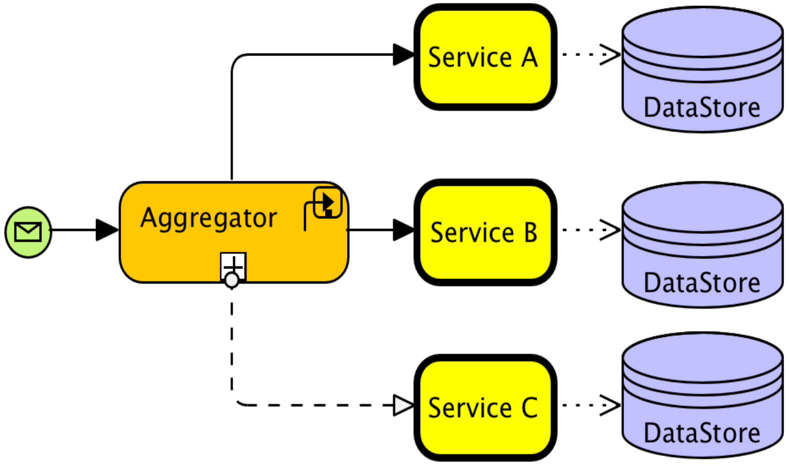
Decomposition: All At Once

- Benefits
  - ✓ Implementation can happen in phases
  - ✓ Existing system remain operational
  - ✓ Costs are confined to the given feature being strangled
  - ✓ Technical debt is restricted
  - ✓ Overall expertise with the Strangler Pattern develops over time
- Risks
  - ✓ Unanticipated technical hurdles
  - ✓ The move to production is a high-risk event
    - Rollback strategies must be bullet proof.
  - ✓ Requires a lot of technical accommodation while performing the feature substitution
  - ✓ Keeping data in sync can be hard

Patterns

Pattern: Aggregator

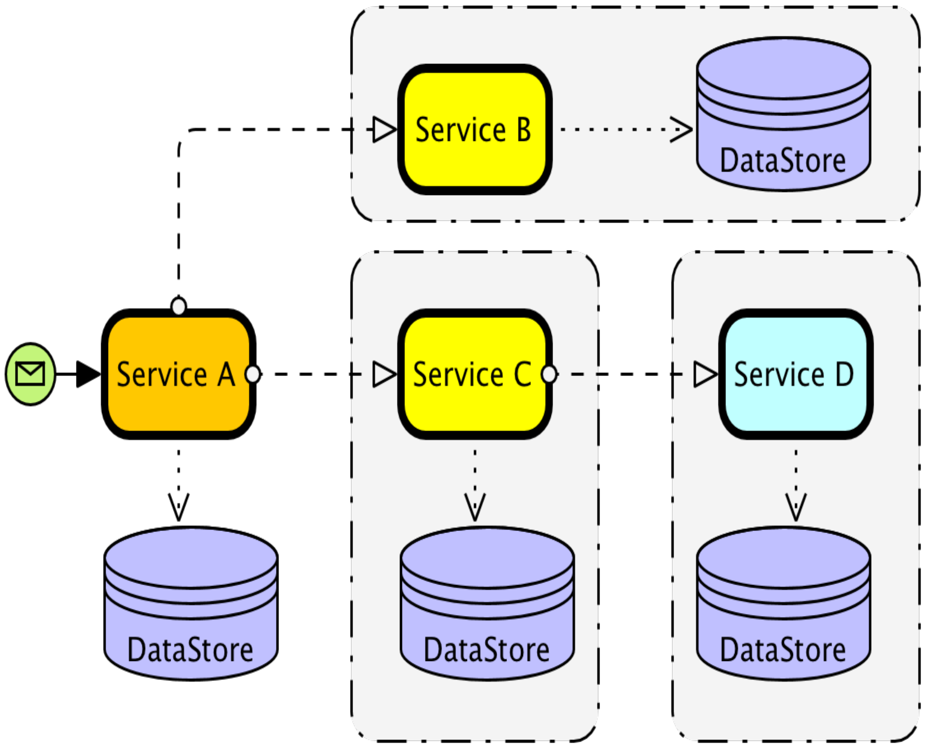
- An Aggregator is a service that invokes multiple services to achieve the required functionality
- An Aggregator could also be a composite service
- This design pattern follows the DRY principle
- An Aggregator is similar to the Façade Pattern



Pattern: Branch

- Description:
  - The Branch design pattern extends the Aggregator design pattern
  - Allows simultaneous response processing from two, likely mutually exclusive, chains of microservices
  - This pattern can also be used to call different chains, or a single chain, based upon the business needs
  - Service A, either a web page or a composite microservice, can invoke two different chains concurrently in which case this will resemble the Aggregator design pattern
  - This may be configured using routing of Enterprise Integration Pattern (EIP) endpoints, and would need to be dynamically configurable

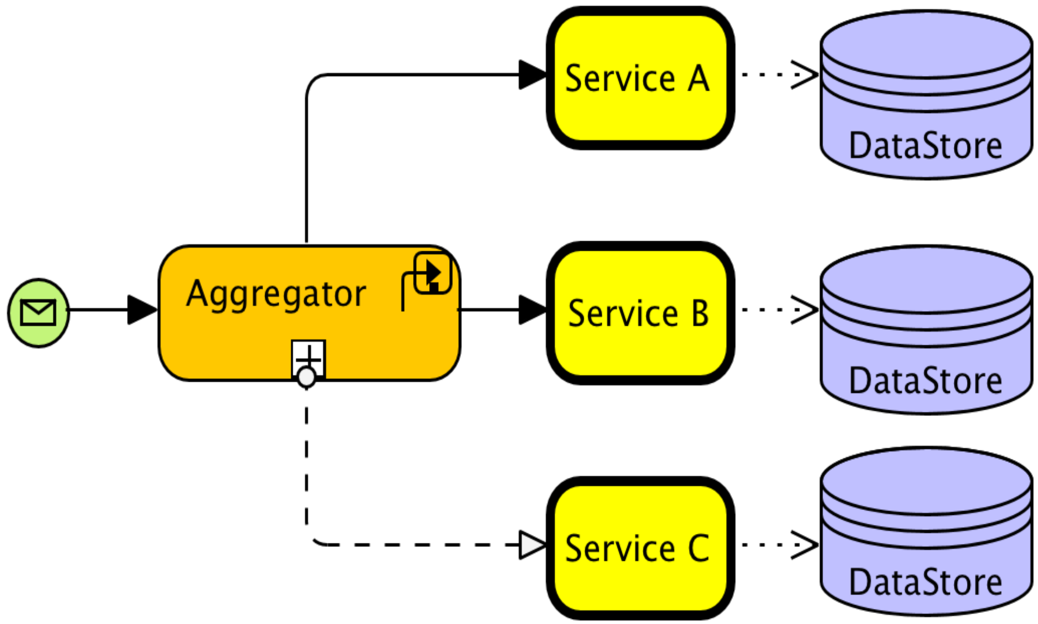
Pattern: Branch



Pattern: Proxy

- Description:
  - Proxy microservice design pattern is a variation of the Aggregator Pattern
  - Different microservice may be invoked based upon the business needs
  - Each Proxy can scale independently
  - A proxy is a consumer facing interface in lieu of exposing individual services
  - May be a dumb proxy that delegates requests to one of the services
  - May be a Smart Proxy, where data is transformed before response returned to consumer

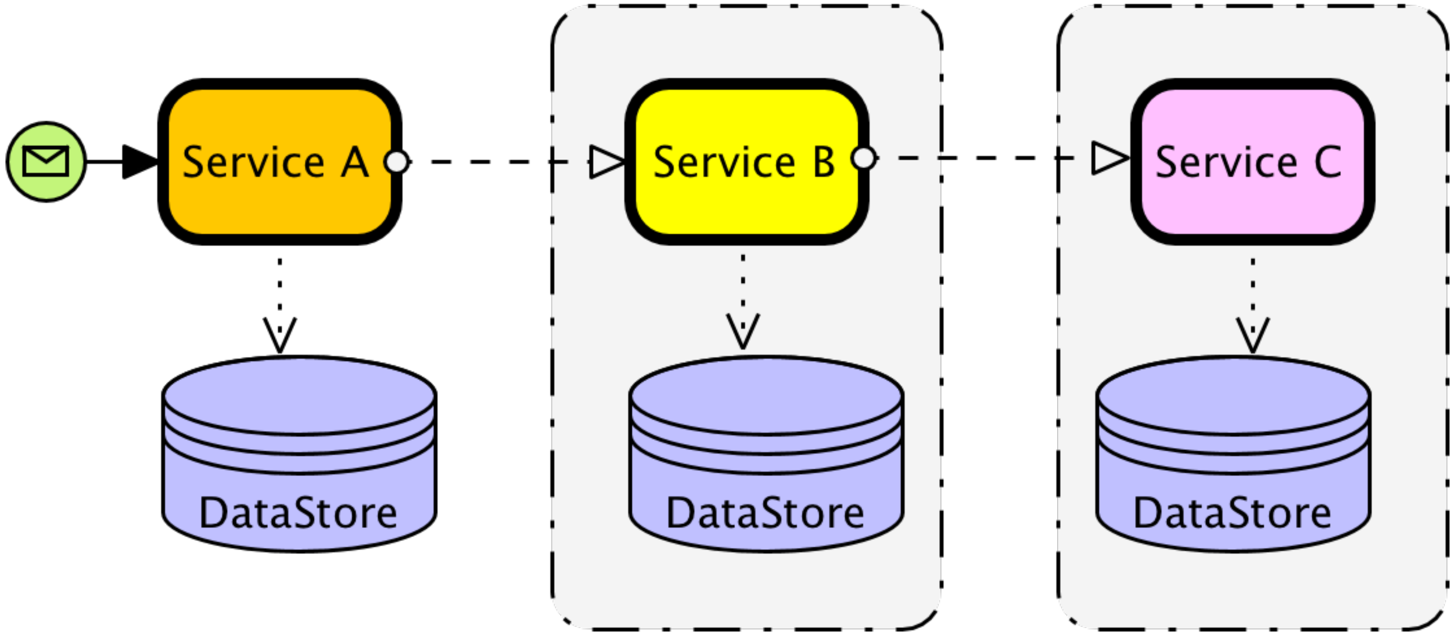
Pattern: Branch



Pattern: Chain

- Description:
  - Produces single consolidated response to request
  - Request is received by Service A, which is chained to Service B, which in turn which is changed to Service C
  - Services are likely using a synchronous HTTP request/response messaging
  - Client is blocked until request/response chain is completed
  - Important not to make chains too long
  - A chain with a single microservice is called singleton chain

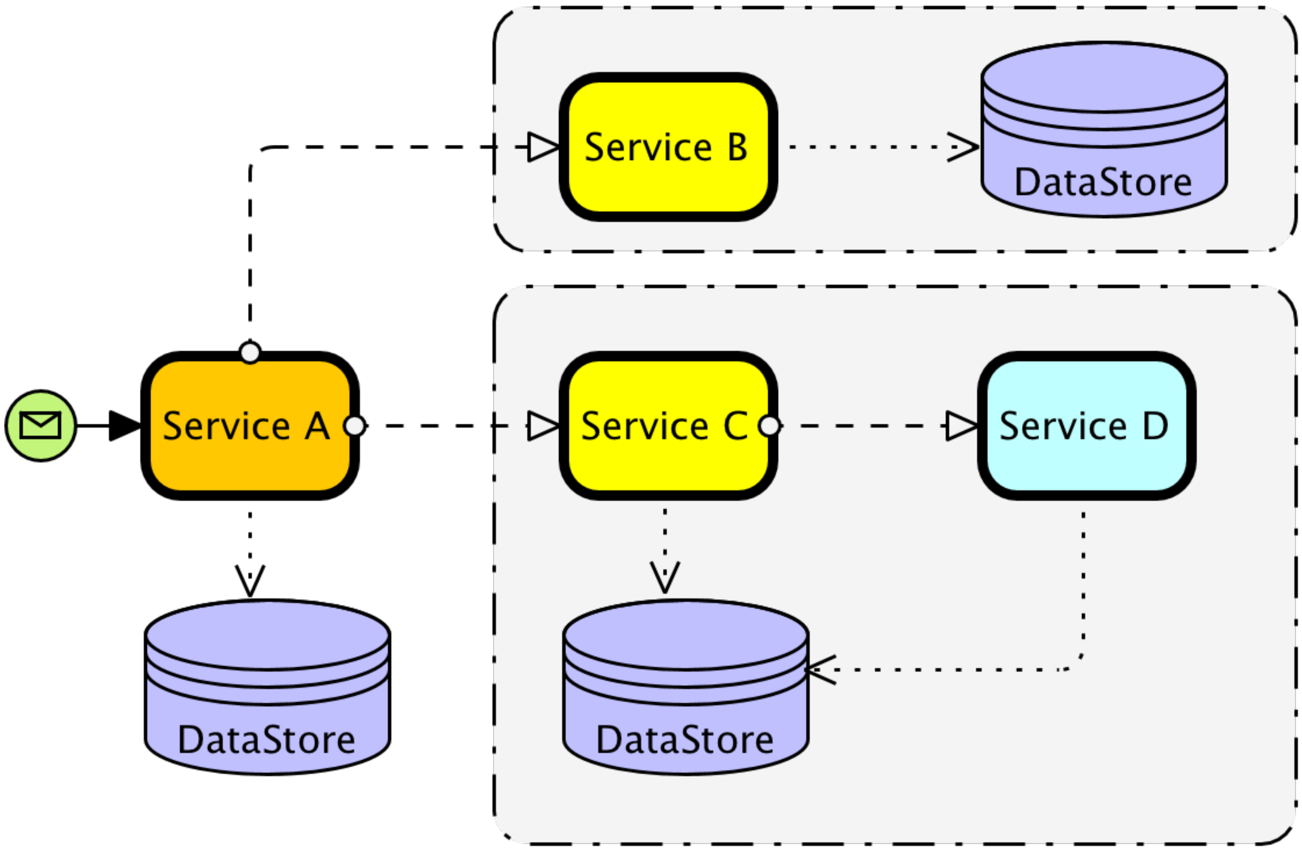
Pattern: Chain



Pattern: Shared Resources (Data)

- Description:
  - ✓ A fundamental design principle of microservice is autonomy
  - ✓ A service should be full-stack and have control of all the components
    - UI, middleware, persistence, transaction, etc.
  - ✓ Service should be polyglot
    - Use the right tool for the right job
  - ✓ Data normalization is typical issue when refactoring existing monolithic applications
  - ✓ This pattern may benefit from shared caching and database stores
  - ✓ This is a transitional pattern until service can be fully autonomous

Pattern: Shared Resources (Data)

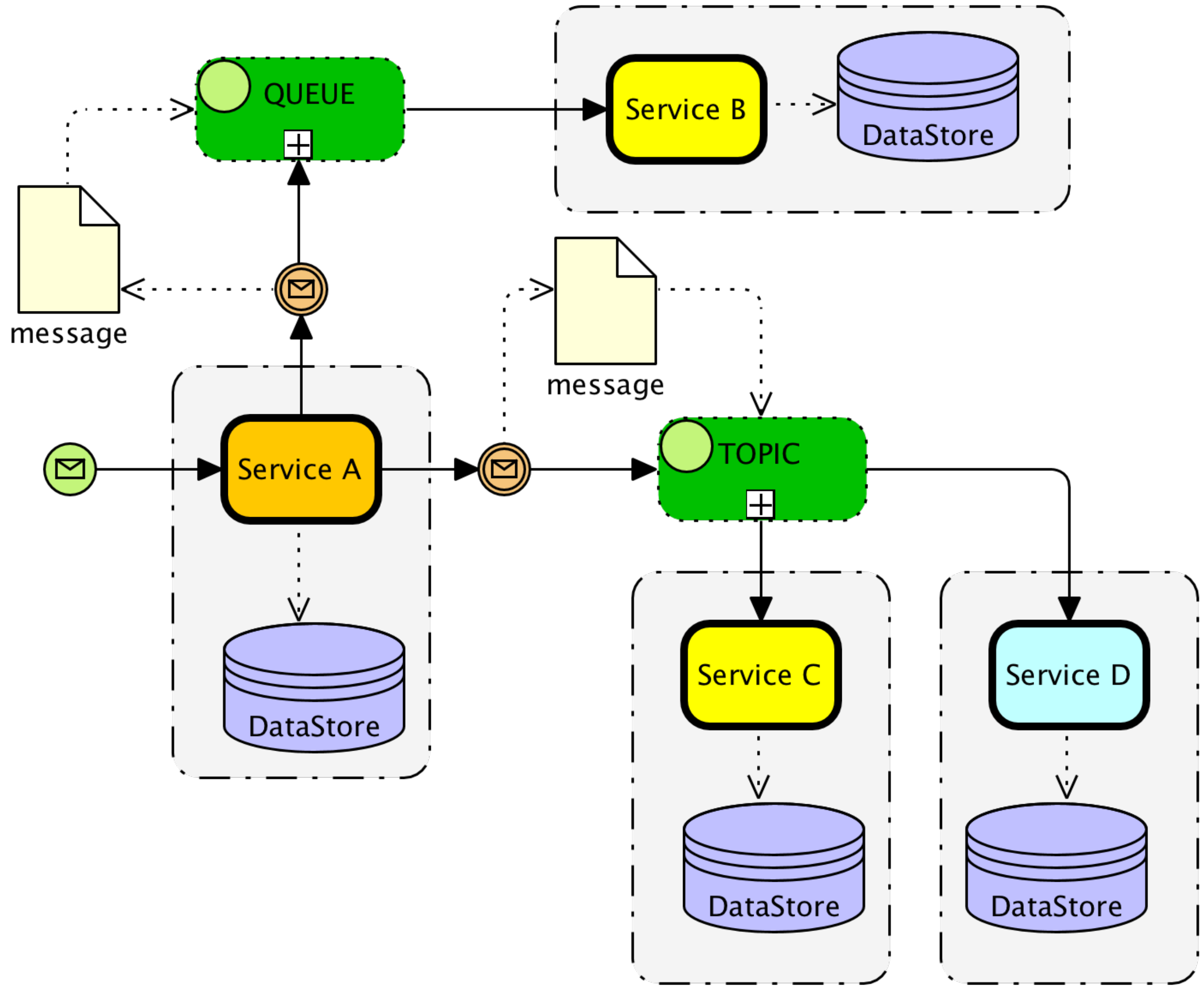


Pattern: Asynchronous Messaging

- Description:
  - ✓ RESTful services are inherently synchronous, and thus blocking
  - ✓ Microservice architectures may use asynchronous messaging instead of RESTful request/response
  - ✓ A service may call another service synchronously, which then communicates with other services asynchronously
  - ✓ Could communicate asynchronously in many different forms:
    - WebSockets, Message Queues, Reactive Streams
  - ✓ Can combine RESTful service and publish/subscribe messaging to accomplish the business needs

LearnQuest

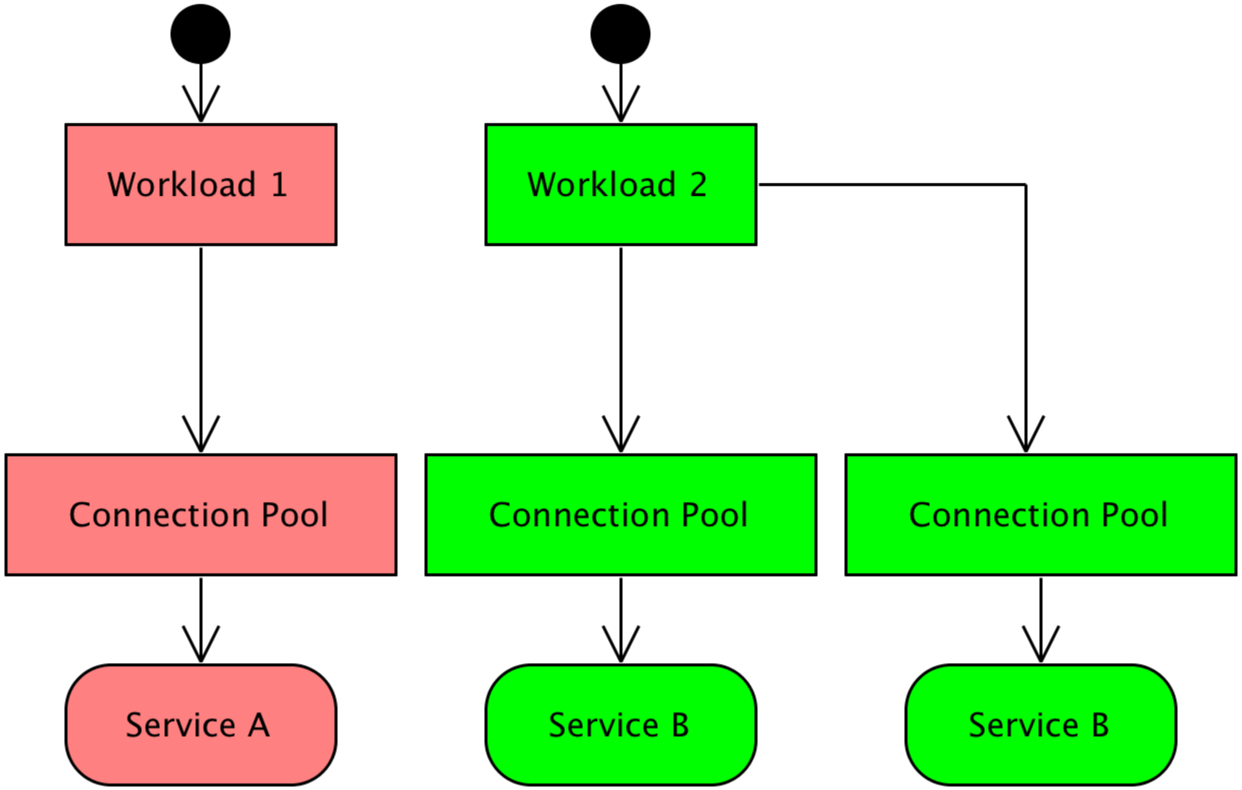
Pattern: Shared Resources (Data)



Pattern: Bulkhead Isolation

- Description:
  - Isolate elements of an application into pools so that if one fails, the others will continue to function
  - This pattern resembles the sectioned partitions of a ship's hull
    - If ship hull is compromised, only the damaged section fills with water, which prevents the ship from sinking
  - Partition service instances into different groups, based on consumer load and availability requirements
  - Isolating failures, and allowing application to sustain service functionality for some consumers, even during a failure

Pattern: Bulkhead Isolation



Pattern: Bulkhead Isolation

- Issues and considerations
  - ✓ Define partitions around the business and technical requirements of the application
  - ✓ Consider level of isolation offered by the technology as well as the overhead in terms of cost, performance and manageability
  - ✓ Consider combining bulkheads with retry, circuit breaker, and throttling patterns to provide more sophisticated fault handling
  - ✓ Use processes, thread pools, and semaphores
  - ✓ Use frameworks for creating consumer bulkheads
  - ✓ Deploy into separate VM’s, containers, or processes
  - ✓ Isolate asynchronous service into different queues
  - ✓ Monitor each partition’s performance and SLA

Questions