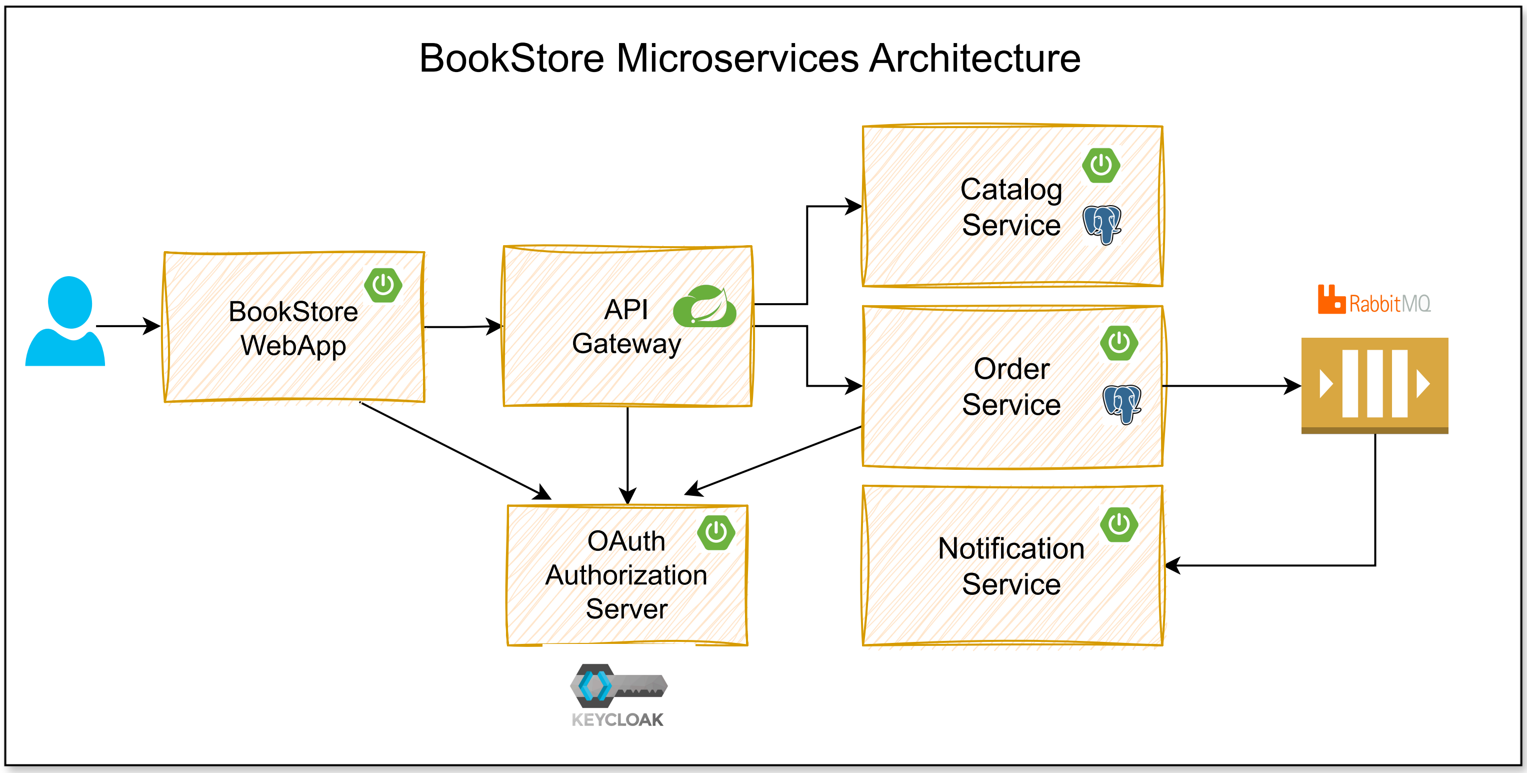


What is Microservices Architecture?

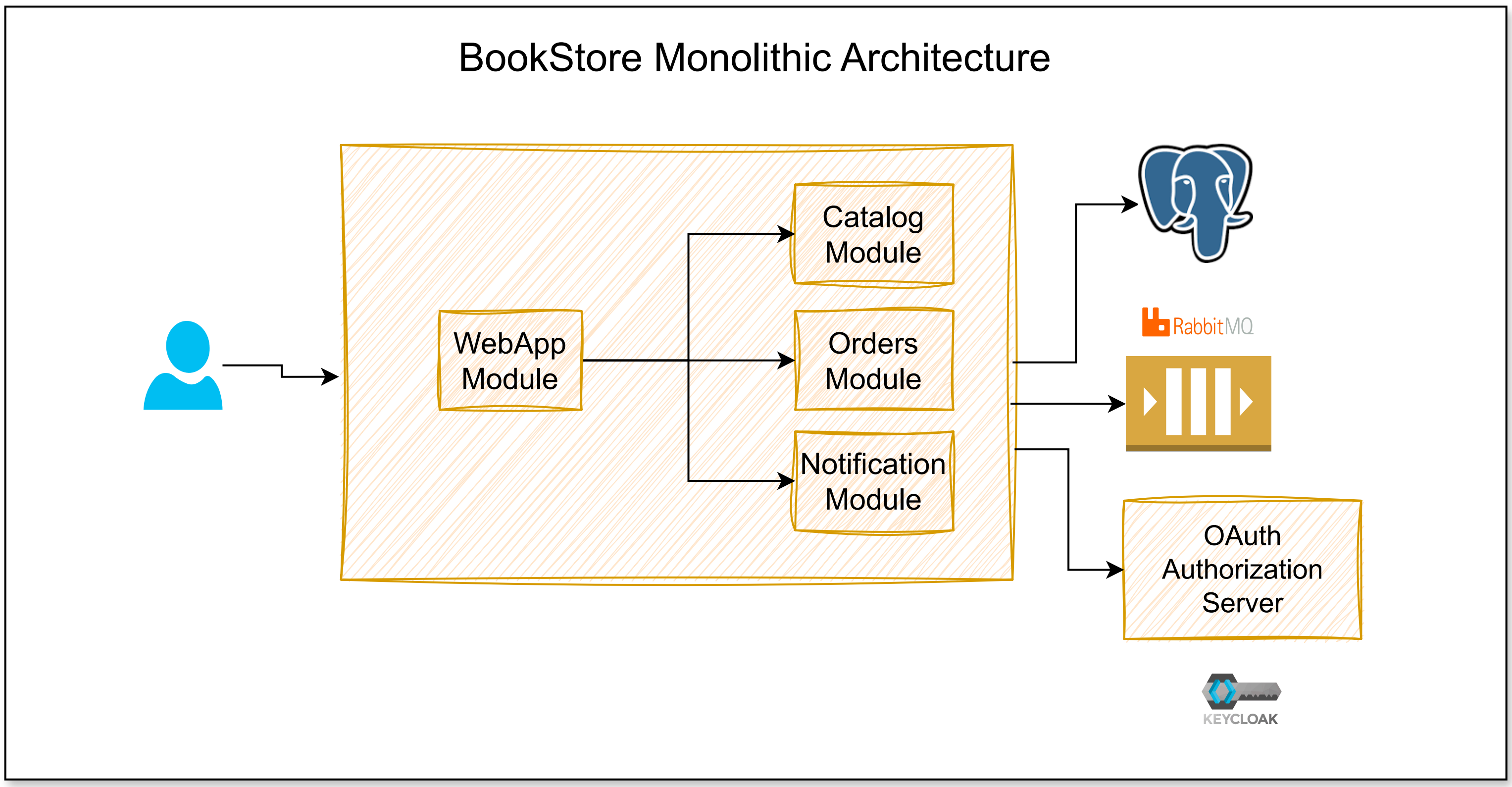
Microservices is an architectural style where large applications are composed of small services that provide specific business capabilities that can be deployed and managed independently.

Learning Objectives

- Building Spring Boot REST APIs
- Database Persistence using Spring Data JPA, Postgres, Flyway
- Event Driven Async Communication using RabbitMQ
- Implementing OAuth2-based Security using Spring Security and Keycloak
- Implementing API Gateway using Spring Cloud Gateway
- Implementing Resiliency using Resilience4j
- Job Scheduling with ShedLock-based distributed Locking
- Using RestClient, Declarative HTTP Interfaces to invoke other APIs
- Creating Aggregated Swagger Documentation at API Gateway
- Local Development Setup using Docker and Testcontainers
- Testing using JUnit 5, RestAssured, Testcontainers, Awaitility, WireMock

Additional Topics(Membership)

- Monitoring & Observability using Grafana, Prometheus, Loki, Tempo
- Kubernetes 101 course
- Deployment to Kubernetes



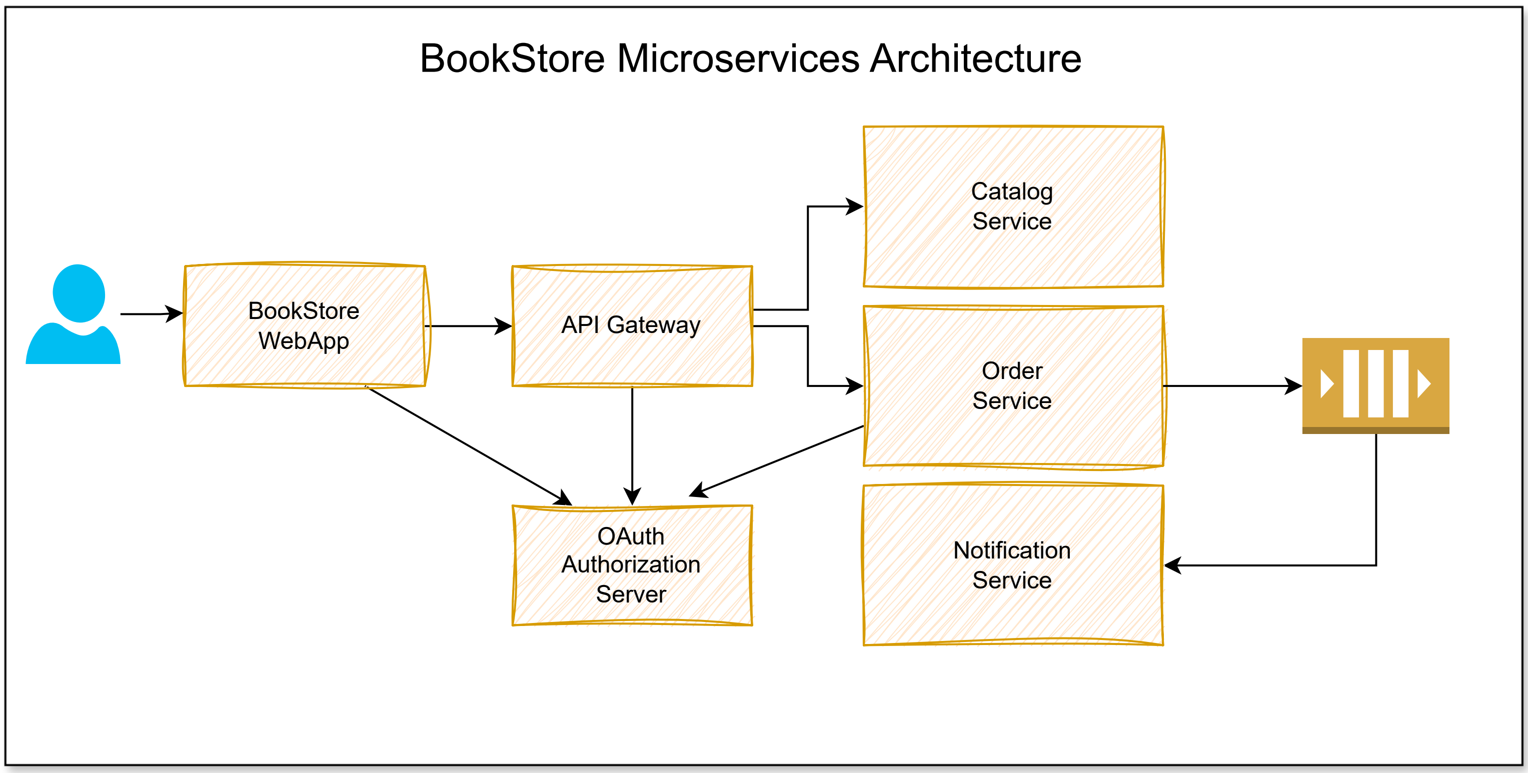
Monolithic Architecture

Pros

1. Simpler Development
2. Easier Testing & Debugging
3. Simpler Deployment

Cons

1. Difficult to scale sub-systems(modules)
2. Difficult to adopt new technologies
3. Higher chance to become big ball of mud



Microservices Architecture

Pros

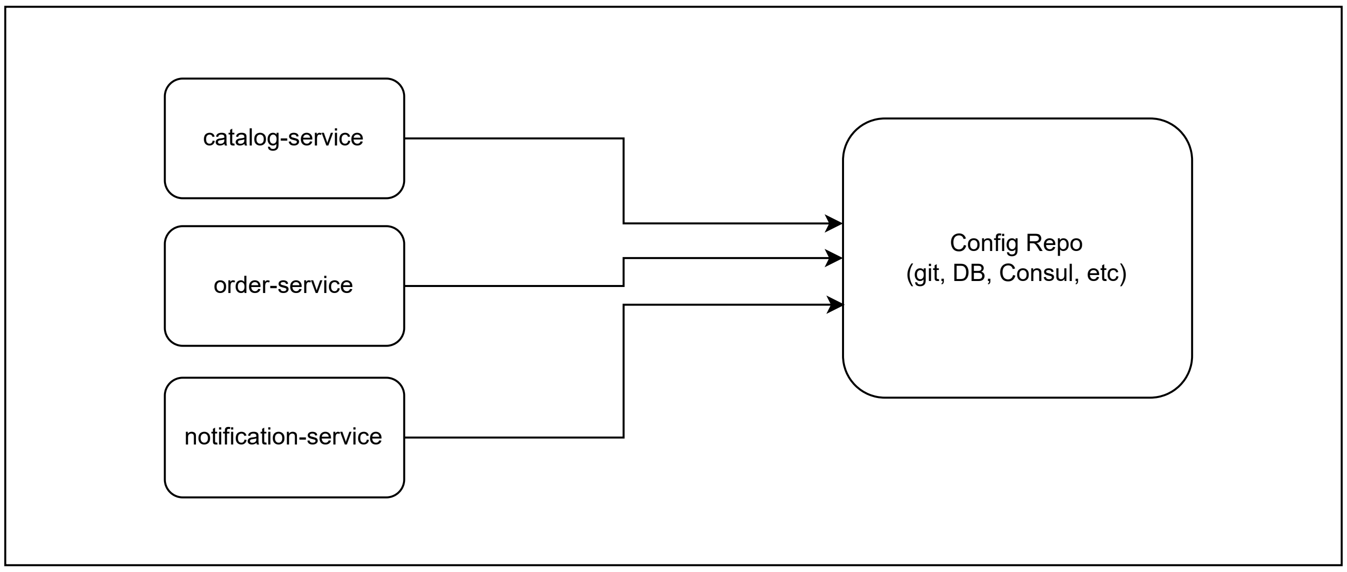
1. Can scale individual services
2. Smaller codebases easy to reason about
3. Easy to adopt newer technologies if needed
4. Less dependency on other team deliverables

Cons

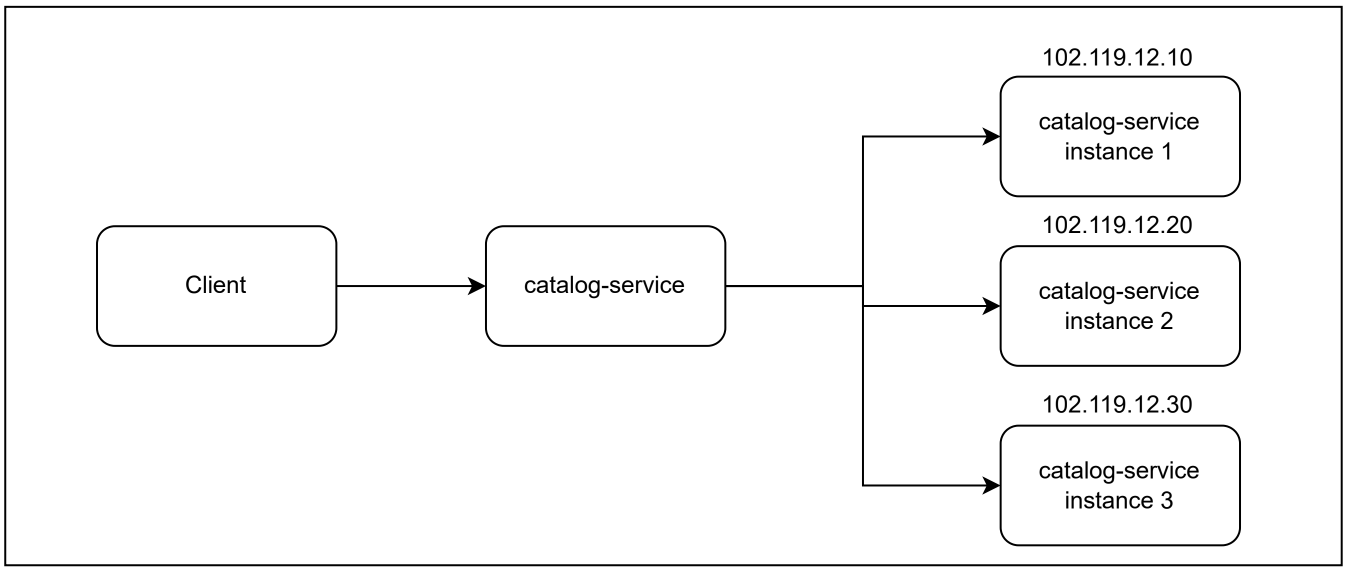
1. Difficult to build & manage distributed systems
2. Difficult to test & debug
3. Complex deployment process
4. Performance Issues

Spring Cloud vs Kubernetes

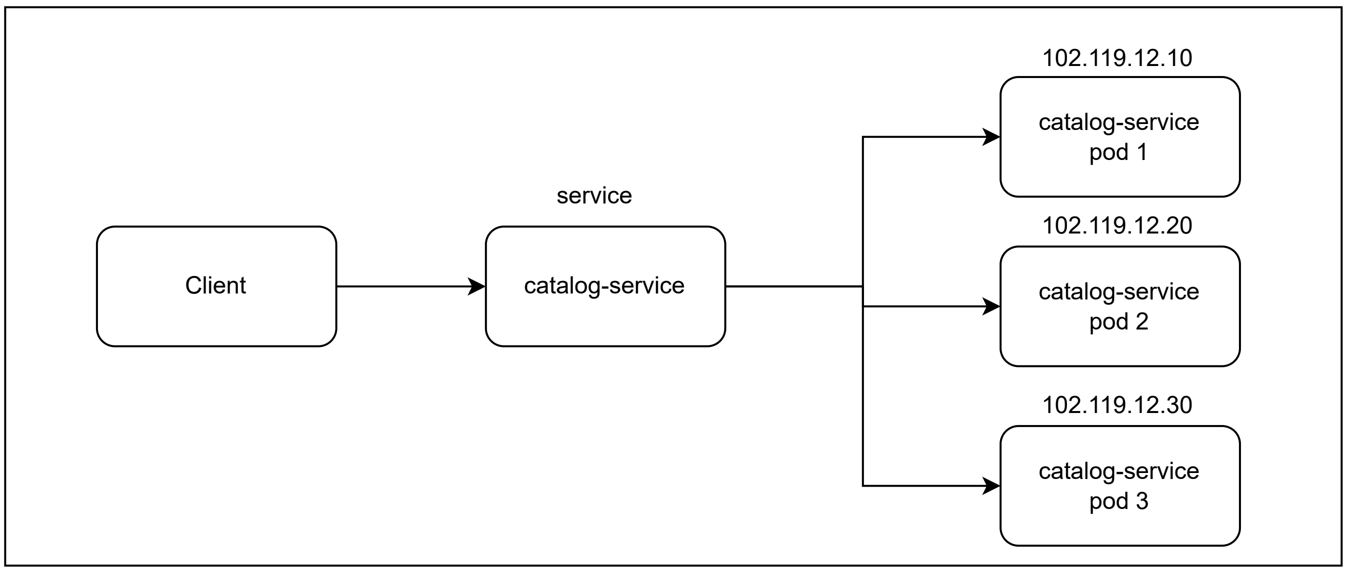
Spring Cloud Config Server



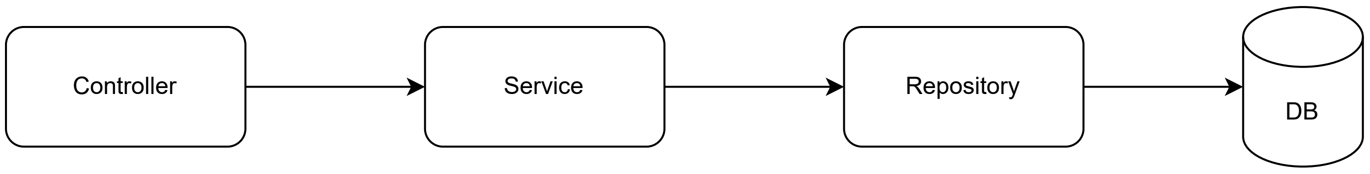
Service Registry (Eureka, Consul)


Client
catalog-service
102.119.12.10
catalog-service instance 1
102.119.12.20
catalog-service instance 2
102.119.12.30
catalog-service instance 3

Kubernetes



Package Structure



Package By Layer

root
- controllers
    - ProductController
    - CustomerController
- services
    - ProductService
    - CustomerService

- repositories
    - ProductRepository
    - CustomerRepository

Package By Feature

root
- products
    - ProductController
    - ProductService
    - ProductRepository

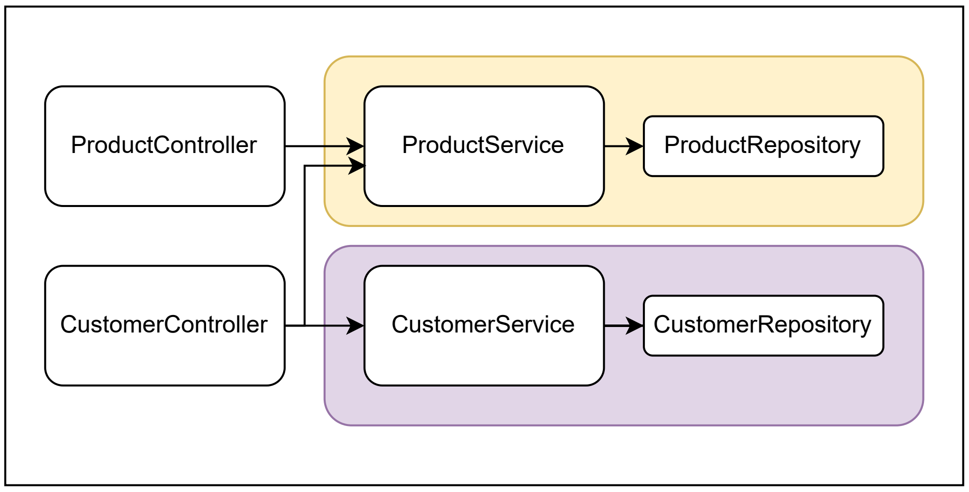
- customers
    - CustomerController
    - CustomerService
    - CustomerRepository

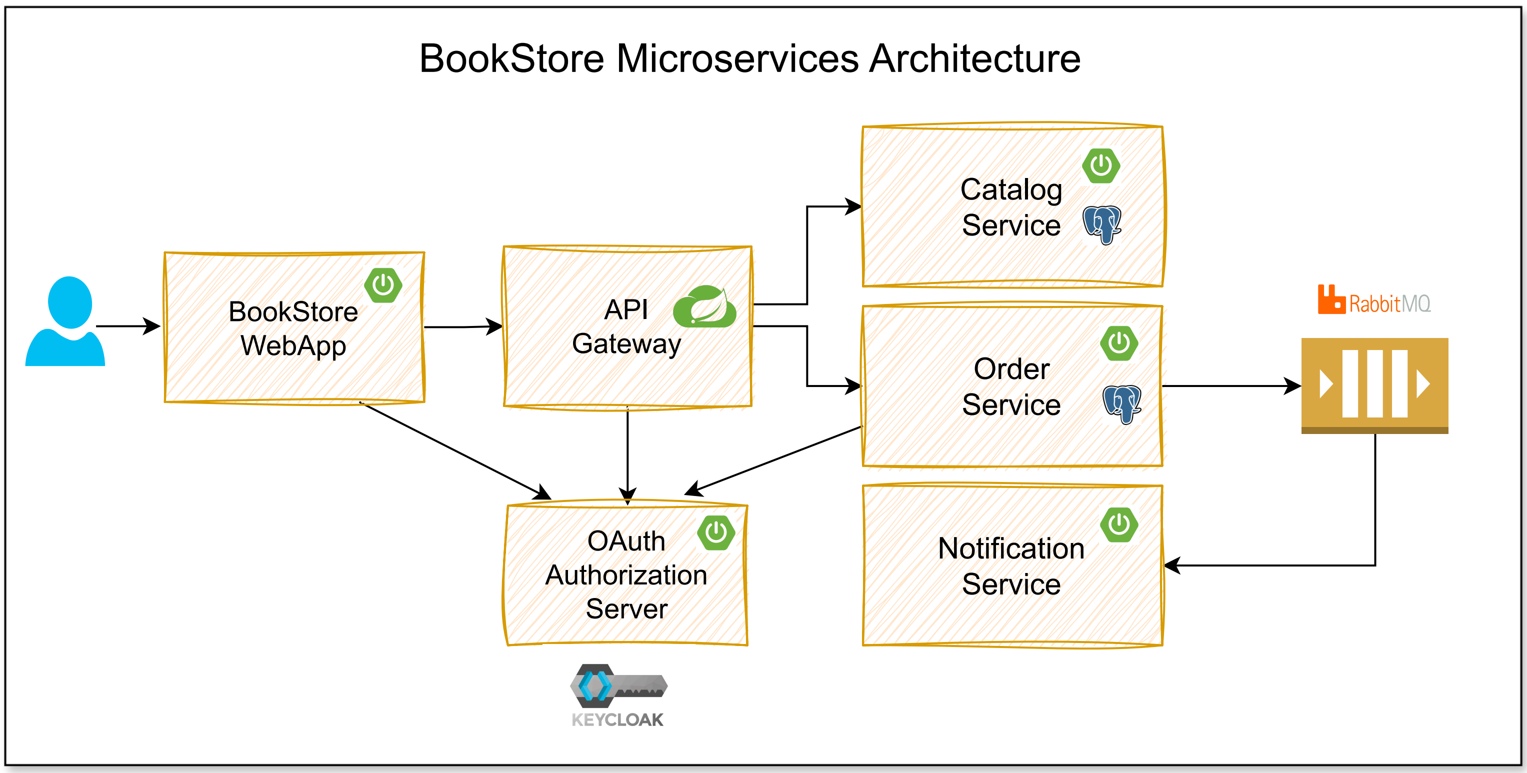
Package By Component

root
- web
    - products
        - ProductController
    - customers
        - CustomerController

- domain
    - products
        - ProductService
        - ProductRepository

    - customers
        - CustomerService
        - CustomerRepository





What is Microservices Architecture?

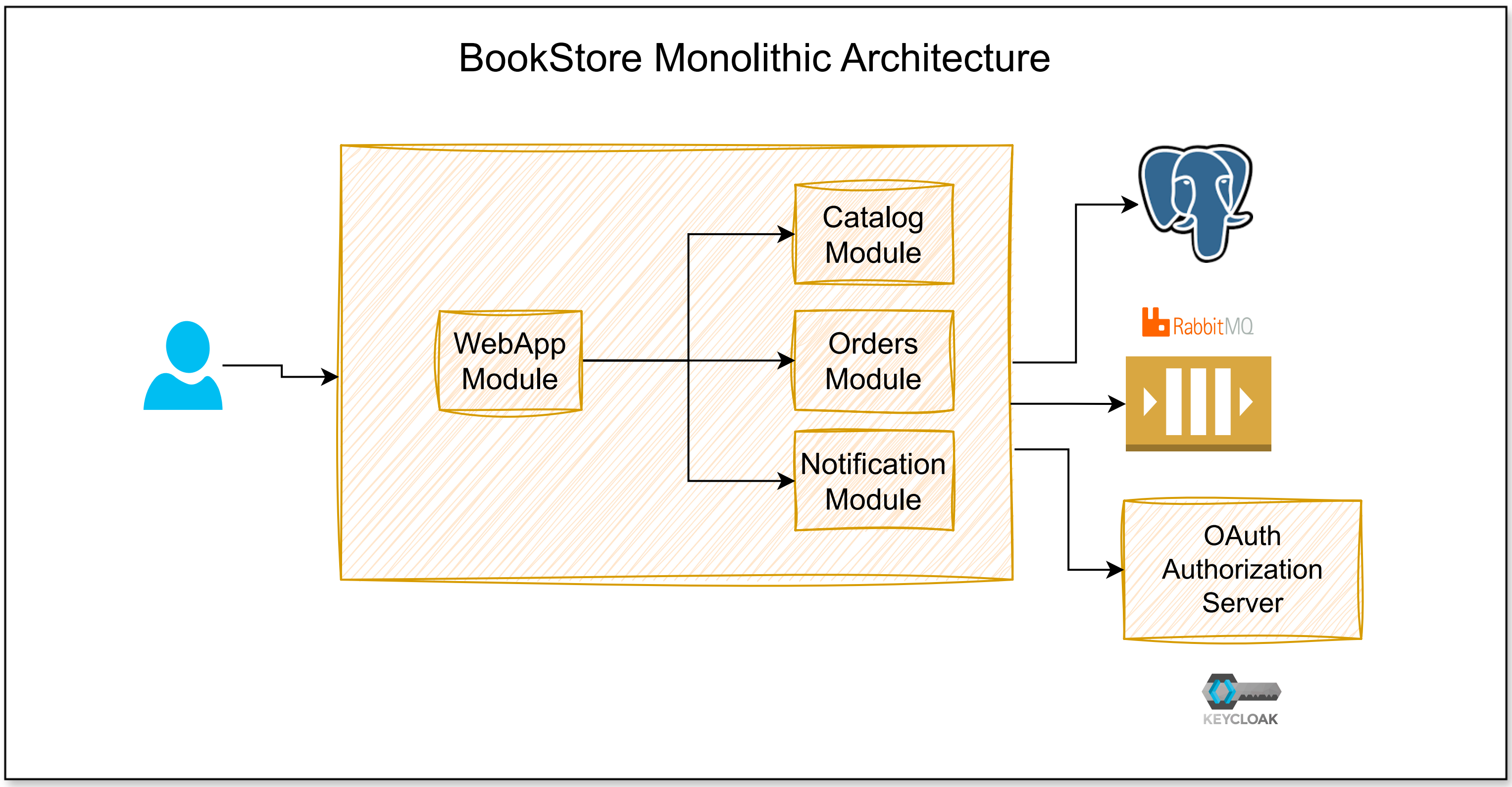
Microservices is an architectural style where large applications are composed of small services that provide specific business capabilities that can be deployed and managed independently.

Learning Objectives

- Building Spring Boot REST APIs
- Database Persistence using Spring Data JPA, Postgres, Flyway
- Event Driven Async Communication using RabbitMQ
- Implementing OAuth2-based Security using Spring Security and Keycloak
- Implementing API Gateway using Spring Cloud Gateway
- Implementing Resiliency using Resilience4j
- Job Scheduling with ShedLock-based distributed Locking
- Using RestClient, Declarative HTTP Interfaces to invoke other APIs
- Creating Aggregated Swagger Documentation at API Gateway
- Local Development Setup using Docker and Testcontainers
- Testing using JUnit 5, RestAssured, Testcontainers, Awaitility, WireMock

Additional Topics(Membership)

- Monitoring & Observability using Grafana, Prometheus, Loki, Tempo
- Kubernetes 101 course
- Deployment to Kubernetes



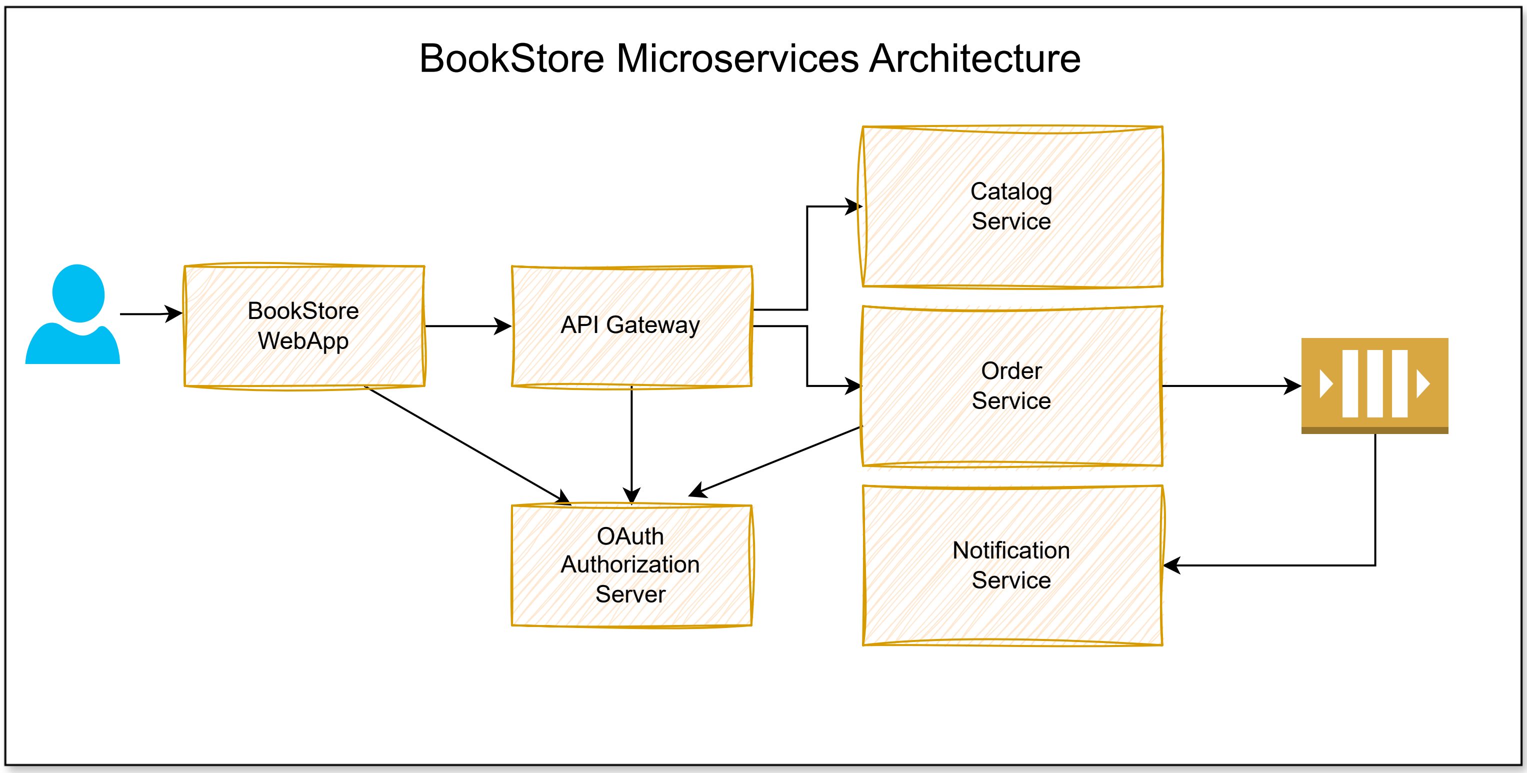
Monolithic Architecture

Pros

1. Simpler Development
2. Easier Testing & Debugging
3. Simpler Deployment

Cons

1. Difficult to scale sub-systems(modules)
2. Difficult to adopt new technologies
3. Higher chance to become big ball of mud



Microservices Architecture

Pros

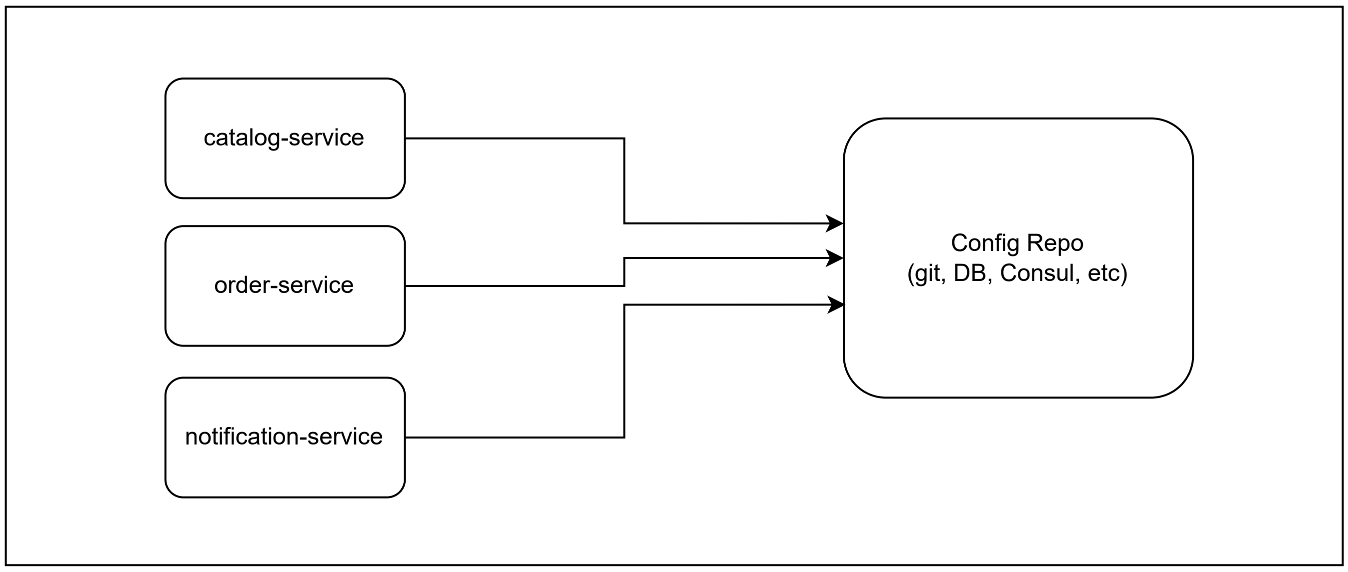
1. Can scale individual services
2. Smaller codebases easy to reason about
3. Easy to adopt newer technologies if needed
4. Less dependency on other team deliverables

Cons

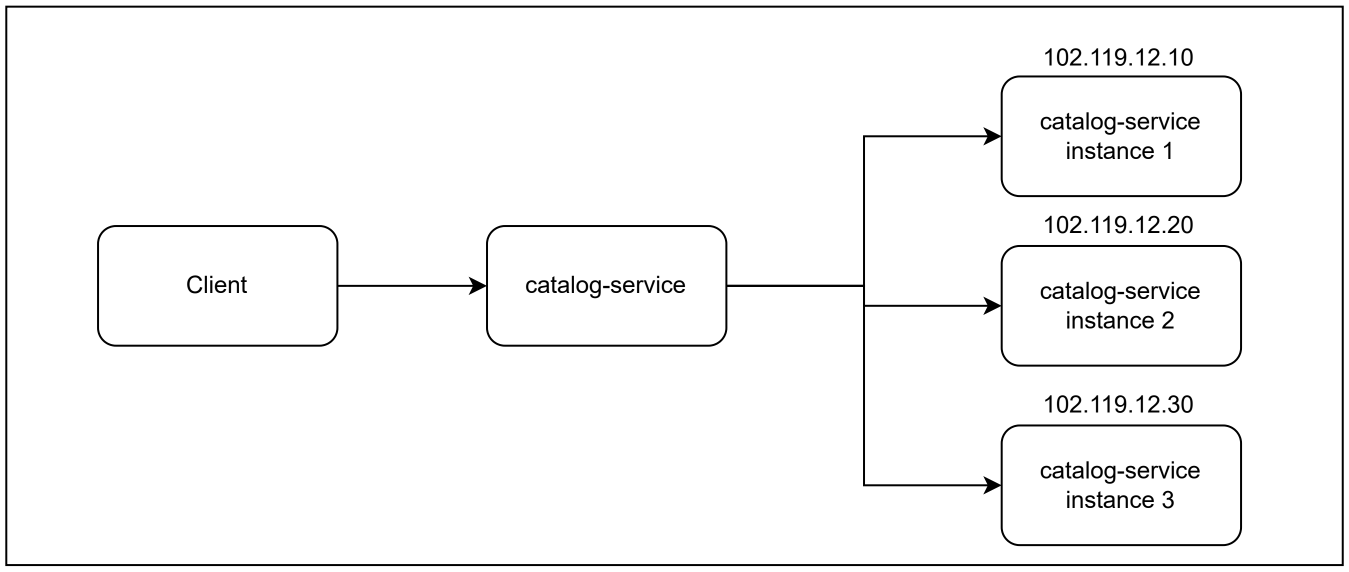
1. Difficult to build & manage distributed systems
2. Difficult to test & debug
3. Complex deployment process
4. Performance Issues

Spring Cloud vs Kubernetes

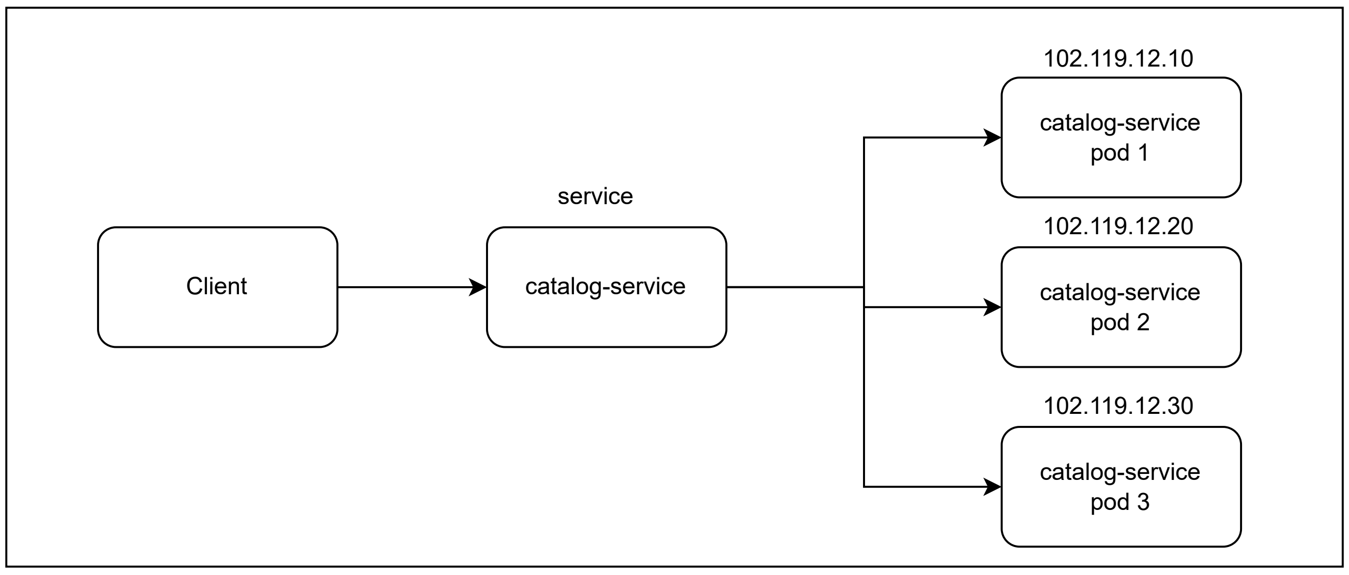
Spring Cloud Config Server



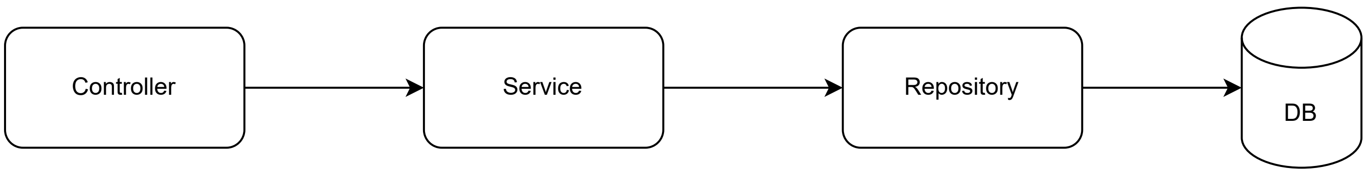
Service Registry (Eureka, Consul)



Kubernetes



Package Structure



Package By Layer

root
  - controllers
    - ProductController
    - CustomerController
  - services
    - ProductService
    - CustomerService

  - repositories
    - ProductRepository
    - CustomerRepository

Package By Feature

root
  - products
    - ProductController
    - ProductService
    - ProductRepository

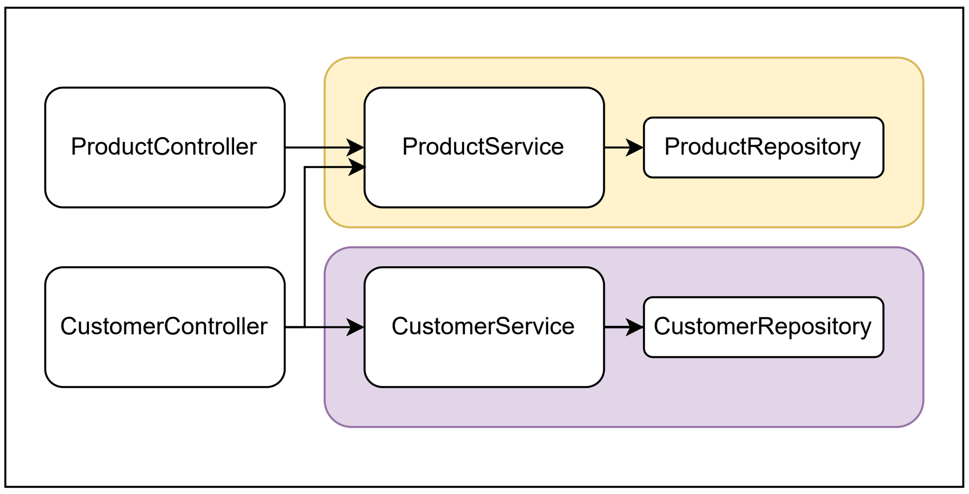
  - customers
    - CustomerController
    - CustomerService
    - CustomerRepository

Package By Component

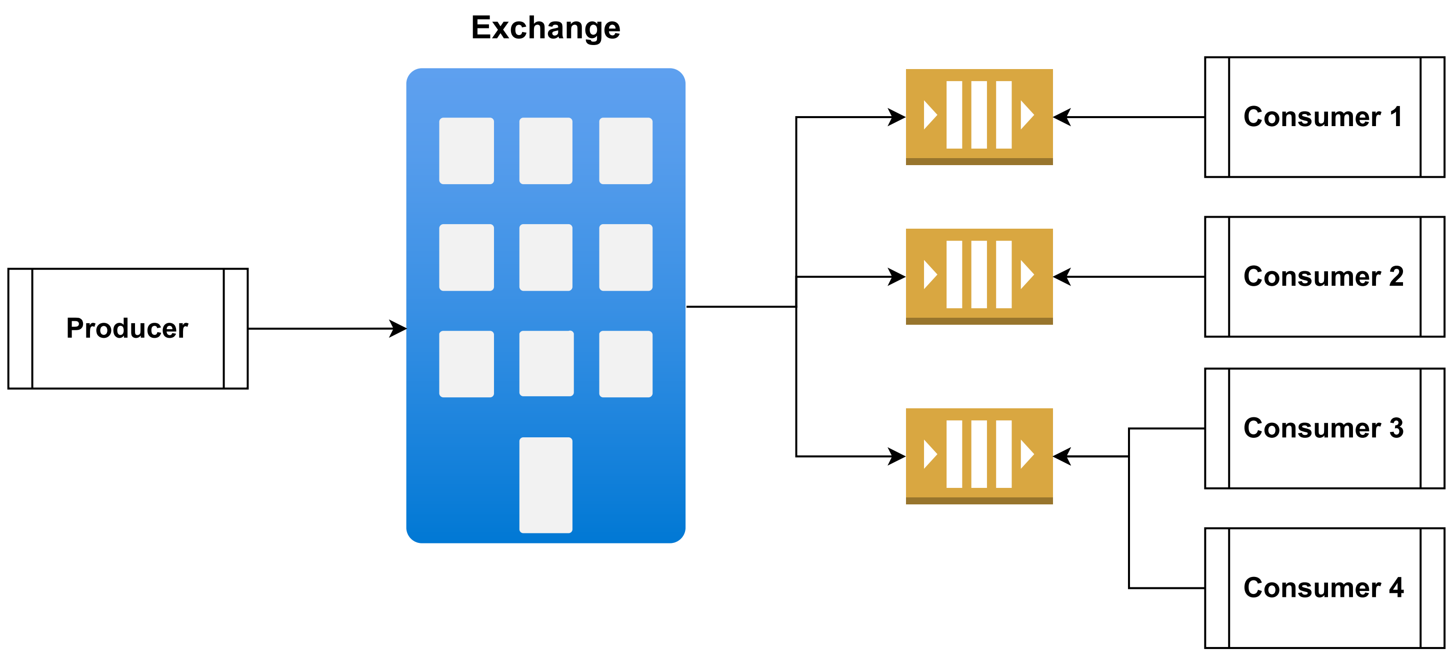
root
  - web
    - products
      - ProductController
    - customers
      - CustomerController

  - domain
    - products
      - ProductService
      - ProductRepository

    - customers
      - CustomerService
      - CustomerRepository



RabbitMQ



Exchange Types

1. Direct Exchange
2. Topic Exchange
3. Fanout Exchange

Direct Exchange

Binding Key: Simple String

Ex: orders, cancellations, accounts, new-orders, delivered-orders, etc

Topic Exchange

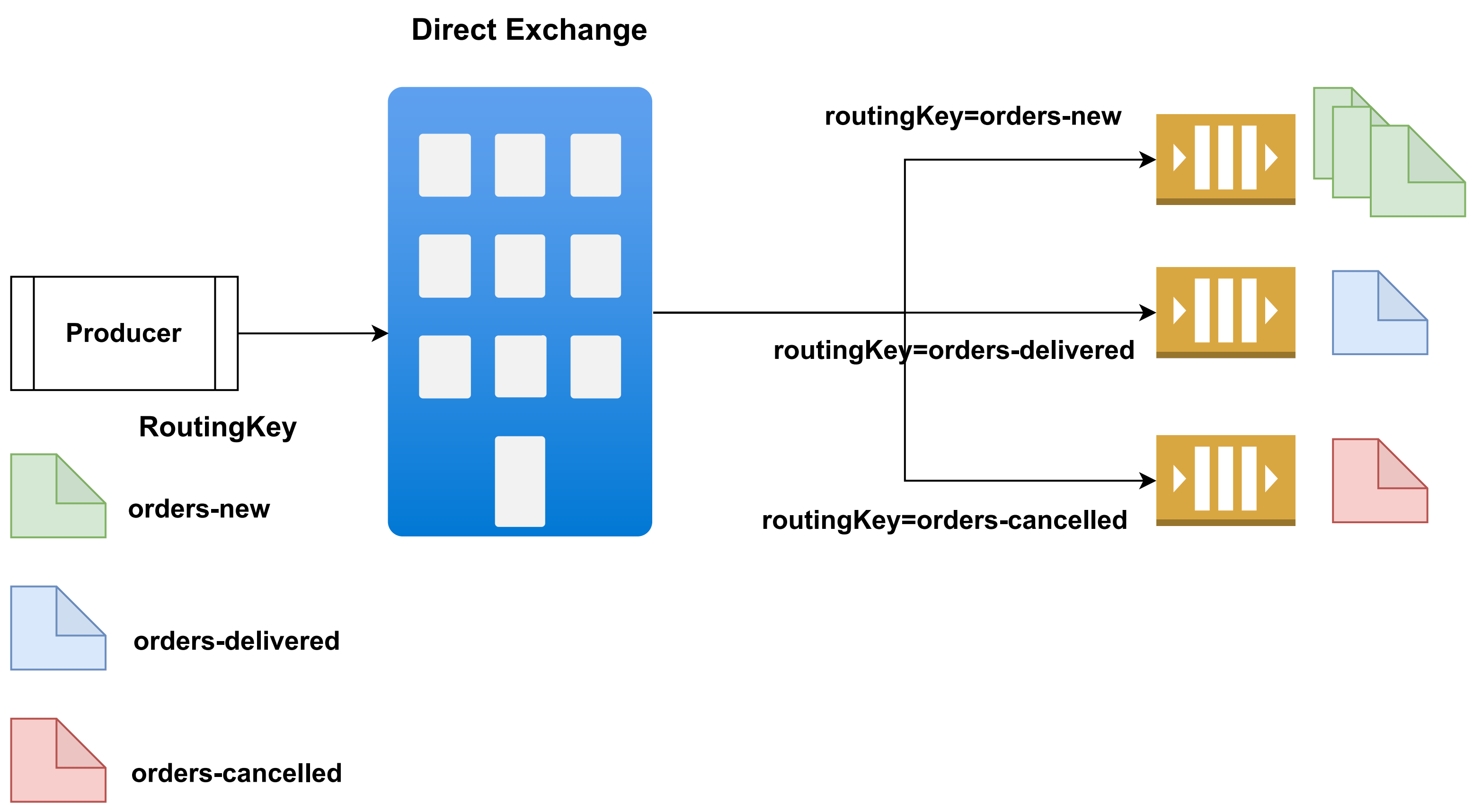
Binding Key: Support Patterns and Wildcards

Ex: orders.new.*, orders.*.cancelled, orders.new.#, etc

Fanout Exchange

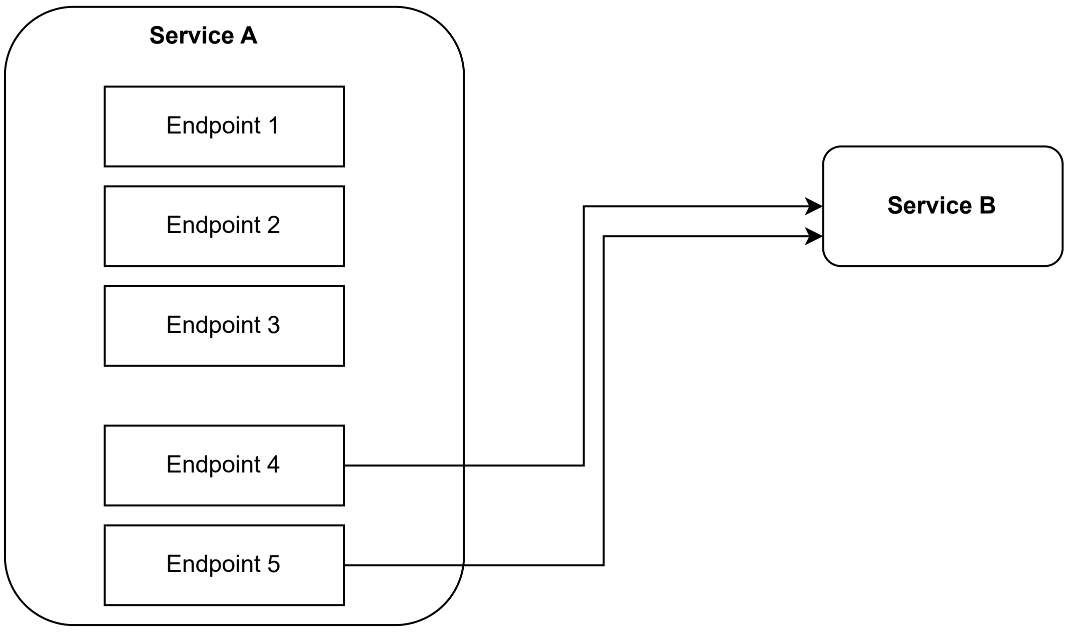
Binding Key: Ignored

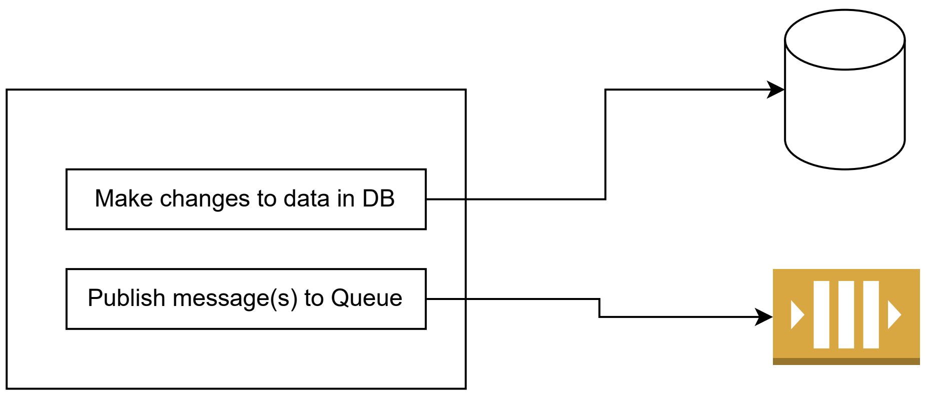
Broadcast the messages to all the queues


Direct Exchange
Producer
RoutingKey
orders-new
orders-delivered
orders-cancelled
routingKey=orders-new
routingKey=orders-delivered
routingKey=orders-cancelled

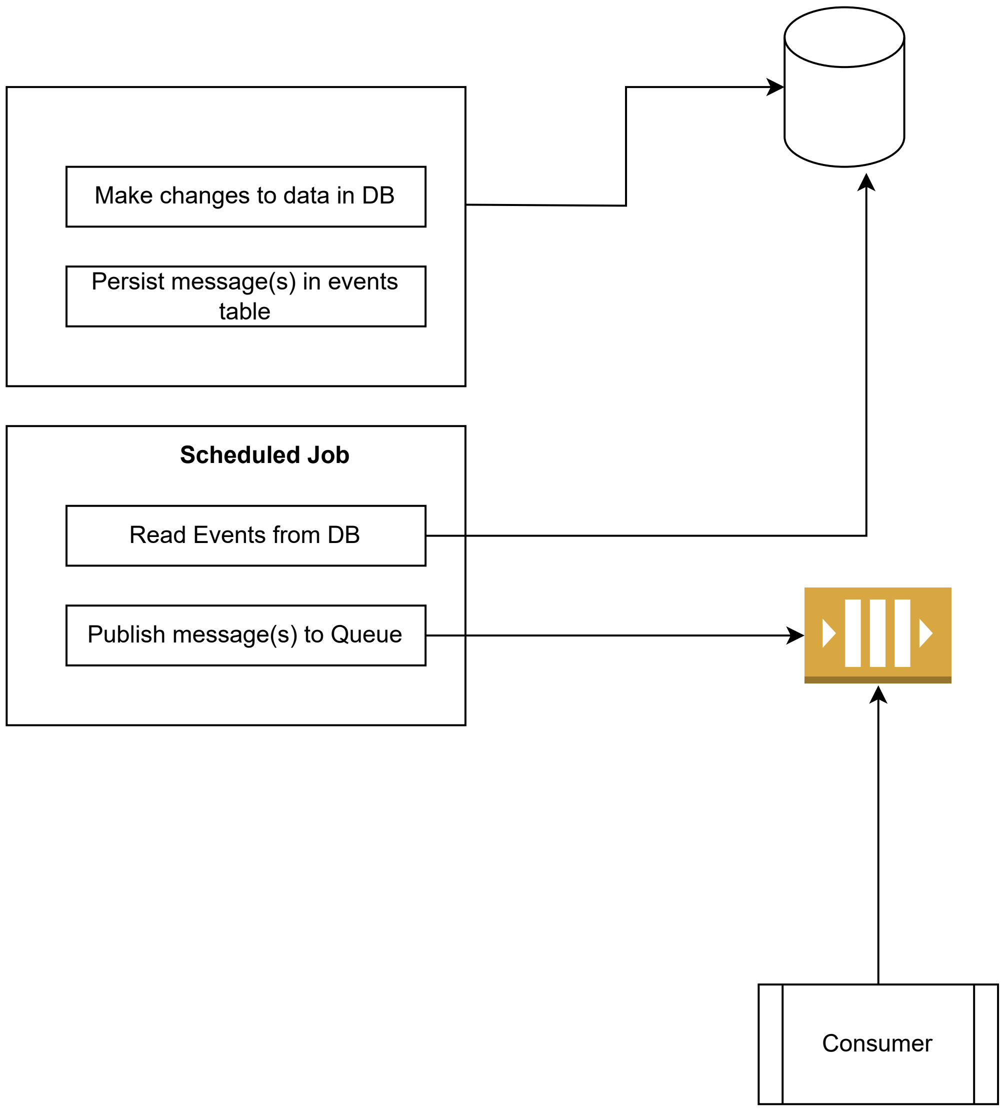
Resilience Patterns

1. Timeout (or) TimeLimiter
2. Retry
3. Bulk Head
4. Circuit Breaker
5. Rate Limiter



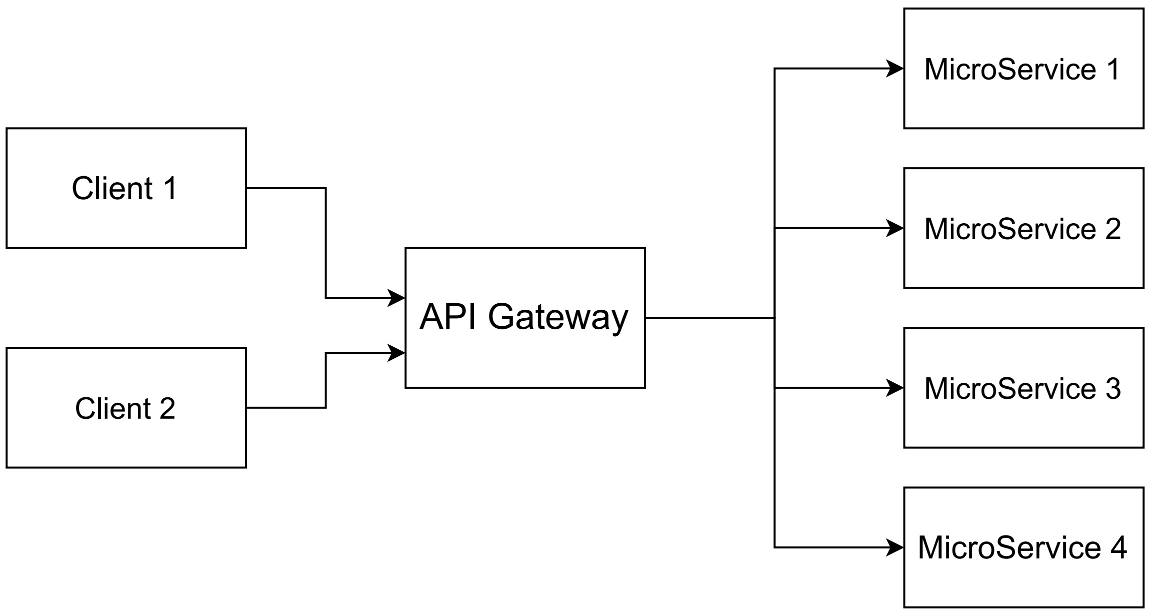


OutBox Pattern


Make changes to data in DB
Persist message(s) in events table
Scheduled Job
Read Events from DB
Publish message(s) to Queue
Consumer

NOTE: Consumer should be idempotent. i.e, should be able to handle duplicate messages

API Gateway



API Gateway

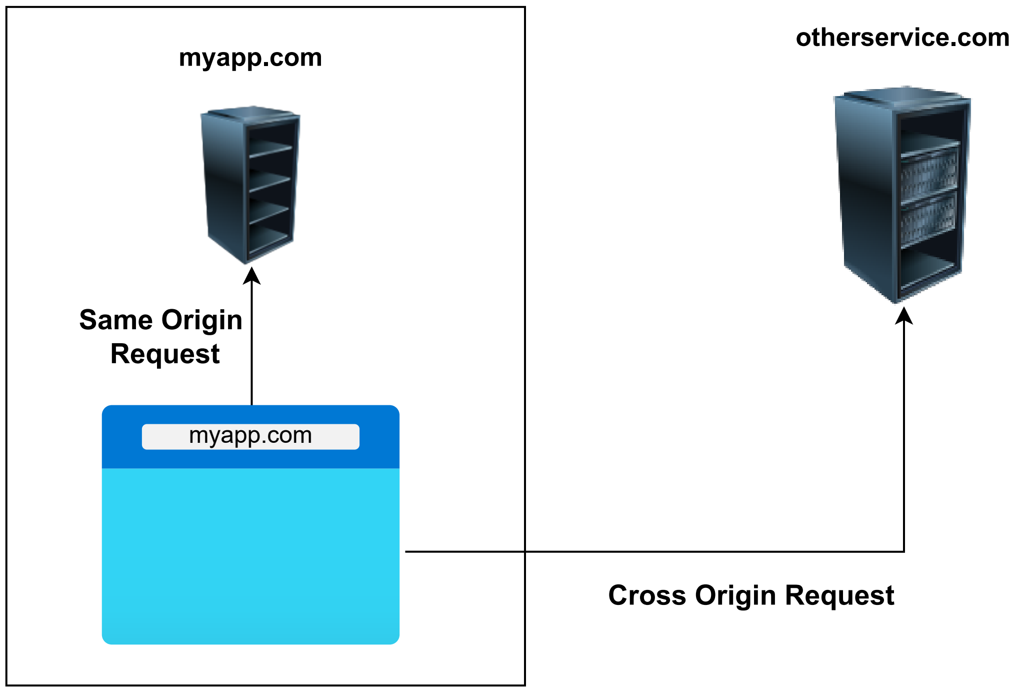
1. Hides internal Technical Architecture complexities from Clients

2. Allows to re-design the backend services without affecting clients

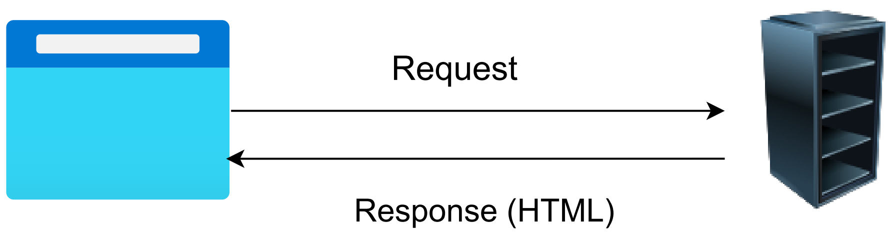
3. Dynamic routing to different API versions

4. Centralized Security, Rate Limiting, Timeout, etc.

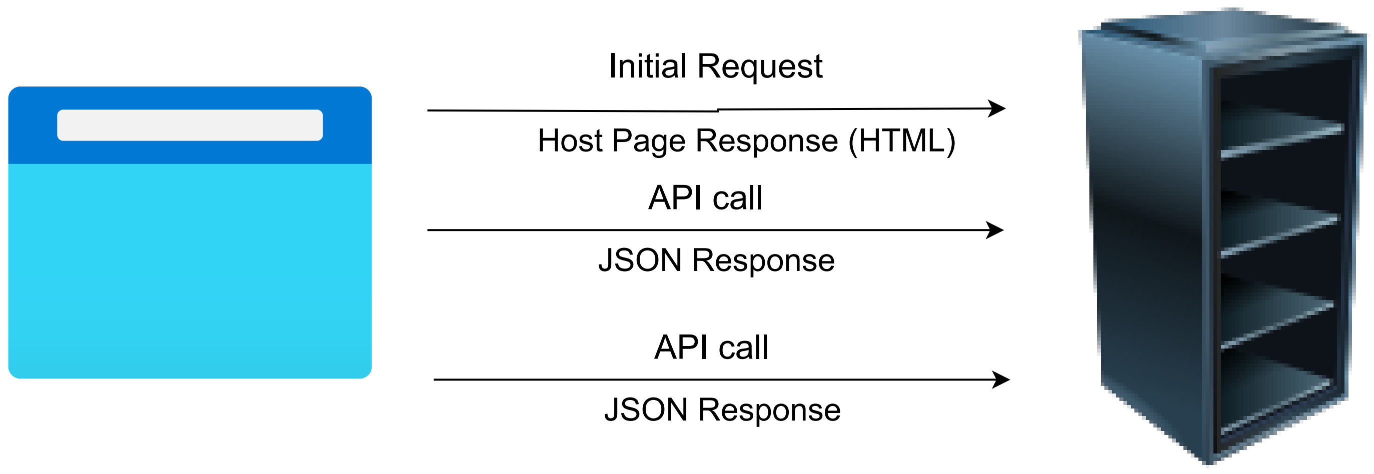
Cross Origin Resource Sharing



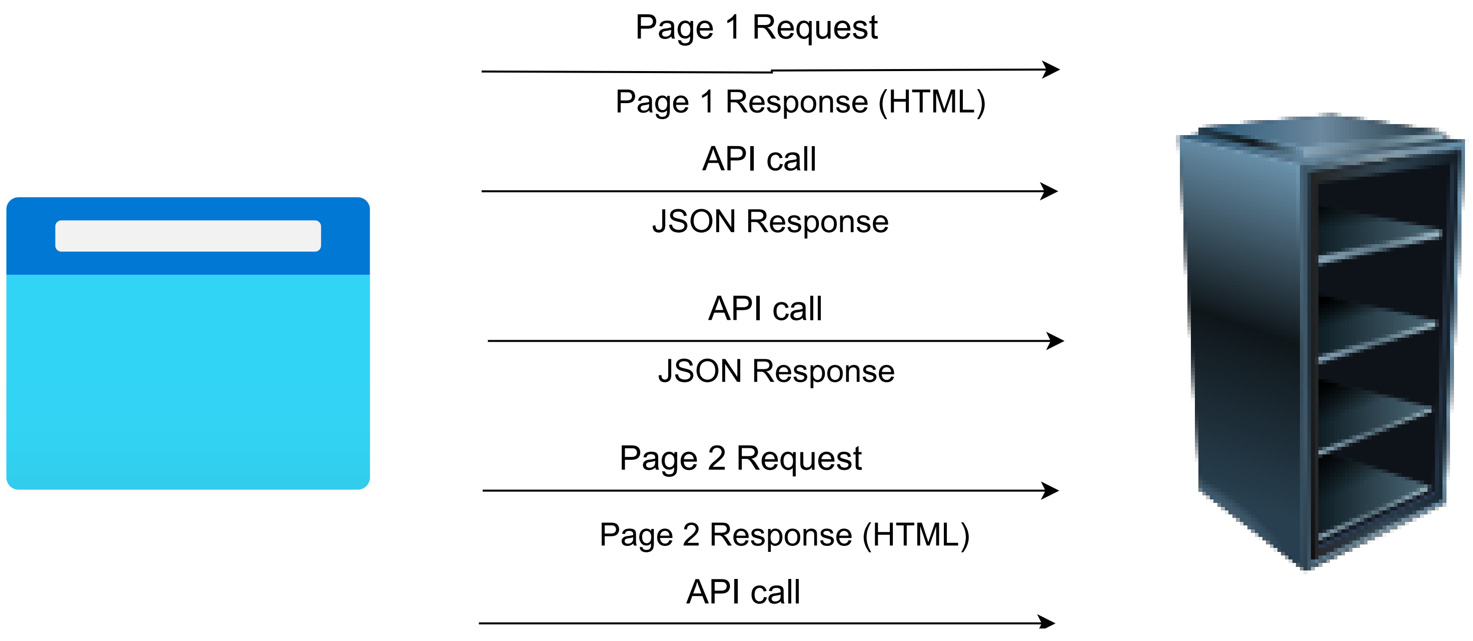
Traditional Server Rendered Web Apps



Single Page Applications (SPAs)

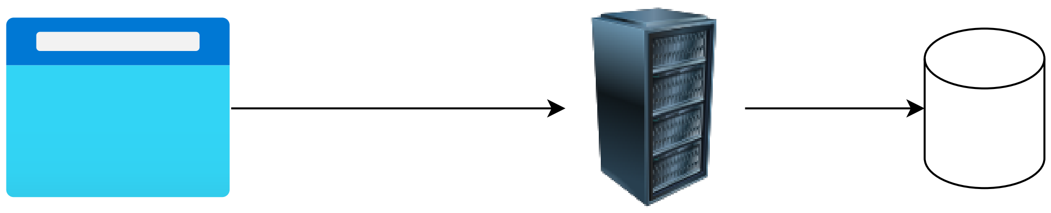


Multi Page Applications



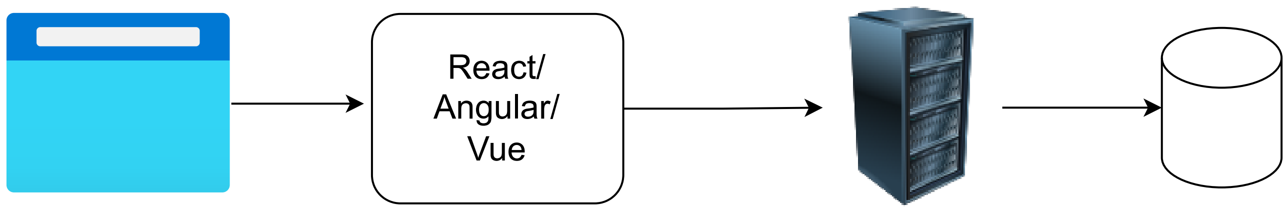
JSON Response

Single Server Rendered Web Application



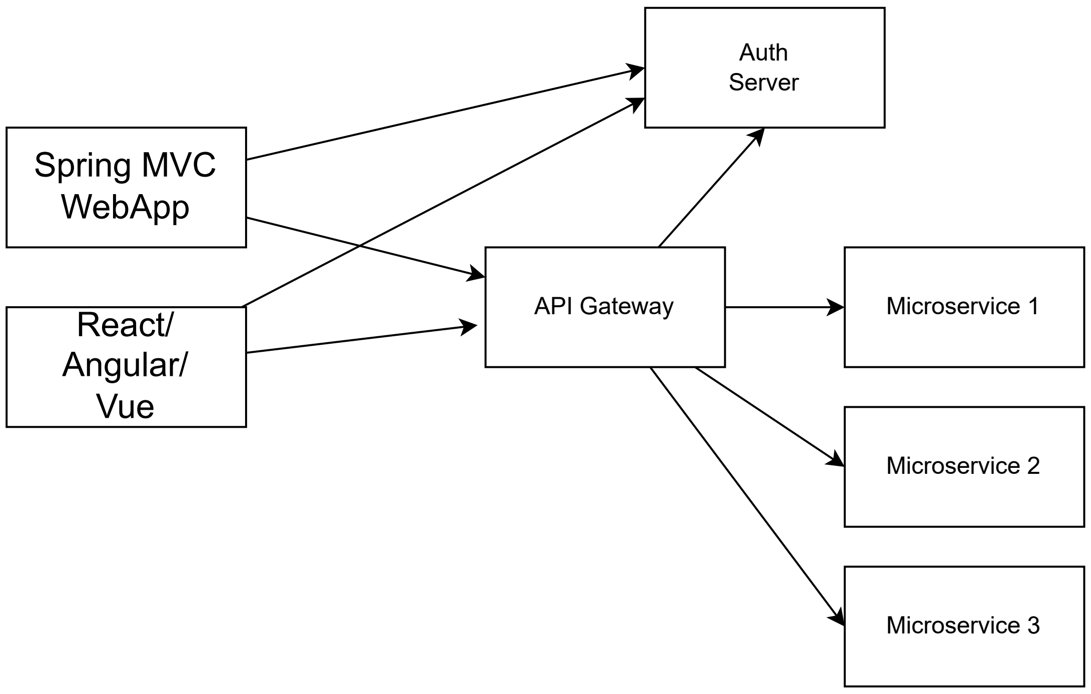
HTTP Session or store client state on DB

SPA Frontend and REST API Backend



Server creates JWT token and the token is stored at client side and pass with each request

Microservices



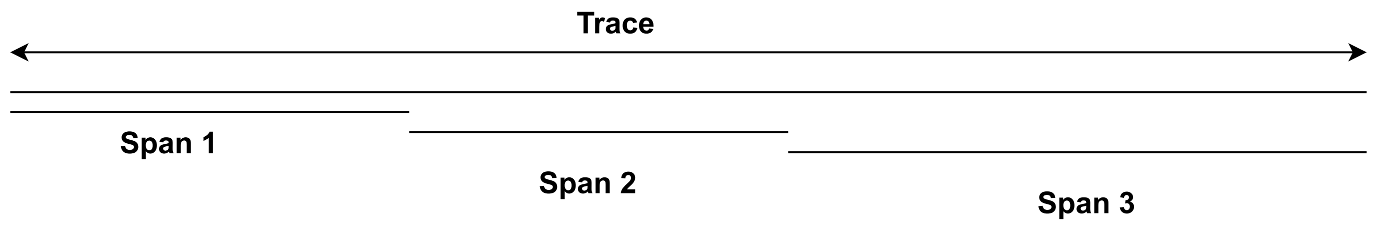
1. Client logins using Auth Server Login page and get access_token and id_token

2. Client applications include the access_token for each API request as Authorization header

3. API Gateway or Microservices validates the access_token and process or reject the request

Observability

1. Logs
2. Metrics
3. Traces


Trace
Span 1
Span 2
Span 3