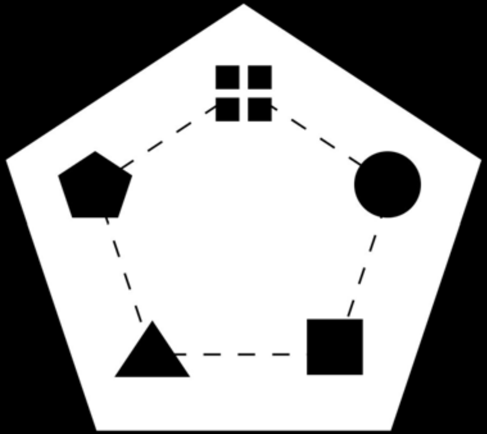
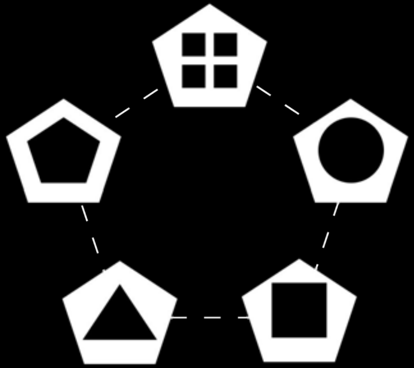




DECEMBER 05 2024

# From Monolith to Microservices: A Guide to Seamless Transitions

**By**
Daniil Koshelev

CONF42

# Agenda

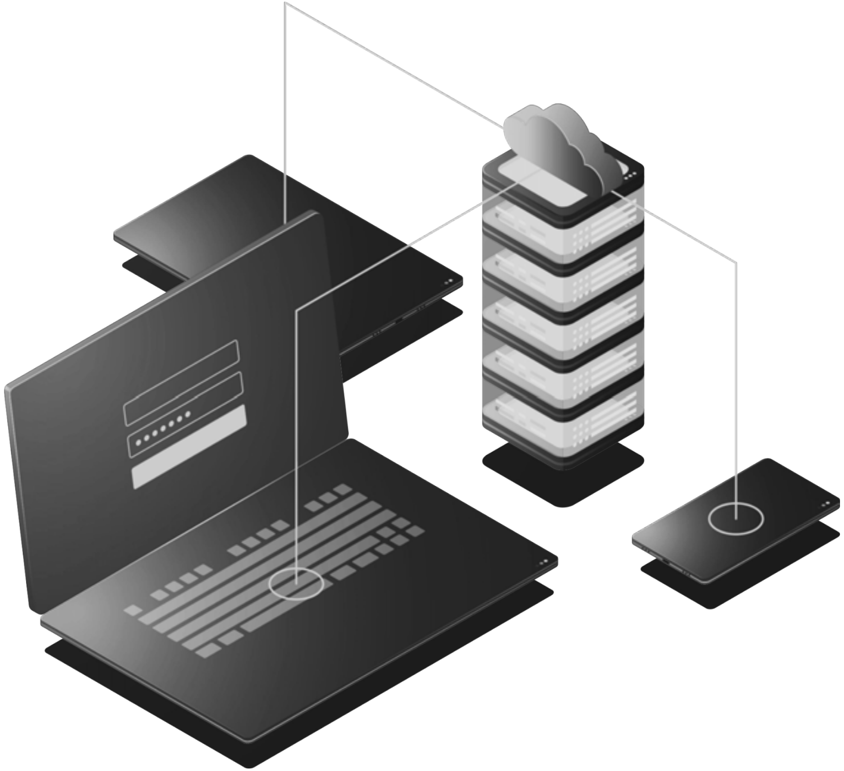
01 Architectures overview

02 Relation to highload

03 Design for failure

04 Challenges and solutions

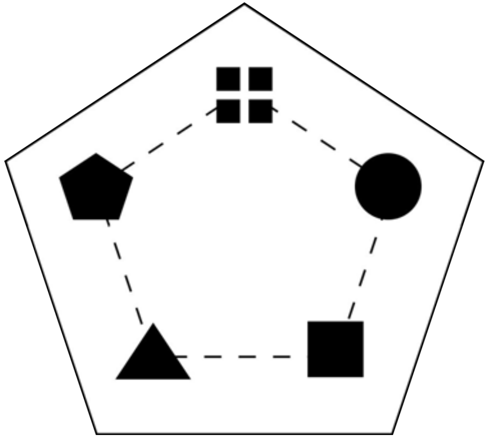
05 Practical Example



Software Architecture

# A method for documenting decisions made for the implementation of an information system

Decisions are presented as a connection between several components. They are provided in a form that is accessible for reuse. Architecture is always considered from different perspectives.



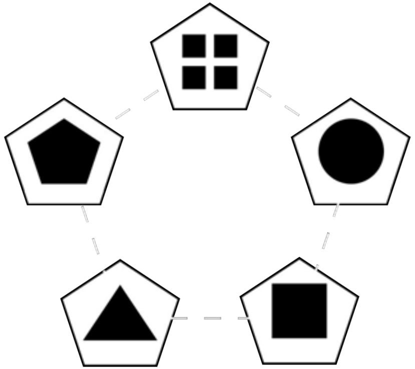
## Monolithic Architecture

A monolithic architecture is a traditional software design approach where the entire application is built as a single, unified unit.

| + | - |
|---|---|
| Simpler Development | Scalability Issues |
| Ease of Deployment | Tight Coupling |
| Performance | Slower Development |
| Centralized Management | Limited Agility |
|  | Deployment Risks |

## Microservices Architecture

A microservices architecture divides an application into a collection of small, loosely coupled, and independently deployable services



| + | - |
|---|---|
| Independent Scaling | Complexity in Management |
| Faster Development and Deployment | Inter-Service Communication Overhead |
| Technology Diversity | Security Challenges |
| Fault Isolation | Data Consistency |
| Agility and Flexibility | Cost |

## Here's when a microservice architecture might be a better choice

1. Scalability and High Demand
2. Complex, Evolving Applications
3. Independent Deployment
4. Team Structure and Ownership
5. Need for Technology Diversity
6. Fault Tolerance and Isolation
7. Global or Distributed Operations
8. Integration with Third-Party Services
9. Agile Development and Innovation
10. Legacy System Modernization

## Microservices might not fit well if

1. The application is simple and small
2. Team expertise is limited
3. Infrastructure resources are constrained
4. Low development velocity is acceptable

## Highload

**Compute-intensive:**
The bottleneck is the **CPU**.
**Data-intensive applications (DIA):**
**Challenges:**

- Volume of data
- Quality of data
- High rate of change
- High level of complexity

## Highload

There is no clear definition of highload.

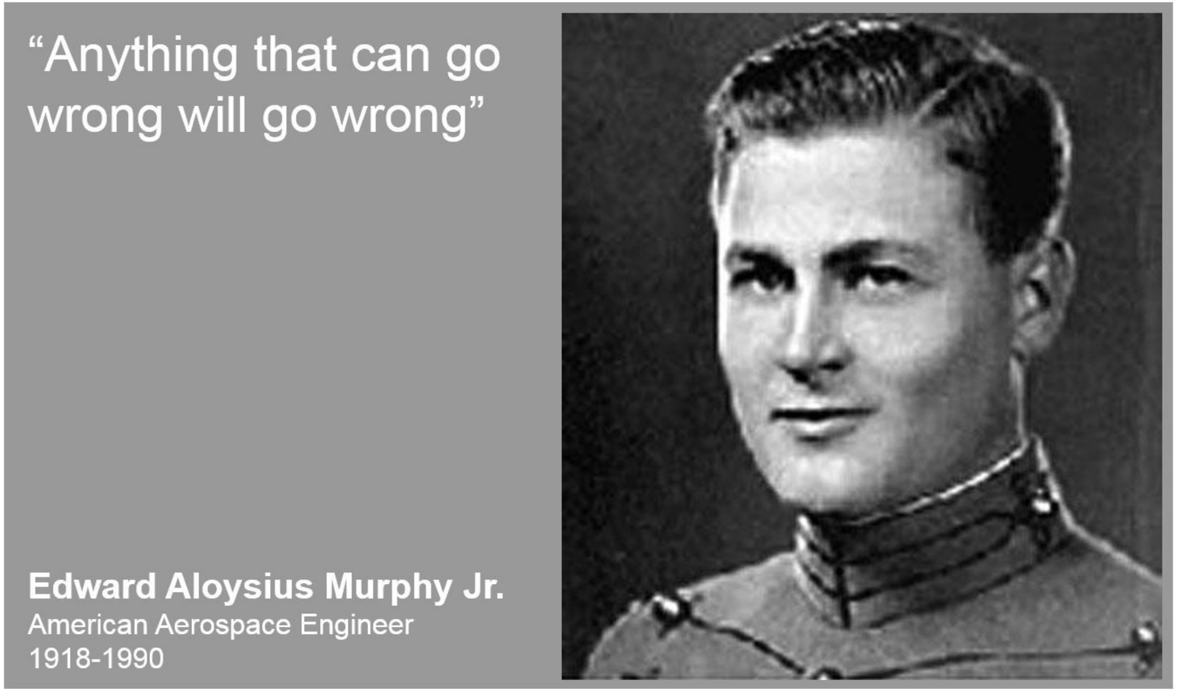
**Signs of highload:**

- The system can no longer handle the current load.
- Common approaches are insufficient.
- There is an urgent need to scale the infrastructure.
- A single server is not enough to serve the customers.
- Hardware cannot cope with the increased loads.
- Existing tools and resources cannot solve the emerging problems**.**

## Key questions

1. Reliability
2. Scalability
3. Ease of maintenance

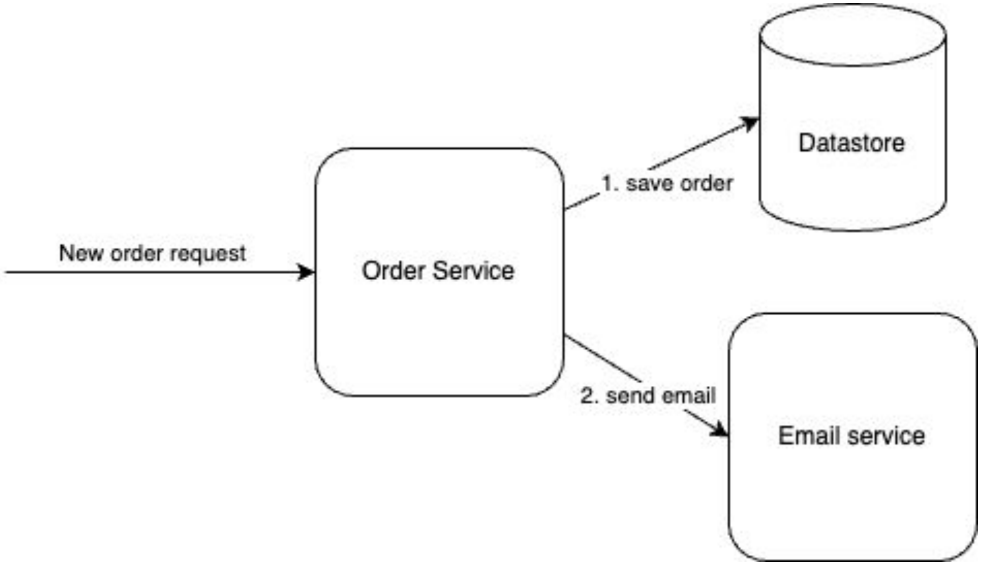
## Design for failure



"Anything that can go wrong will go wrong"

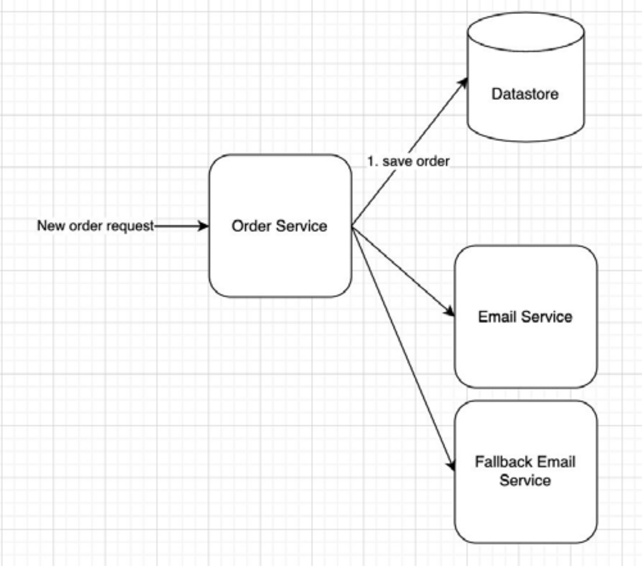
**Edward Aloysius Murphy Jr.**
American Aerospace Engineer
1918-1990

## Design for failure



Example – order processing service

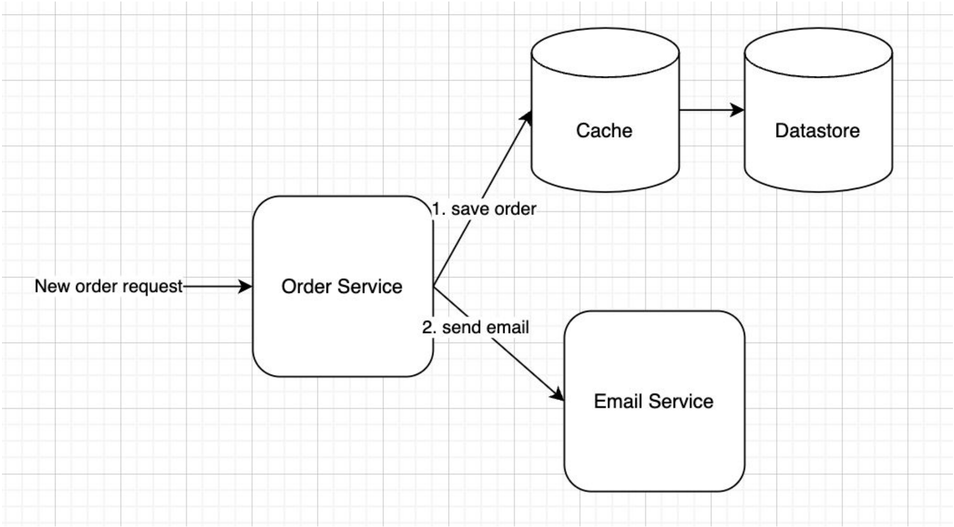
## Handling service degradations



In the case of a service failure – external or internal to the system – it must be properly handled.

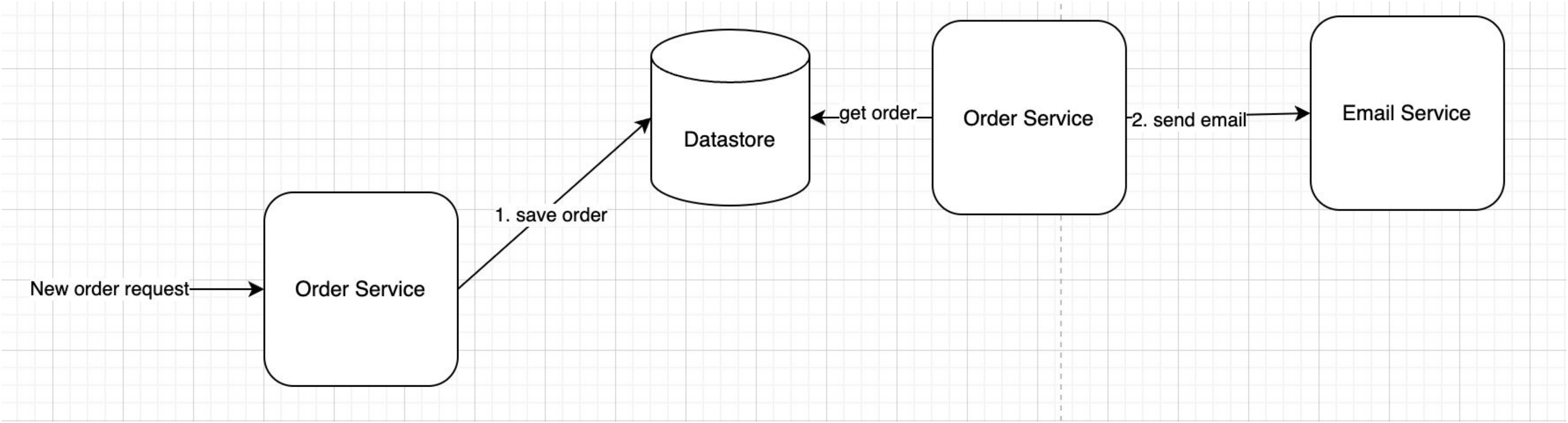
An good option for handling degradation is using a **fallback** service

## Handling service degradations



An good option for handling degradation is to use a **cache** service for the data storage. Multi-level caching is also possible

## Handling service degradations



If you need to save some data to persistent storage + send a message to a queue, you must use the **transactional outbox** template to implement **delivery guarantees**.
https://microservices.io/patterns/data/transactional-outbox.html

Code Complexity ⊘ Responsibilities ⊘ Caches ⊘ Distribute

Dilemma ⊘ Security ⊘

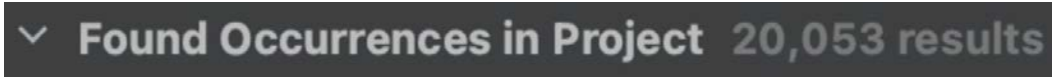
# Challenges

⊘ Pro

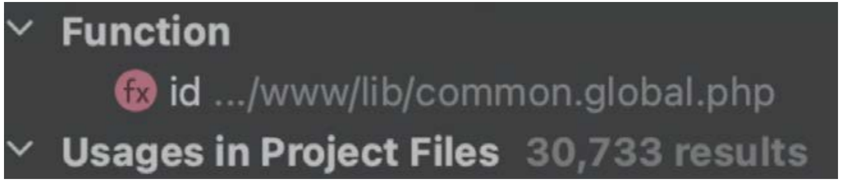
⊘ Data Migrations ⊘ Technologies ⊘ High Entry Barri

## Code Complexity

1. Tightly Coupled Code
2. Hard to Understand & Maintain
3. Slow Development & Deployment
4. Limited Flexibility
5. High Risk of Bugs
6. Low tests coverage
7. High entry barrier
8. Recruitment challenges



Found Occurrences in Project 20,053 results



Function

fx id .../www/lib/common.global.php

Usages in Project Files 30,733 results

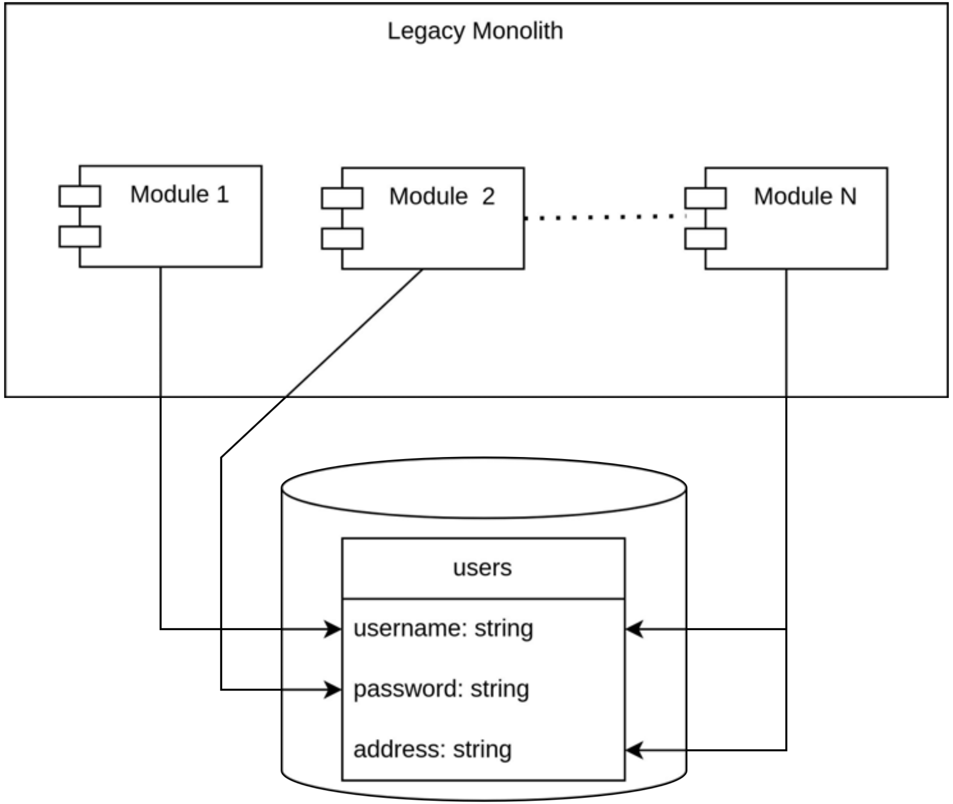
## Data Management and Decoupling

Monolithic systems often have tightly coupled data structures. Migrating to microservices requires segregating these data sources into smaller, independent modules.

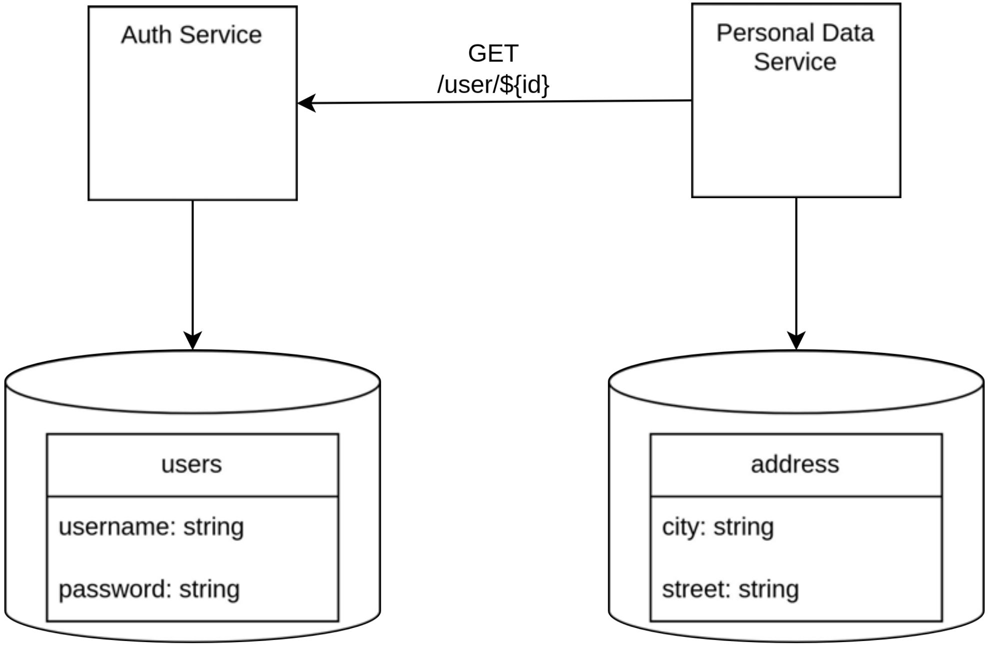
**Solutions**:

- Database-per-service pattern
  https://microservices.io/patterns/data/database-per-service.html
- Event sourcing or change data capture (CDC)
  https://microservices.io/patterns/data/event-sourcing.html
- Data replication and shared databases as temporary measures while gradually migrating

# Data Management and Decoupling



## Data Management and Decoupling



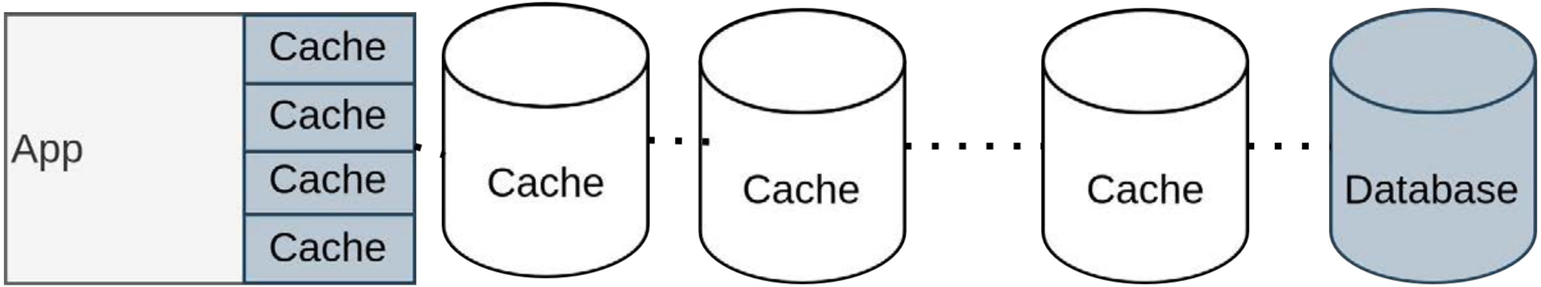
| users |
|---|
| username: string |
| password: string |

| address |
|---|
| city: string |
| street: string |

## Multilayered Caches

The method call may look like this:

`userService.GetUser(id)`, which encapsulates all the logic for working with the multi-layered cache.



**Question:** What problems do you see with such schemes?

## Service Communication and Interdependencies

Microservices rely heavily on communication between services, introducing latency, failure points, and complex dependencies.

**Solutions**:

- Message queues (e.g., RabbitMQ, Kafka) to reduce coupling
- Service discovery tools (e.g., Consul, Eureka) for seamless inter-service communication
  https://microservices.io/patterns/index.html#service-discovery
- Circuit breakers and retries with backoff to handle failures
  https://microservices.io/patterns/reliability/circuit-breaker.html

### Security

Microservices increase the number of communication endpoints, exposing the system to vulnerabilities.

**Solutions**:

- API gateways to centralize authentication, authorization, and request validation https://microservices.io/patterns/apigateway.html
- OAuth 2.0 and token-based authentication (e.g., JWT) for secure access control https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-01
- Penetration testing and audits

## Deployment and Continuous Integration/Continuous Deployment

Deploying and managing multiple microservices is complex, especially when transitioning from a monolithic system.

**Solutions**:

- Containerization (e.g., Docker) and orchestration tools (e.g., Kubernetes) for consistency in deployments.
- CI/CD pipelines to automate builds, tests, and deployments.
- GitOps for infrastructure as a code

## Monitoring and Debugging

Distributed systems are harder to monitor and debug due to numerous services and potential points of failure.

**Solutions**:

- Centralized logging using tools like ELK Stack or Fluentd
- Distributed tracing solutions (e.g., Jaeger, Zipkin) https://microservices.io/patterns/observability/distributed-tracing.html
- Service mesh technologies (e.g., Istio, Linkerd) for observability and traffic management

### Organizational Resistance and Skill Gaps

Teams may resist change due to a lack of familiarity with microservices or fear of increased workload.

**Solutions**:

- Training programs and workshops for teams.
- Transition of responsibilities to smaller, cross-functional teams
- Start with a pilot project

## Managing Legacy System Integration

Legacy systems often need to remain operational during the migration process, leading to challenges in integration.

**Solutions**:

- Strangler patterns
- Facades or adapters to bridge monolithic and microservices environments
- Maintain compatibility layers

## Performance and Scalability

Microservices introduce network overhead and require careful scaling strategies.

**Solutions**:

- Lightweight protocols like gRPC
- Horizontal scaling and autoscaling features https://microservices.io/patterns/deployment/service-deployment-platform.html
- Performance testing

### Dependency Management

Managing dependencies between microservices is complex and can lead to cascading failures.

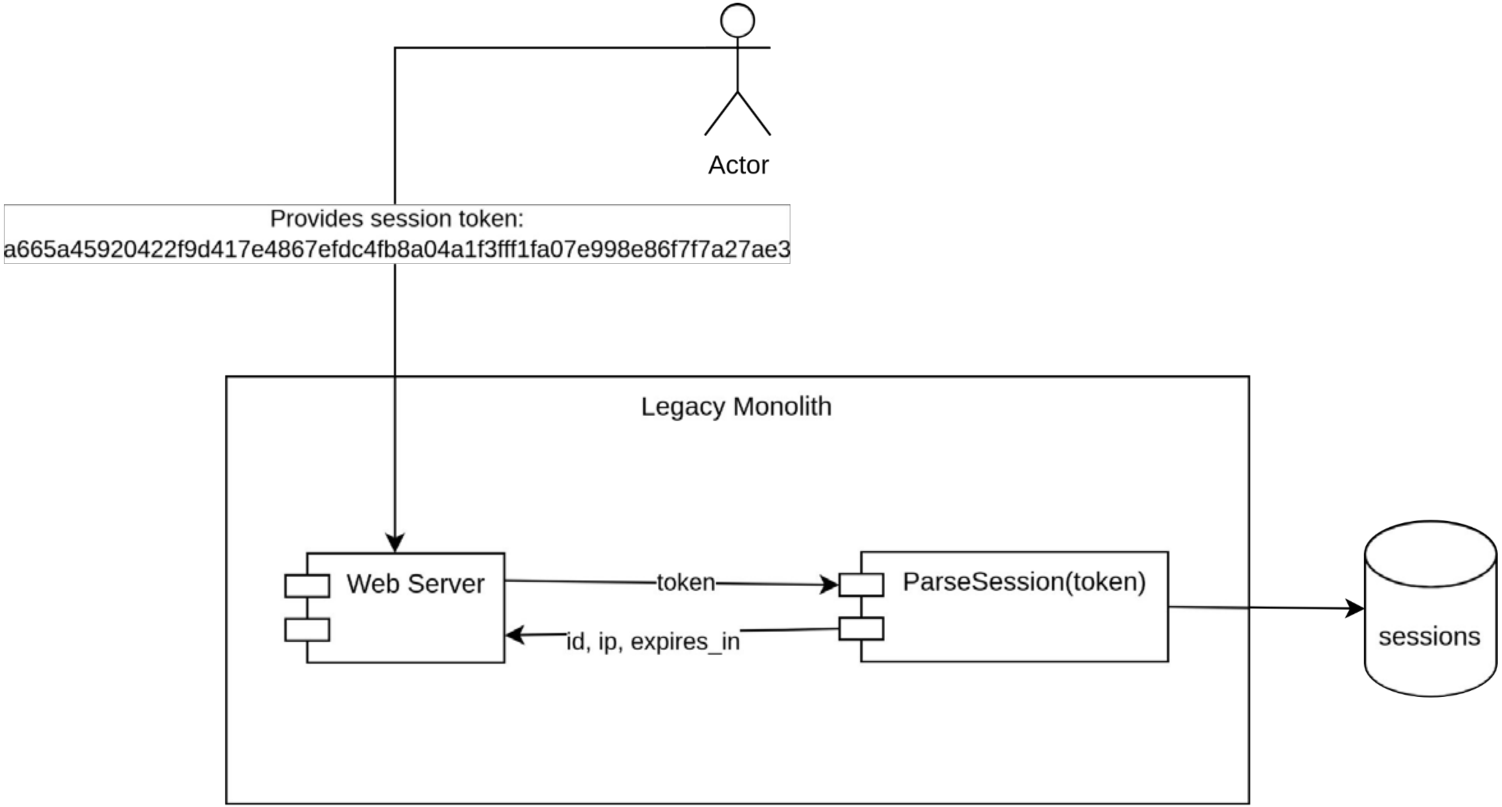
**Solutions**:

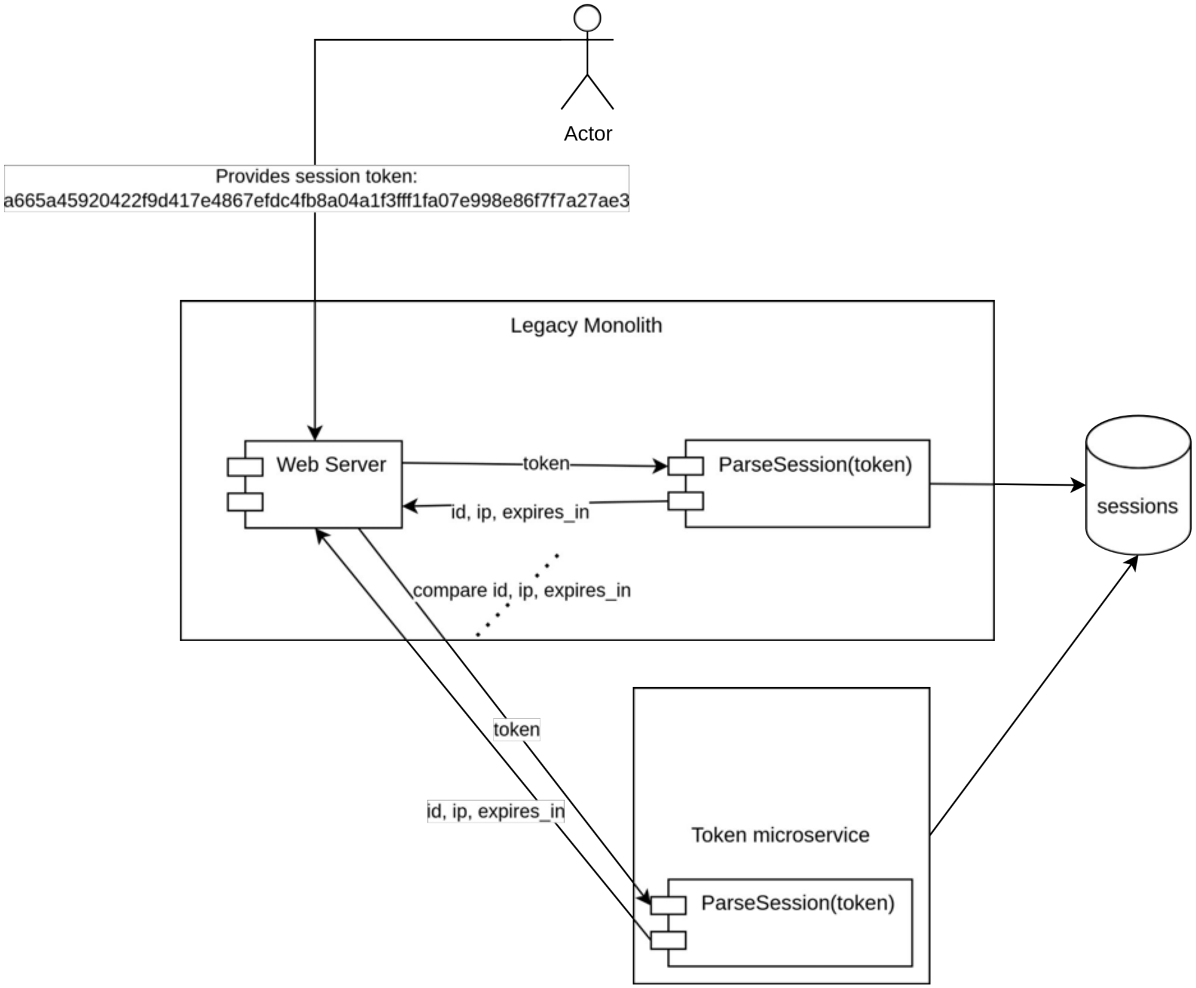
- Event-driven architectures https://microservices.io/patterns/data/domain-event.html
- Clear service contracts (e.g., API specs using OpenAPI/Swagger)
- API versioning for backward compatibility

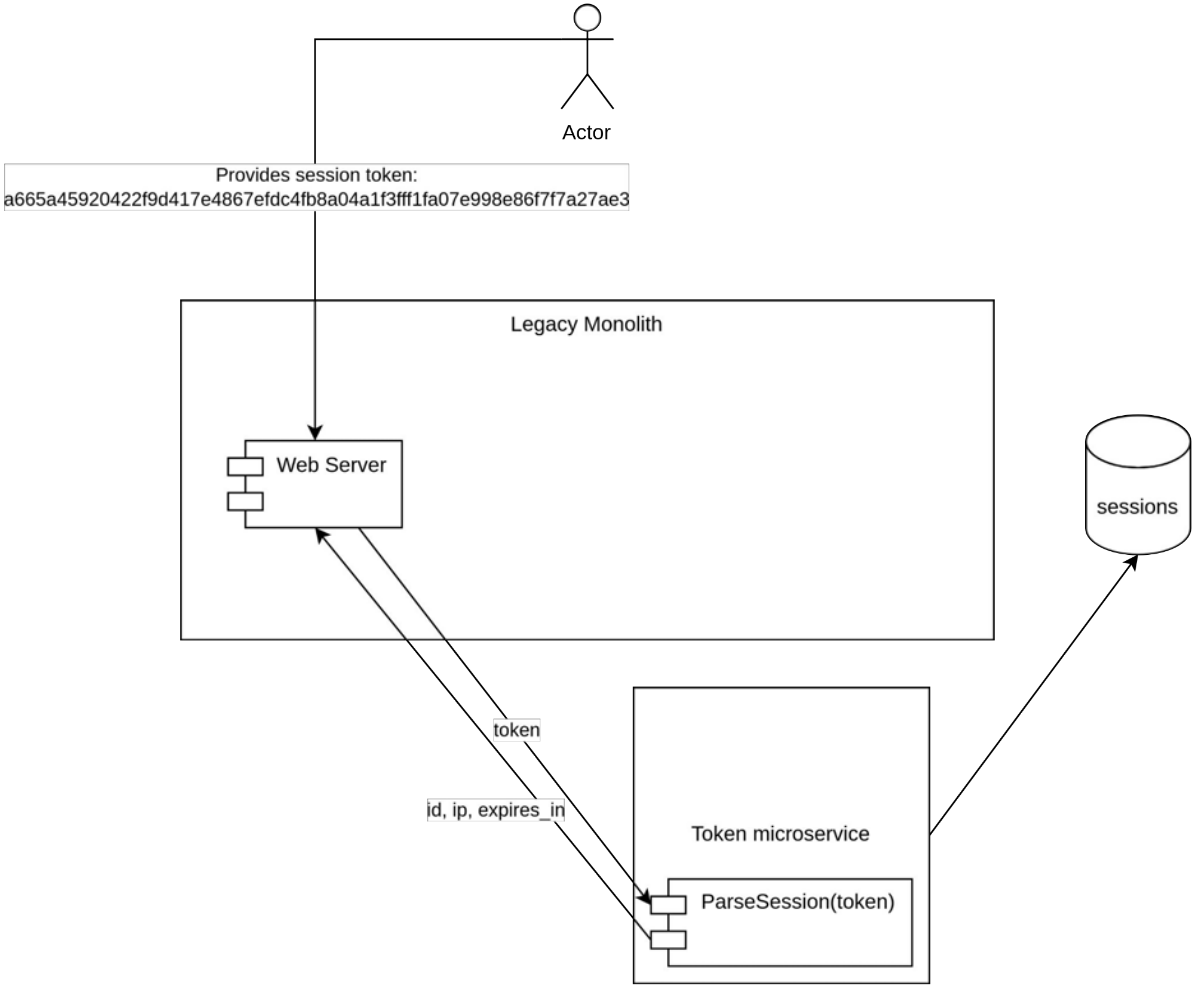
# Practical Example Token management service

Will generate and check API access tokens

# User wants to view some content and provides his token




Actor
Provides session token:
a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3
Legacy Monolith
Web Server
token
ParseSession(token)
id, ip, expires_in
sessions
compare id, ip, expires_in
token
id, ip, expires_in
Token microservice
ParseSession(token)



Actor

Provides session token:
a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3

Legacy Monolith

Web Server

sessions

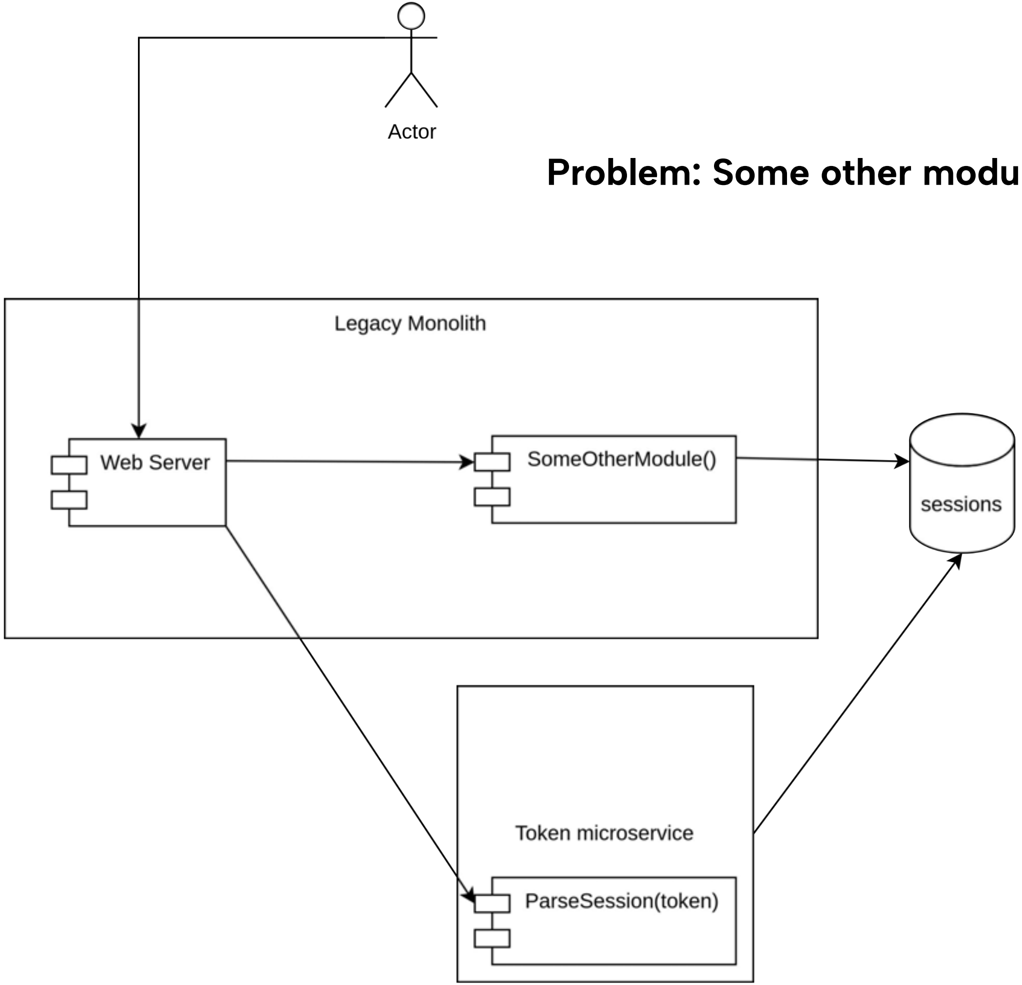
token

id, ip, expires_in

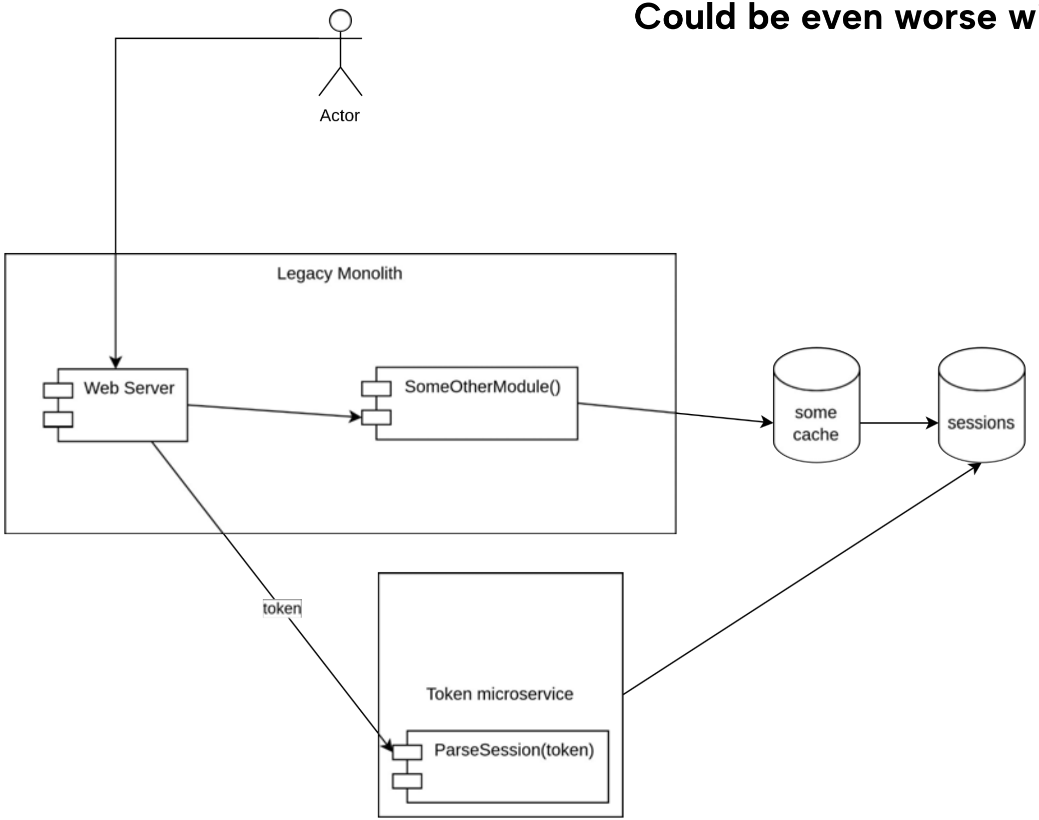
Token microservice

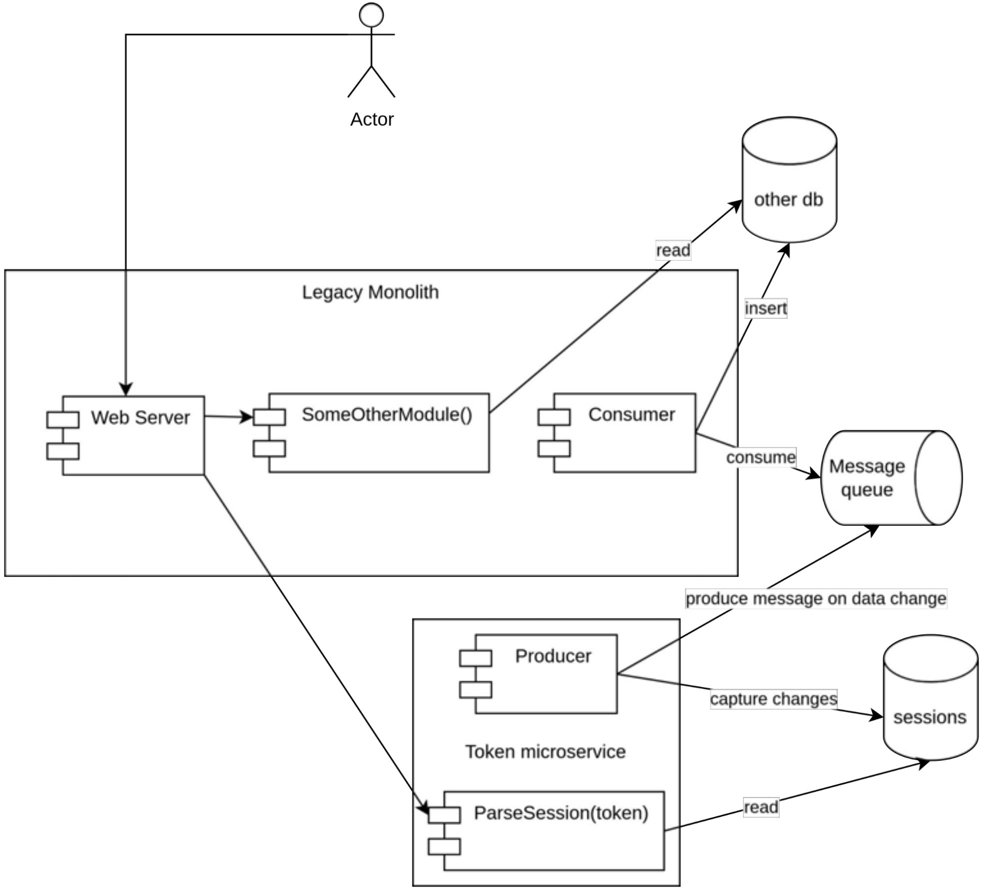
ParseSession(token)

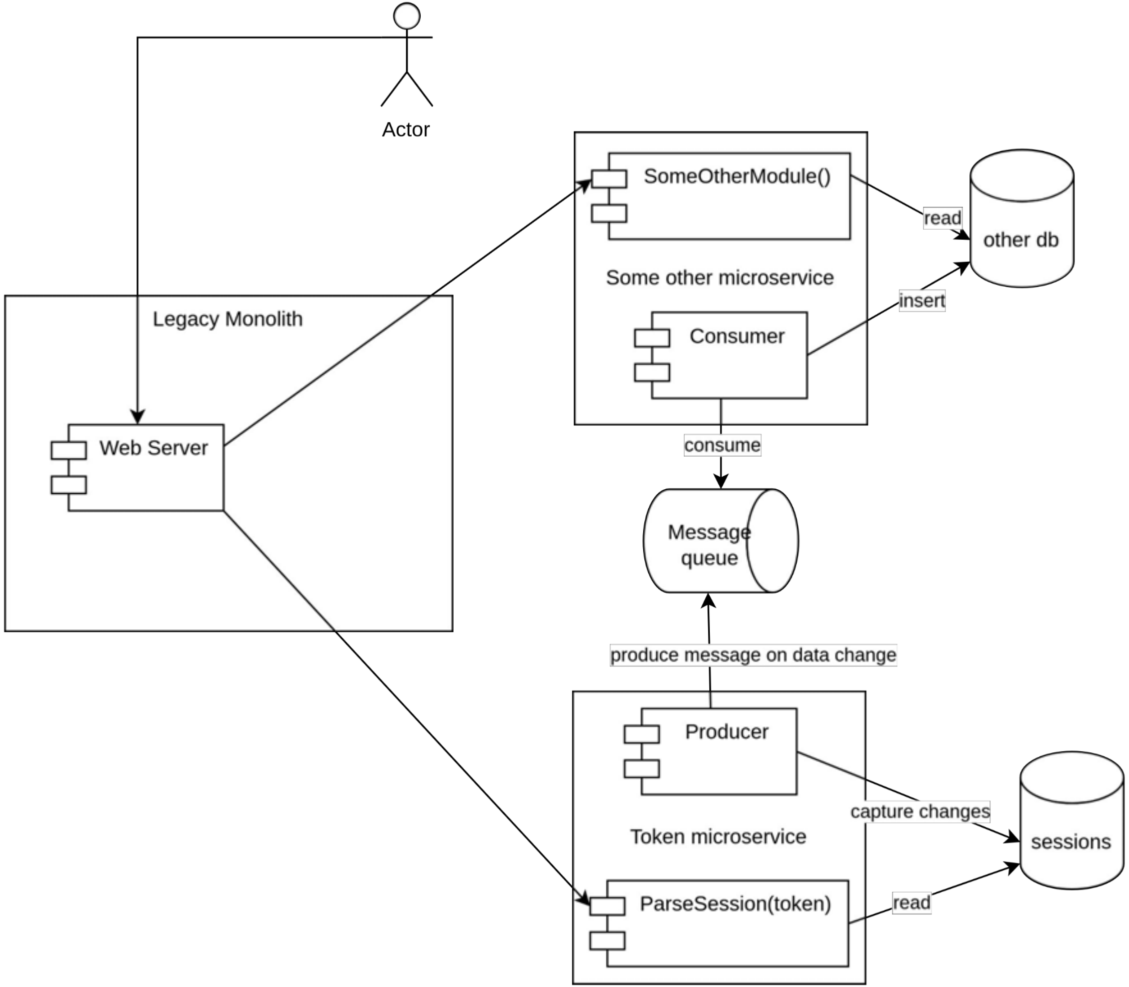
## Problem: Some other module uses sessions database



## Could be even worse with different caches




Actor
other db
read
insert
Legacy Monolith
Web Server
SomeOtherModule()
Consumer
consume
Message queue
produce message on data change
Producer
capture changes
sessions
Token microservice
ParseSession(token)
read


Actor
Legacy Monolith
Web Server
SomeOtherModule()
Some other microservice
Consumer
read
insert
other db
consume
Message queue
produce message on data change
Producer
Token microservice
ParseSession(token)
capture changes
read
sessions

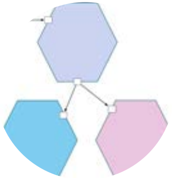
# References

### Designing Data-Intensive Applications

By M. Kleppmann

### Building Microservices with Go.

By Nic Jackson



### Microservices.io

https://microservices.io/patterns/index.html

### Distributed Systems 4th edition (2023)

https://www.distributed-systems.netaindex.php/books/ds4/

# Thank you for your attention!

**Contacts:**
LinkedIn**:** https://www.linkedin.com/in/daniil-koshelev/
Telegram**: @kdaniil405**
Email**: dkoshelev405@gmail.com**

CONF42