Robert Sedor
Chief Architect Application Development

Jeremy Davis
Chief Architect Application Development

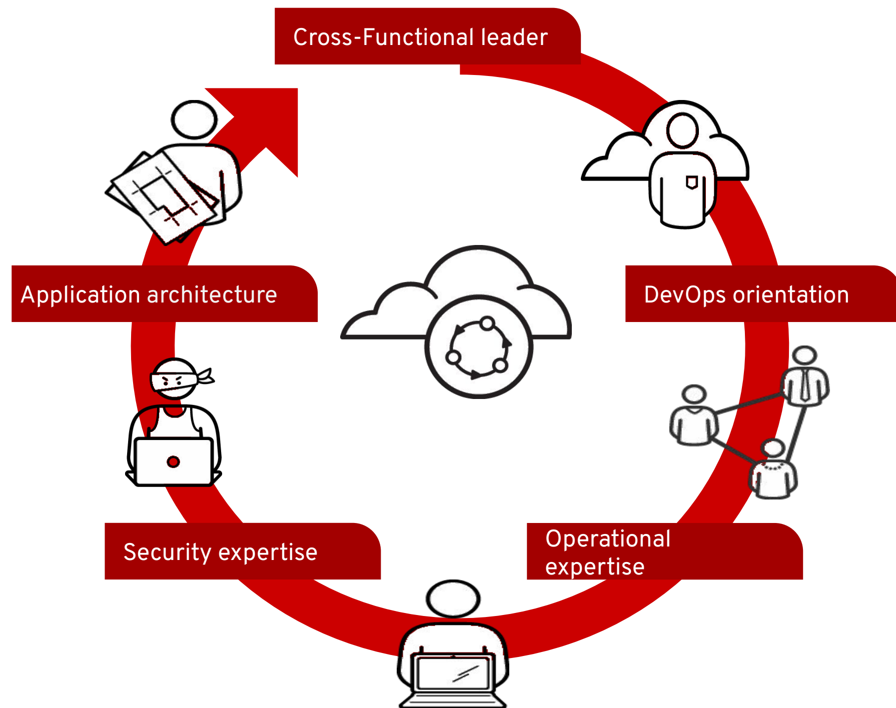
Agenda

- Coupling
- Modularity
- Decomposition
- Reuse
- Distributed Data
- Sagas
- Conclusion

Red Hat

What is Application Modernization

- Adapting legacy applications to deliver value for modern business needs
- Leverage field tested strategies, patterns and practices to
  - Eliminate risk and speedup execution
  - Adopt methodical engineering approaches to evolve systems
  - Address different application types across the business portfolio
  - Eliminate guesswork and create predictable results



Red Hat

Application Modernization:

Coupling

Red Hat

What types of coupling are there?

- Static coupling
  - How do static dependencies resolve within the architecture via contracts?
  - This includes frameworks, libraries, operating system dependencies via transitive dependency management in order to operate
- Dynamic coupling
  - How do components and services communicate with each other at runtime? e.g. either synchronously or asynchronously

Red Hat

Independently Deployable

- Independently deployable - a separate deployable unit within an architecture
    - Monolithic architectures are deployed as a single unit
        - This could be a unit such as a single executable or a distributed monolithic architecture
    - Microservices tend to deploy services independently
        - A service within a microservice architecture represents its own architectural unit contingent on coupling
    - Representing architectural units as deployable assets
        - Creates boundaries representing architectural concerns, developer concerns and operations concerns
    - Understanding the scope
        - Architects understand coupling characteristics
        - Developers understand a scope of behavior
        - Operations teams understand the deployable characteristics

Independently Deployable

- Understanding the architectural units representing static coupling
  - Architects can work for proper granularity of services
  - Developers can determine the best set of trade-offs for granularity of services
  - Operationally, we can determine trade-offs around deployability such as:
    - Release cadence
    - Effect of deployability on other services
    - What practices do we need to involve in the deployment and deployment boundaries with other units in our systems or other systems
- Understanding independent deployability forces us to consider common coupling points such as
  - Databases - e.g. a shared database, distributed database, SQL2.0 database etc...
  - User interfaces

High Functional Cohesion

- Traditionally, high functional cohesion refers to proximity of related elements
    - Classes, components, services etc...
- From a domain standpoint, high functional cohesion refers to the goals of a bounded context in domain-driven design
    - Behavior and data that implements a particular domain workflow

Red Hat

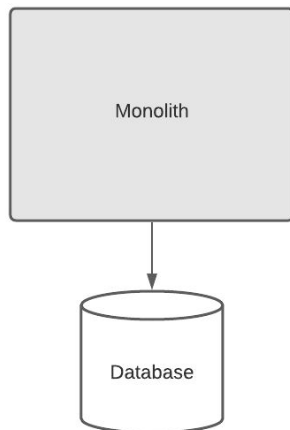
High Functional Cohesion

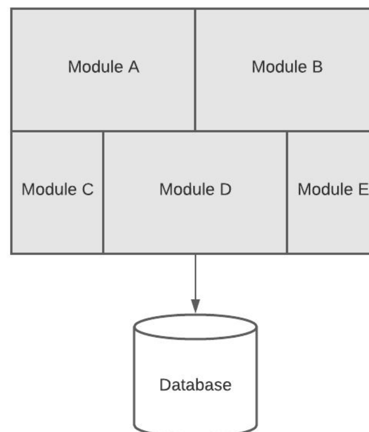
- A monolithic architecture is a single deployable unit
  - A monolith isn't highly functionally cohesive but includes the functionality of the entire system
- A microservices architecture ideally has each service
  - Model a single domain or workflow
- Cohesion isn't about how services interact to perform work but how independent and couple one service is to another service

Red Hat

High Static Coupling

- High static coupling is about how elements internally to each architectural unit are tightly wired together which represents an aspect of contracts
- REST is a contract format
  - But IP addresses and URLs are also contracts and coupling points
- What is the measure of static coupling?

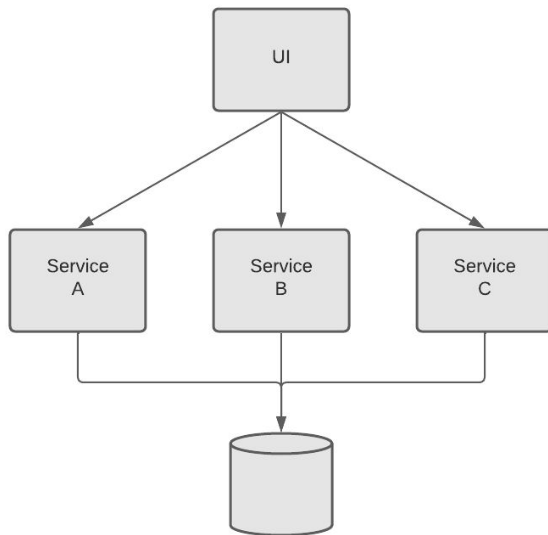




Red Hat

High Static Coupling

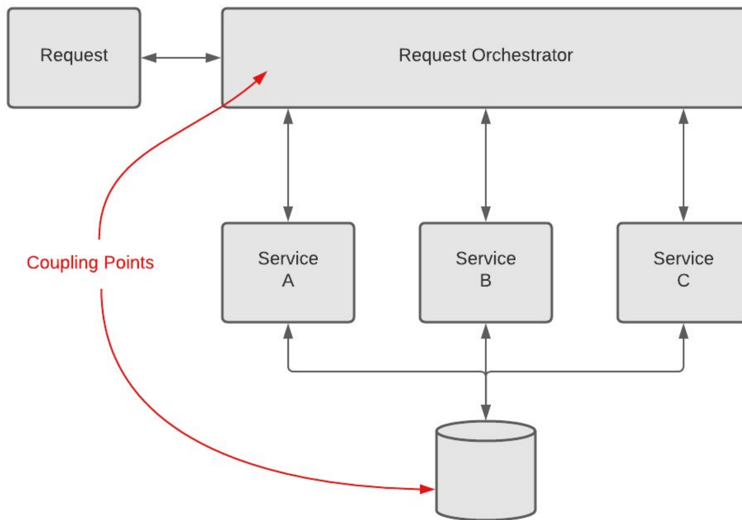
- Distributed architectures typically feature decoupling at the component level
- A service based architecture may display isolation at the component level like microservices but is still high static coupling because it has a single relational database



Red Hat

High Static Coupling

- Distributed architectures create the possibility of lower static coupling but any holistic coupling point necessary for an architecture to function forms an architecture coupling point around it



Red Hat

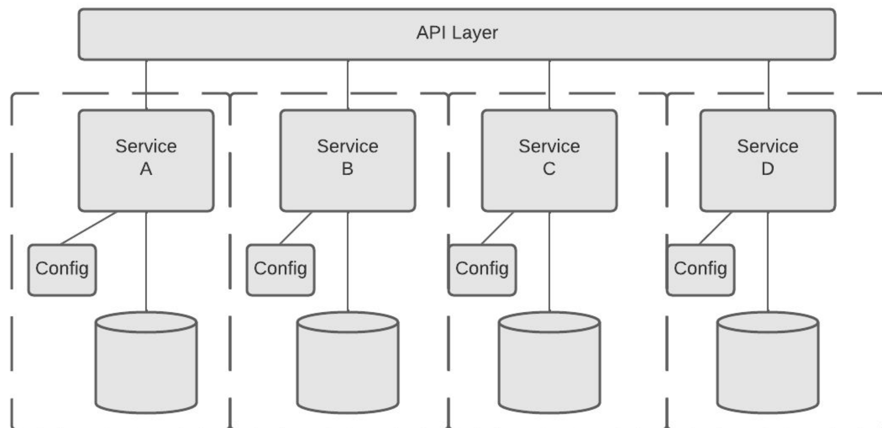
High Static Coupling

- Static coupling assesses the coupling dependencies between architectural and operation components
  - The operating system, data store, message broker, container orchestration and all other operational dependencies for the static coupling points of an architecture
- A microservices architectural style features highly decoupled services including data dependencies
  - Architects and developers using microservices will favor high degrees of decoupling and take care not to create coupling points between services

Red Hat

High Static Coupling

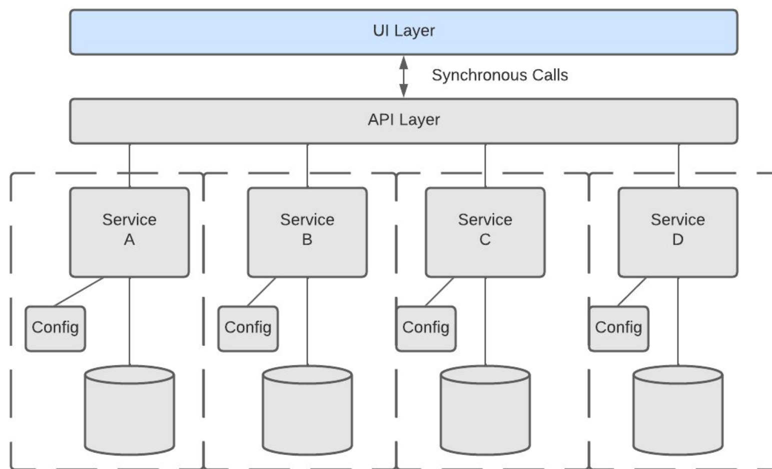
- With the microservices style, each service acts as a bounded context and have its own set of architecture characteristics
    - One service may have higher scalability or security
- Granular architecture and high degrees of decoupling allow teams to work on a service to move as quickly as possible without breaking other dependencies



Red Hat

High Static Coupling

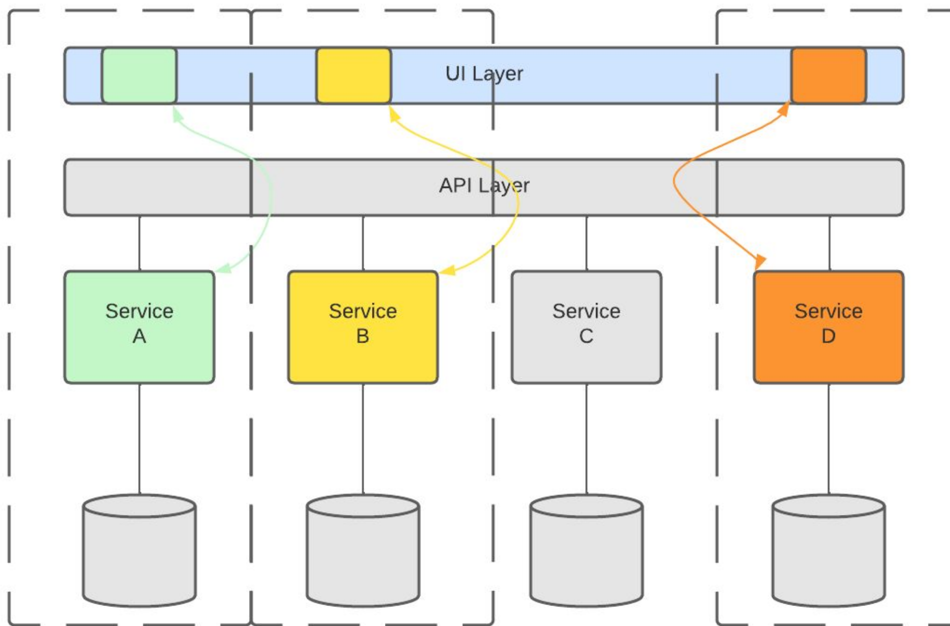
- Adding the UI layer may reintroduce a coupling between the front and back-ends
- Considerations to reduce this coupling may be a micro-frontend and reactive UI facilitating loosely coupled communication between components using events



Red Hat

High Static Coupling

- Using a micro-frontend, we can decouple the architecture again and allow for that loosely coupled communication between components



Red Hat

Dynamic Coupling

- What are the behaviors of architectural units as they interact with one another to form workflows in a system?
- How services call one another creates trade-off decisions because it represents a multidimensional decision space influence by three factors:
    - *Communication* - The type of connection synchronicity used: synchronous or asynchronous
    - *Consistency* - Whether the workflow communication requires atomicity or can utilize eventual consistency
    - *Coordination* - Whether the workflow utilizes and orchestrator or whether the services communicate via choreography

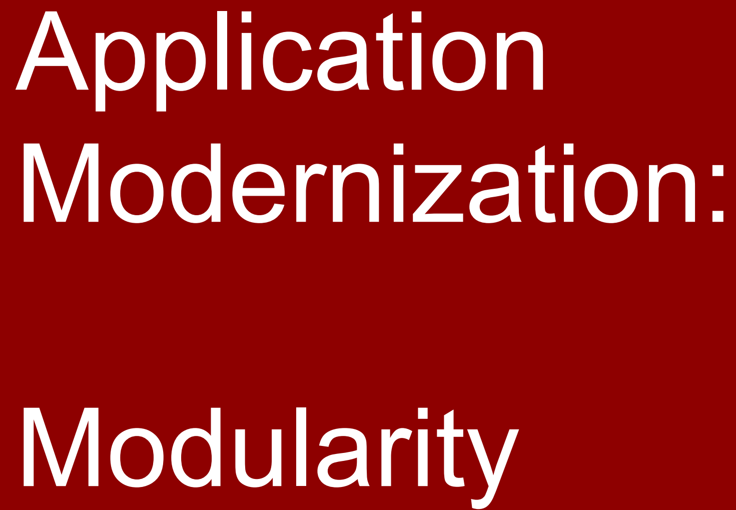
Dynamic Coupling - Communication

- A fundamental question for an architect regarding communication between two services is should the communication be synchronous or asynchronous
  - *Synchronous communication* requires the requestor to wait for a response from the receiver
    - The calling service makes a call (using a protocol such as gRPC) and blocks (does no further processing) until the receiver returns a value (or status or error condition)
  - *Asynchronous communication* occurs when the caller posts a message to the receiver (typically via a mechanism such as a message queue) and when the caller gets acknowledgement the message will be processed, it returns to work.
    - If the request required a response, then the receiver would use a reply queue to asynchronously notify the caller of the result
    - There are other implementations using asynchronous communications without message queues by using libraries and frameworks

Red Hat

Trade-offs and Techniques

- Although static coupling is easier to manage, consider
  - Frameworks and libraries such as Quarkus for reactive services
  - Messaging between services
  - Versioning and deployment options using Kubernetes
- Dynamic couple is more difficult to manage
- Whether using static or dynamic coupling there will be some considerations around communication and effect on
  - Synchronization
  - Error handling
  - Transactionality
  - Scalability
  - Performance



Red Hat

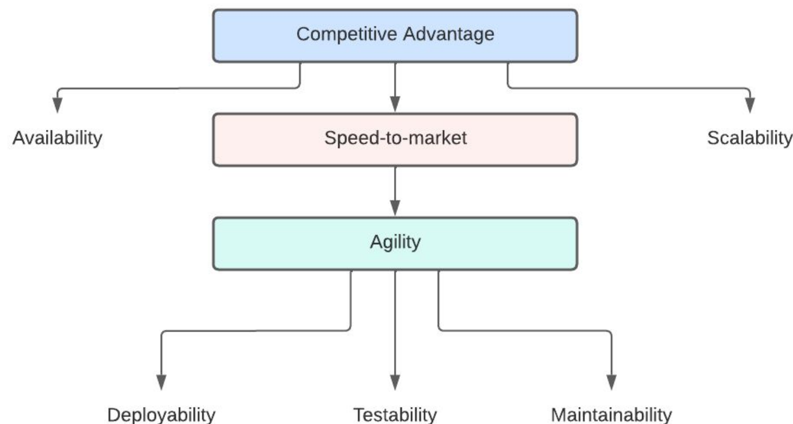
Modularity

- Large monolithic single deployment systems generally do not provide the level of scalability, agility and extensibility required to support today's business use-cases.
- As a monolithic application grows, it consumes more and more resources; threads, memory, CPU and storage resources. If it reaches a point where we need to increase capacity by introducing a second server, the new server will contain the same amount of resource requirements as the first server and continue to grow.
- An aspect of modularity is breaking monolithic applications into separate and smaller parts to allow for more capacity for further scalability and growth while facilitating constant and rapid change to achieve a companies goals.

Red Hat

Modularity Drivers

- Architects shouldn't break a system into smaller parts unless clear business drivers exist
    - *Speed-to-market* - This is achieved through architectural agility and the ability to respond to change made up of maintainability, testability and deployability
    - *Competitive advantage* - This is achieved through speed-to-market combined with scalability and overall application availability and fault tolerance.
        - Fault tolerance is the ability of an application to fail and continue to operate and it is necessary to ensure that as parts of an application fail, other parts are still able to function as normal minimizing the overall impact to an end user



Red Hat

Modularity Drivers

- Modularity does not always mean a distributed microservices architecture

- *Maintainability*, *testability* and *deployability* may be achieved through monolithic architectures such as a modular monolithic architecture
    - A modular monolithic architecture may contain components that are groups into well-formed domains known as a domain partitioned architecture
    - A microkernel architecture may have functionality segregated into separate plug-in components allowing for smaller testing and deployment scopes

- The consideration should be “what is the business value” from modular architecture and not instantly moving to a highly distributed architecture

Modularity Drivers - Maintainability

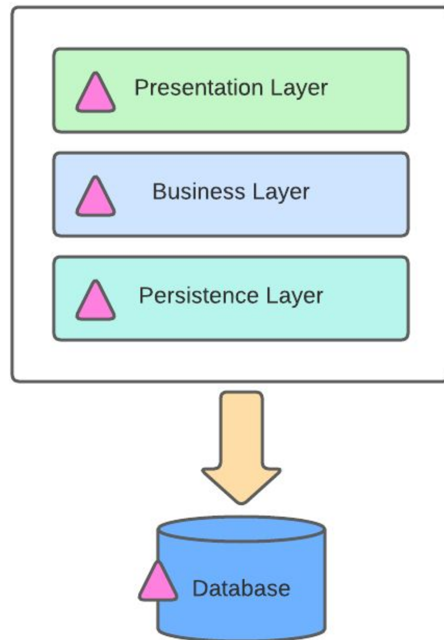
- Maintainability is a measure of ease of adding, changing or removing features
    - This can also involve applying internal changes including maintenance patches, framework upgrades, third-party upgrades etc.. *von Zitzewitz maintainability metric
- Typical metrics for determining maintainability
    - *Component coupling* - The degree and manner of which components know about one another
    - *Component cohesion* - The degree and manner to which the operations of a component interrelate
    - *Cyclomatic complexity* - The overall level of indirection and nesting within a component
    - *Component size* - The number of aggregated statements of code within a component
    - *Technical versus domain partitioning* - Components aligned by technical usage or by domain purpose

Alexander von Zitzewitz maintainability metric

Red Hat

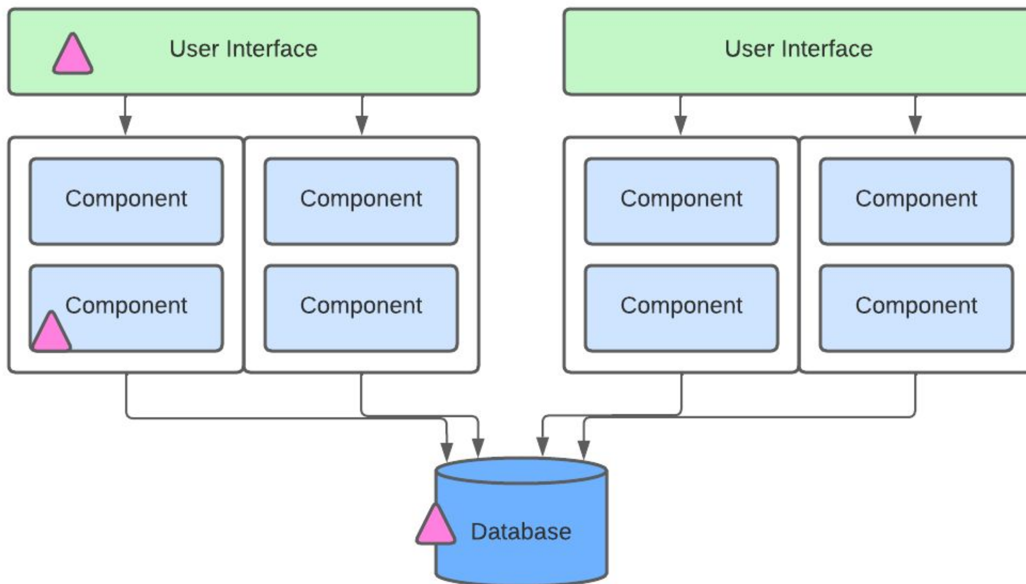
Modularity Drivers - Maintainability

- Monolithic architectures typically have low levels of maintainability due to
  - Technical partitioning of functionality into layers
  - Tight coupling between components
  - Weak component cohesion from a domain perspective
- Adding a simple change could require coordination between multiple teams
  - Triangle represents a where a change occurs



Modularity Drivers - Maintainability

- A distributed service based architecture allows the change scope at a domain level within a particular domain service
- In a service-based architecture, the change is at the domain level

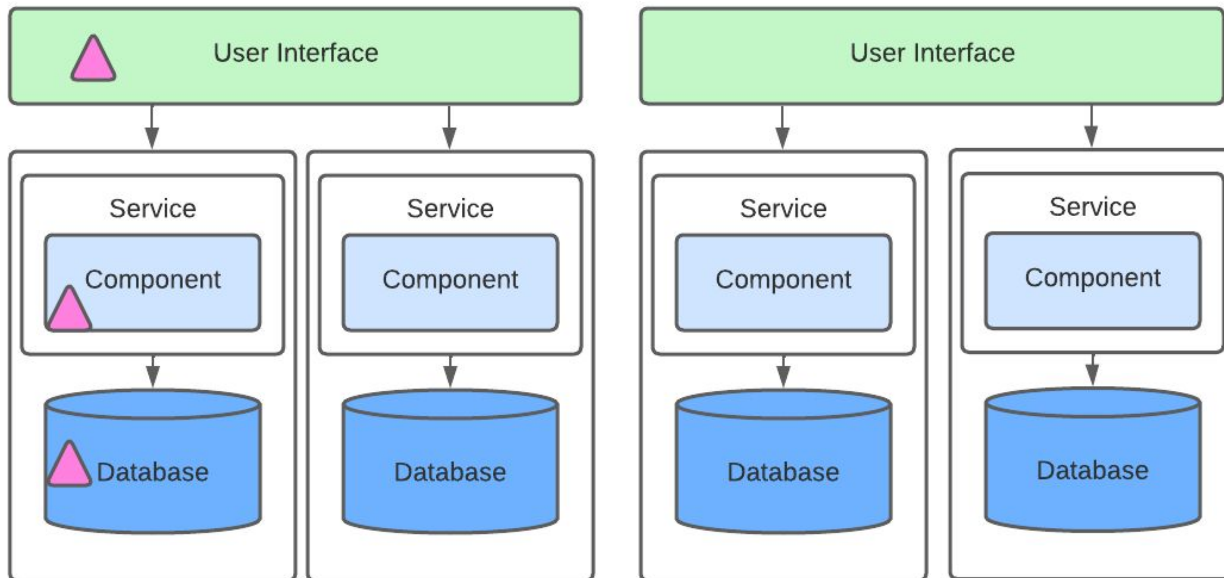


Change

Red Hat

Modularity Drivers - Maintainability

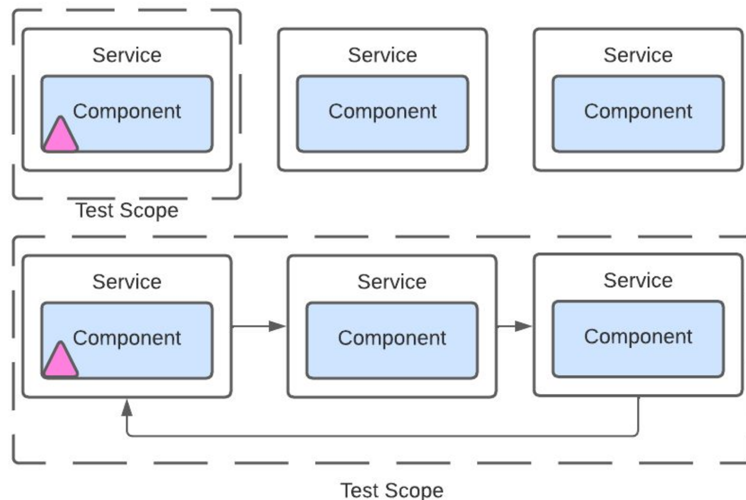
- Change is at a functional level in a microservices architecture demonstrating that as modularity increases, so does maintainability



Red Hat

Modularity Drivers - Testability

- Testability is a defined as ease of testing
- Large monolithic architecture styles like a layered architecture, support low levels of testability
  - Difficulty of regression testing features within the deployable unit
  - Size of the suite of full regression tests and time associated with running them
- Architectural modularity reduces the overall testing scope for changes
  - Allows for better completeness of testing
  - Easier to maintain the unit and automated tests
- A trade-off is that the testing scope can increase as services communicate with one another



Red Hat

Modularity Drivers - Deployability

- Deployability is about the frequency of deployment and the overall risk of deployment
- Less frequent deployment increases the risk of deployment (grouping multiple changes together)
- Monolithic architectures support low levels of deployability
  - Due to factors such as code freezes, mock deployments
  - Increased risk something might break due to lack of comprehensive regression testing when new features or bug fixes are deployed
- Modular architectures reduces risk due to
  - Less deployment ceremony
  - Better testing
- Deployability can be negatively impacted as services become smaller and communicate more with each other

Matt Stine on orchestrating microservices

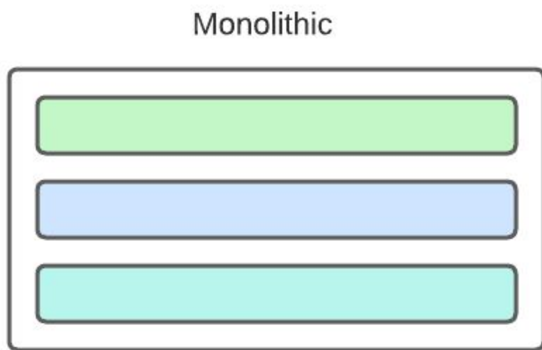
Red Hat

Modularity Drivers - Scalability

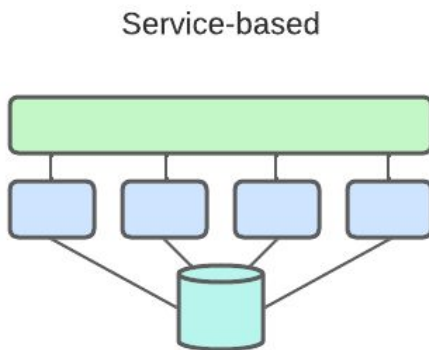
- Scalability is the ability of a system to remain responsive as user load increases over time
  - Elasticity is the ability of a system to remain responsive during significantly high instantaneous and erratic spikes in user load

- Both are characteristics of responsive systems as a function of concurrent requests - but they are handled differently
  - Scalability occurs over a longer period of time as a function of normal growth
  - Elasticity is the immediate response to a spike in user load

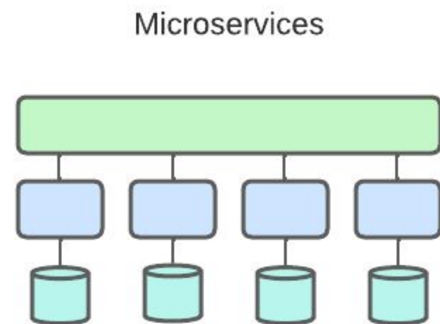
Modularity Drivers - Scalability



Scalability
Elasticity
Application-level scalability



Scalability
Elasticity
Domain-level scalability



Scalability
Elasticity
Function-level scalability

Red Hat

Modularity Drivers - Availability

- Availability and fault tolerance are the ability for some parts of a system to remain responsive and available as other parts of the system fail
- Monolithic systems offer low levels of fault tolerance
  - This could be somewhat mitigated by multiple instances and load balancing but this is expensive and relatively ineffective
- Modularity is key to domain-level and function-level fault tolerance
  - Catastrophic failure can be isolated to a deployment unit
  - A trade-off is that synchronous dependency on a service will not achieve fault tolerance
  - Asynchronous communication between services is key to a good level of fault tolerance

Red Hat

Trade-offs and Techniques

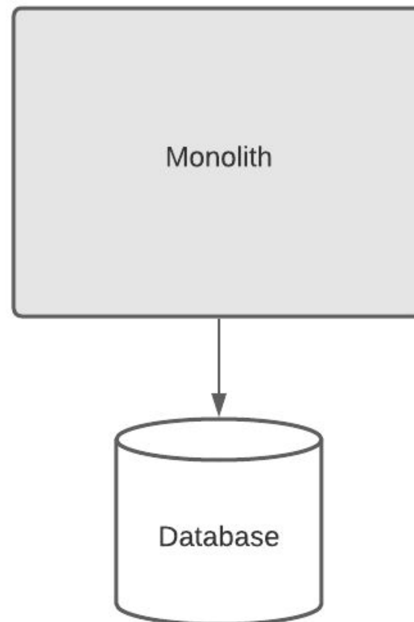
- Most modularity decisions lead to increase modularity
    - This is more complex to manage and deploy
- Consider Kubernetes and container deployments
- Use messaging for communications between services
- Leverage frameworks and libraries for reactive design such as Quarkus
- For elasticity consider serverless such as Knative versus scalability with native Kubernetes
- Look at your package structure (e.g. Java)

Red Hat

Red Hat

What are Monoliths

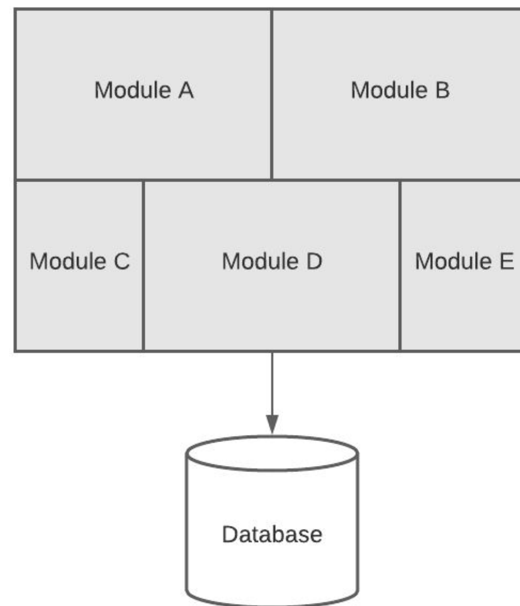
- When referring to monoliths, what are we really talking about?
  - A unit of deployment
  - Code packed into a single process
  - A multi-tier application
- When all the functionality of a system is deployed together and any change requires the system to be redeployed



Red Hat

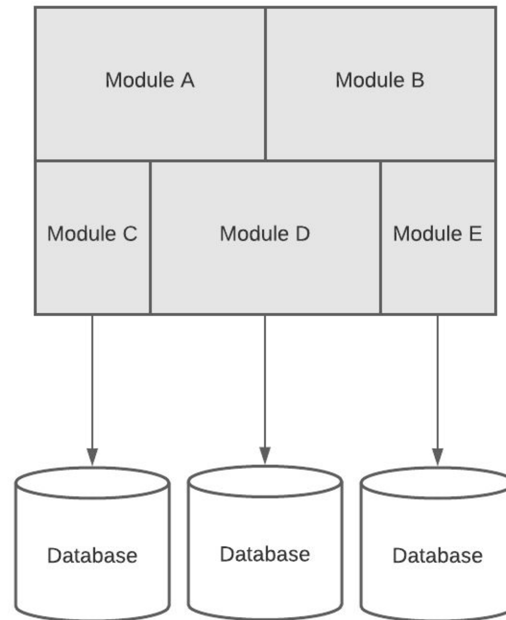
What are Modular Monoliths

- A variation on the monolith is the modular monolith
- The single system consists of separate modules
- Each module can be worked on independently
- The modular monolith is still a system of deployment as any module change requires the entire system to be redeployed
- When boundaries are well defined, the modular monolith is a good choice for organizations which want to avoid challenges of distributed microservice architectures
- Typically, the database is monolithic and changes will incur cost of modifying the entire system



What are Modular Monoliths with Decomposed Databases

- Typically consists of multiple services with decomposed databases
- Service oriented design
- A distributed monolith
- Low cohesion and high coupling lead to changes across service boundaries requiring the entire system to be redeployed for a change



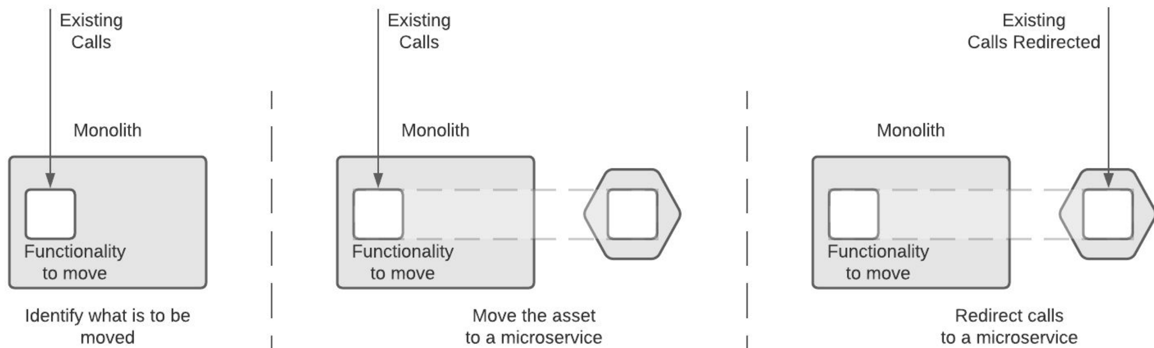
A structure is stable if cohesion is high, and coupling is low

-- Larry Constantine

Red Hat

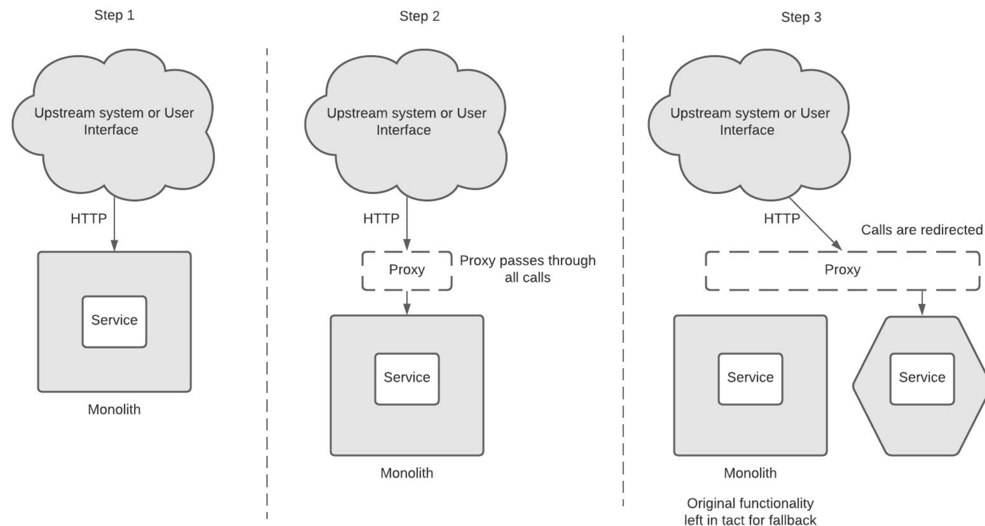
Pattern: Strangler Fig Pattern

- Based on Martin Fowler’s pattern inspired by a type of fig
- Sometimes used to migrate a monolith to a monolith
- Relies on three steps
  - Identify parts of the system to migrate
  - Implement the functionality in a new microservice
  - Reroute calls from the monolith



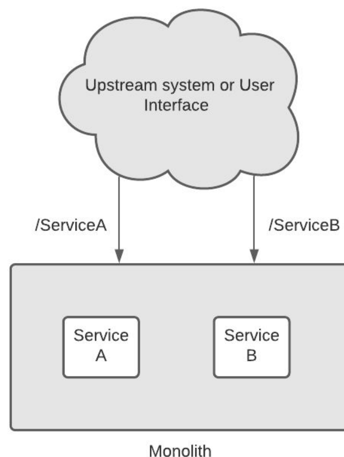
Example: Strangler Fig Pattern Proxy

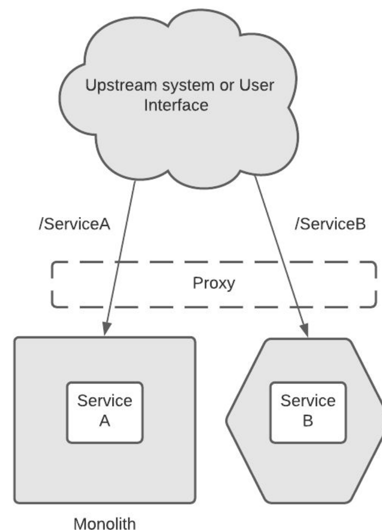
- Monoliths with HTTP interfaces allow for easier interception and redirection
- Typically three steps
    - Insert proxy
    - Migrate functionality
    - Redirect calls



Example: Strangler Fig Pattern Redirection

- Opt for a dedicated proxy
- Redirect if possible around URI paths
  - Can make use of REST resources
- Perform an incremental rollout
  - Redirect part of the behavior
- Break the work into stages
  - Deliver alongside other delivery work
  - Combine feature and technical stories to bring work together
- Deliver incremental changes

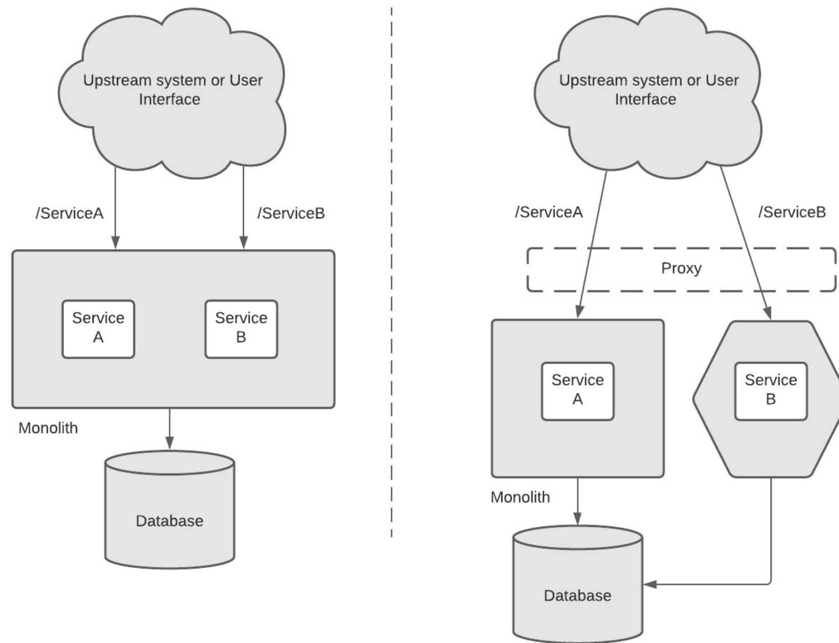




Red Hat

Example: Strangler Fig Pattern Shared Database

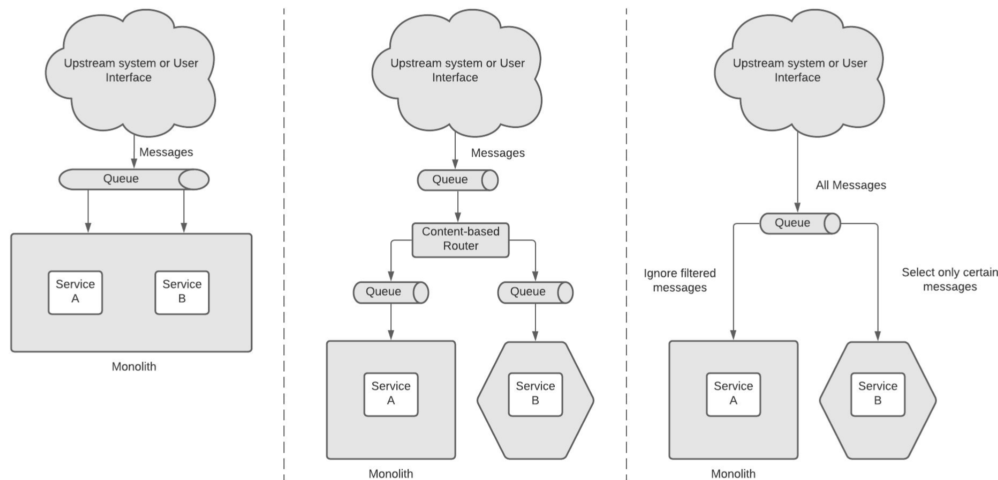
- A variation of the proxy pattern with a shared database for services
  - Decompose the bits for the database internally to the database
- Allows for incremental change to services being decomposed



Red Hat

Pattern: Content-based Routing

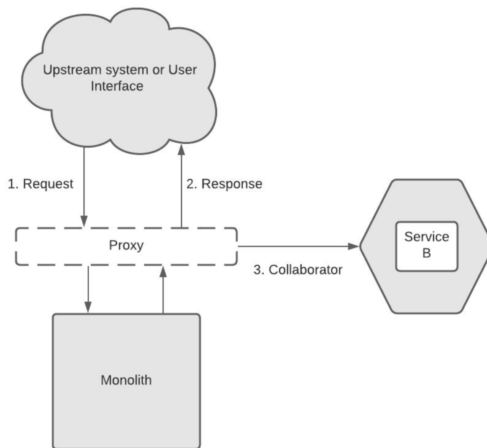
- A type of message interception
- Content-based routing is from Enterprise Integration Patterns
  - Intercept all messages and filter and route the messages to a new location
- Advantage of leaving the monolith untouched
- We could use filtering to reduce latency and selectively ignore messages also

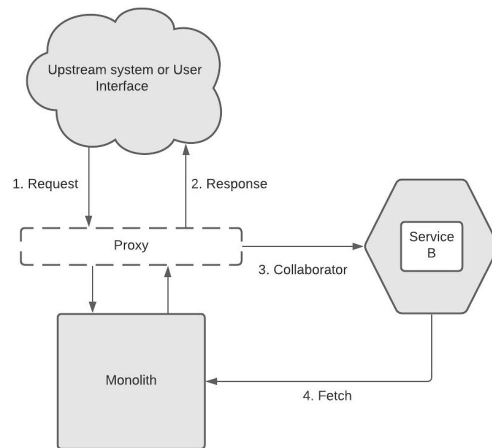


Red Hat

Pattern: Decorating Collaborator

- Use a proxy to determine if functionality is called in the monolith by using the decorator pattern with a proxy call to an external service
- Advantage of not changing the monolith
- Disadvantage of putting logic in the proxy
  - The new service may need additional information from the monolith





Red Hat

Trade-offs and Techniques

- With decomposition, migrating monolithic applications to distributed applications can be hard
- Look for seams within the application
- Make sure components have well-defined roles and responsibilities within a system as well as a well-defined set of operations
- You may have to redesign your namespaces and directory structures for you code to match the new component design
- When breaking a monolithic application into a distributed architecture, build services from components/modules and not individual classes
- A services based architecture may be a suitable stepping stone for migrating to microservices
  - Allows an architect to determine domains and granularity
  - Service-based architecture does not require the database to be decomposed immediately
  - Service-based architecture does not require operational automation or containerization immediately (you can still deploy an EAR)

Red Hat

Application Modernization:

Reuse

Code Reuse

- Code reuse has been a normal part of the software development process for many years
  - Common domain functionality such as formatters, calculators, validators, and auditing are a few examples of of code reuse
  - Common infrastructure functionality such as security, logging and metrics are common
- In a monolithic architecture, reuse can be as simple as including a package and sharing class files

- In a distributed architecture, what is the impact of a reused security component update across business domains or a common enterprise library shared across multiple organizations?

- A frequently used phrase such as “reuse is abuse” or “share nothing” are used by architects working on distributed systems

- In OO programming, DRY (Don’t repeat yourself) has been a guiding principle, but in modular distributed systems, WET (Write everything twice) has become the acronym of choice

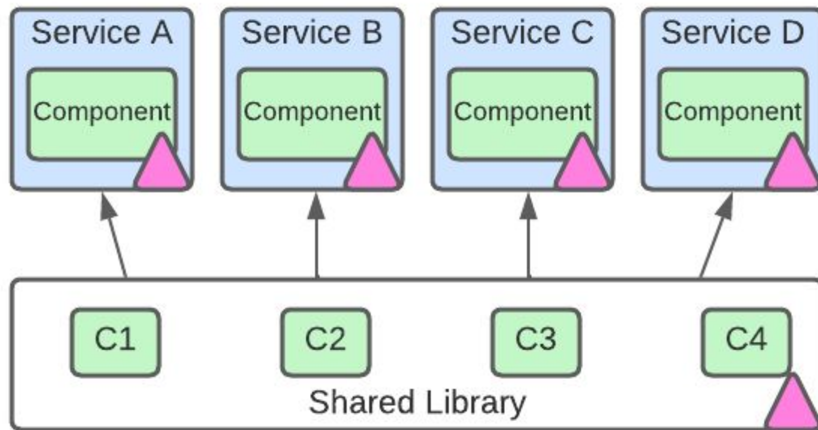
Code Replication

- Code replication is an example of older microservices techniques at concepts of “share nothing” and “bounded context”
- Code is “copied” into each service
- In practice, it falls apart quickly
    - Example, a bug now requires an update to all services containing the replicated code

Red Hat

Shared Libraries

- A more pervasive approach to reuse is sharing code through a library (JAR or DLL)
- Though this seems straightforward, it offers complexities in granularity and versioning
- Two of the trade-offs with using a shared library are dependency management and change control

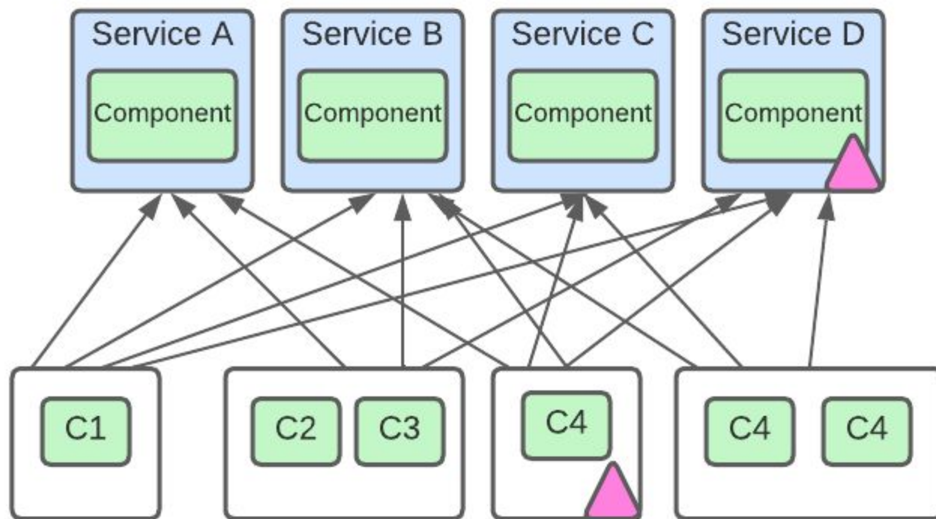


Change

Red Hat

Shared Libraries

- Breaking shared code into smaller functionality-based shared libraries (e.g. security, formatters, annotations, calculators etc...) is better for change control and maintainability but quickly turns into a dependency matrix mess

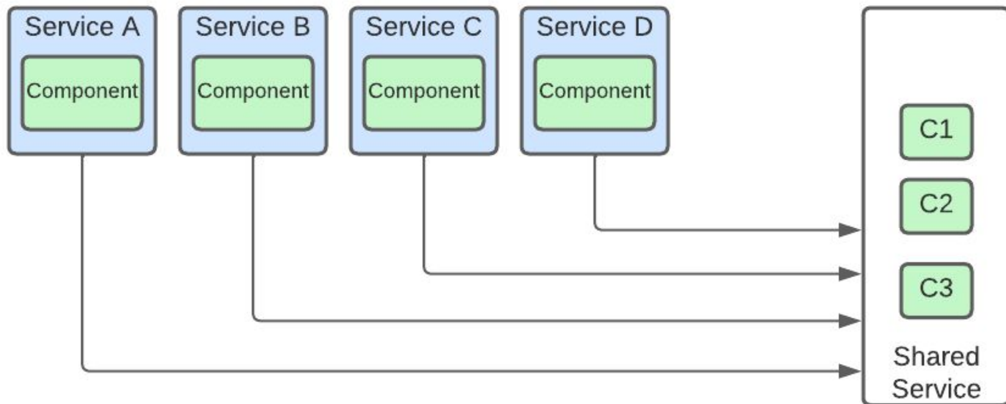


Change

Red Hat

Shared Service

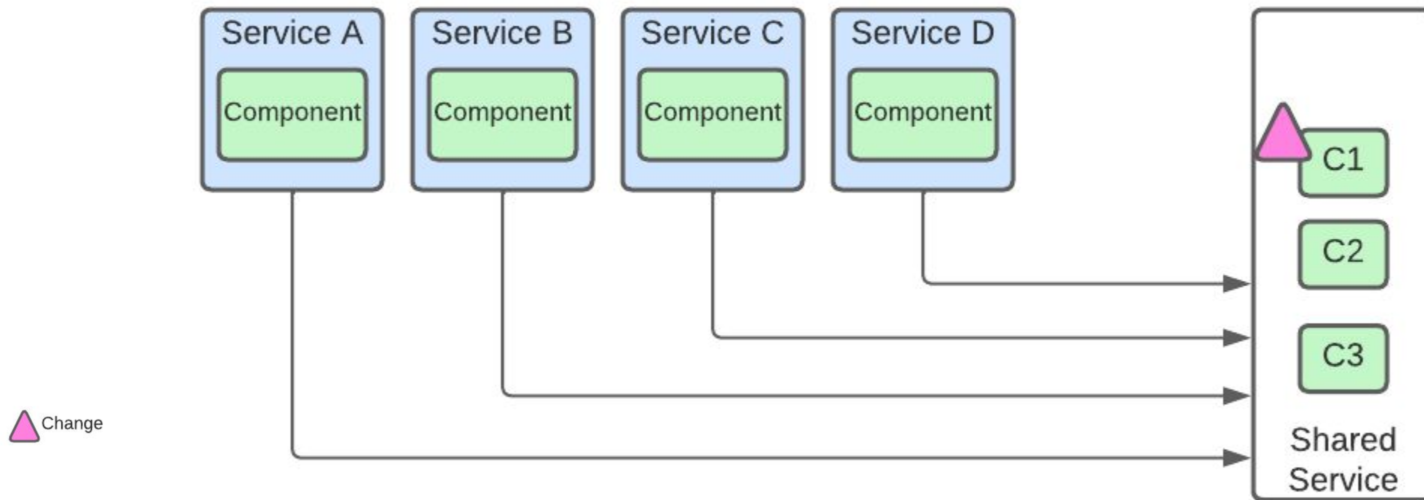
- A shared service allows for placing the shared functionality in a separately deployable service
- The feature of a shared service is that the code must follow a composition over inheritance strategy for code-reuse
- Changes to shared functionality no longer require redeployment of services
- Since changes are isolated to a separate service, they can be deployed without redeploying other services



Red Hat

Shared Service

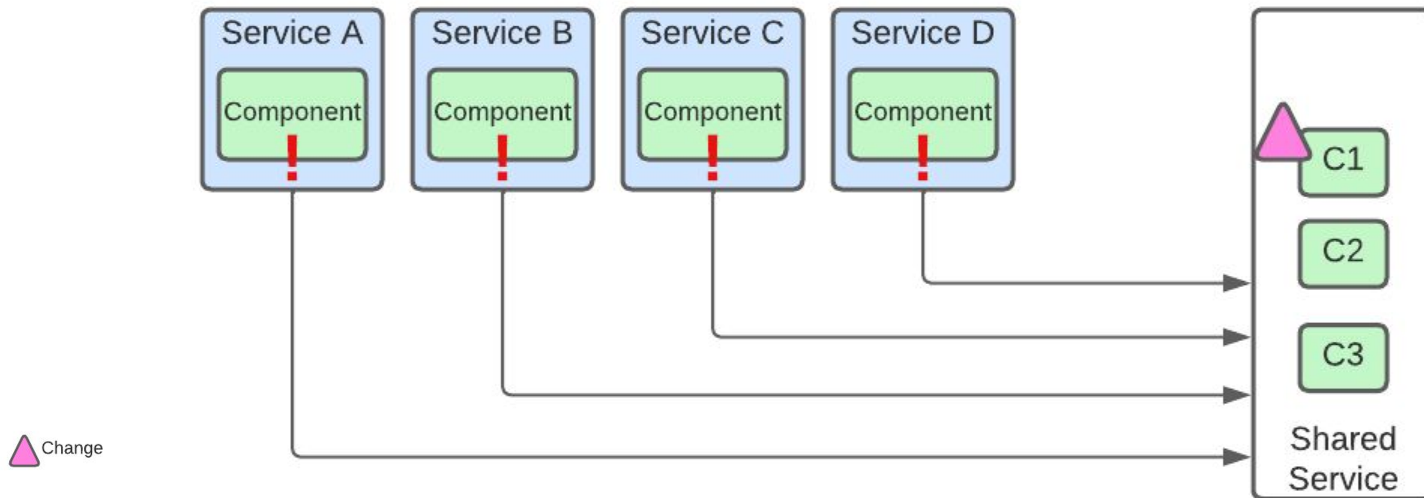
- But there is a risk with shared services
- The shared services can be changed and redeployed without retesting and redeploying all the other services



Red Hat

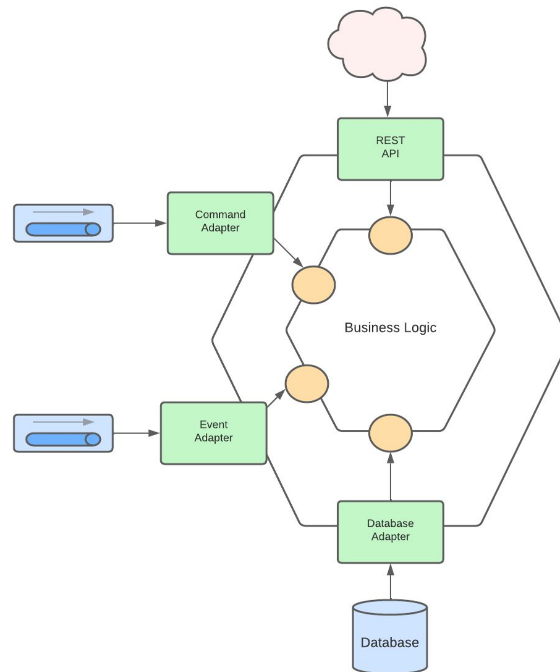
Shared Service

- So where is the risk?
- A shared service is a runtime change and a single change could bring down an entire system
- Shared services introduce scalability, fault tolerance, and performance issues as the system grows



Sidecar and Service Mesh

- A common solution in a distributed microservices architecture is to use a sidecar pattern
- This pattern is based on an earlier architecture pattern called hexagonal architecture (also known as Ports and Adaptors Pattern)
- Unlike DDD (Domain Driven Design), a database is just another adaptor

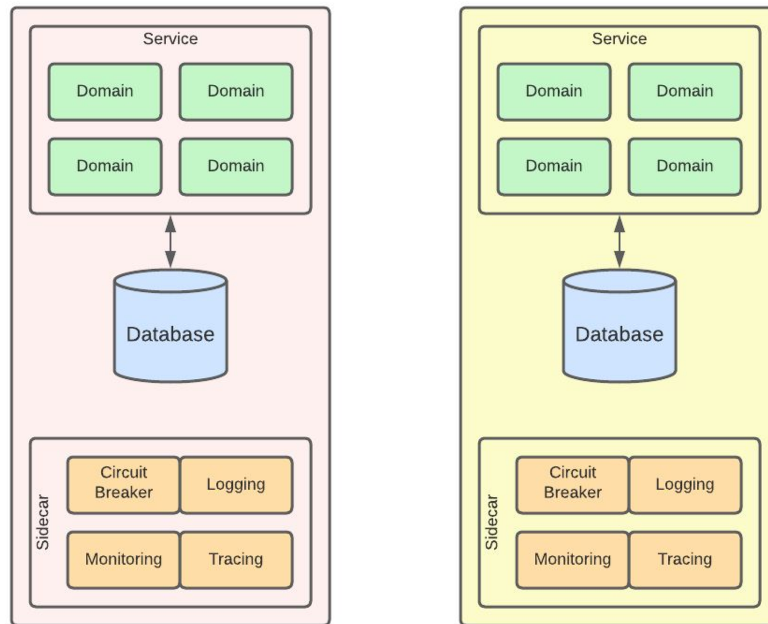


Hexagonal Architecture

Red Hat

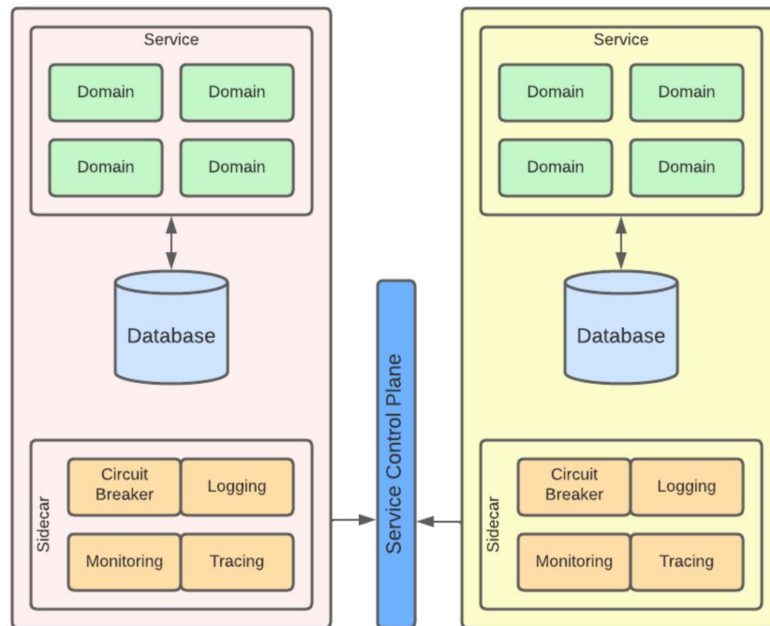
Sidecar and Service Mesh

- Conceptually, a sidecar is the same as hexagonal architecture in that it decouples the domain logic from the infrastructure logic
- Services can now split between operational concerns and domain concerns
- Any desire for operational consistency can be separated into the sidecar component



Sidecar and Service Mesh

- If we add operational consistency across services we can interconnect each service into a "mesh"
- This allows for DevOps to create operational dashboards, control operational characteristics such as scale and host other capabilities
- We can now add unified governance and support cross-cutting concerns



Red Hat

Sidecar and Service Mesh

- A sidecar represents a way to decouple operational capabilities from domains
- A sidecar is an orthogonal reuse pattern
- An orthogonal reuse pattern presents a way to reuse some aspect counter to one or more seams in the architecture
- Sidecars allow architects to isolate concerns in a cross-cutting and consistent layer through the architecture
- Sidecars and service mesh allow us to spread cross-cutting concern across a distributed architecture and are equivalent to the GoF Decorator Pattern

Red Hat

Trade-offs and Techniques

- Consider a service mesh for externalizing common functionality
- Use open standards such as OpenTelemetry which are compatible with a service mesh e.g. Tracing and logging
  - Microprofile in Java
- Copy paste is a good practice
- Sidecar offers
  - A consistent way to create isolated coupling
  - Allows consistent infrastructure coordination
  - Ownership per team
  - But
    - Implementing a sidecar may create a sidecar per platform
    - May grow large and complex

Application Modernization:

Distributed Data

Red Hat

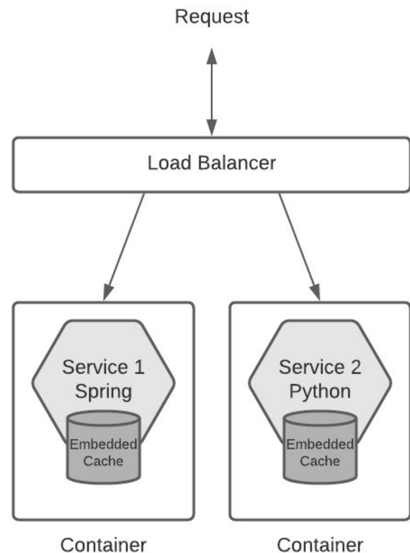
Caching

- Many developers and architects think of caching as a way to increase overall responsiveness
    - Storing data within an in-memory cache, retrieving data time can be reduced from milliseconds to nanoseconds
- Caching is an effective tool for distributed data access and sharing
- Leveraging replicated in-memory caching, data that is needed by other services is made available to each service without having them ask for it
- A replicated cache has data held in-memory within each service and is continuously synchronized so that all services have access to the exact same data

Red Hat

In-memory Caching

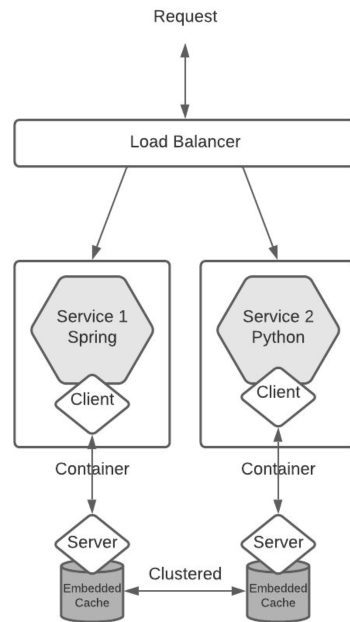
- A single in-memory caching model is simple
- It does not support data synchronized between caches meaning each service has its own unique data specific to that service
- This model does have limited impact on responsiveness and scalability within each service but it is not useful for sharing data between multiple services because of lack of cache synchronization between services



Red Hat

Distributed Caching

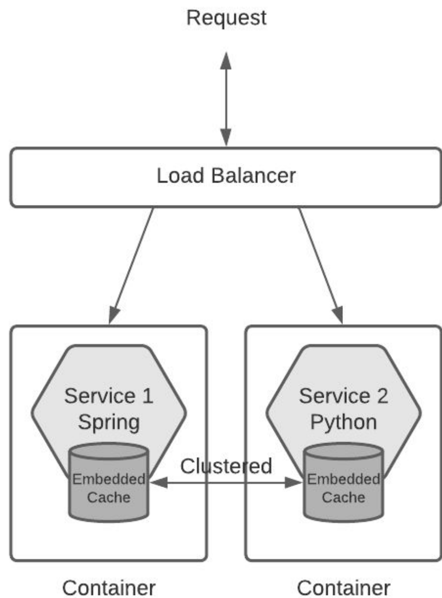
- A distributed cache used in a distributed architecture allows for a caching model where data is not held in-memory but with a caching server
- Services use a protocol to make requests to the caching server to retrieve or update shared data
- Data can now be shared among services



Red Hat

Replicated Caching

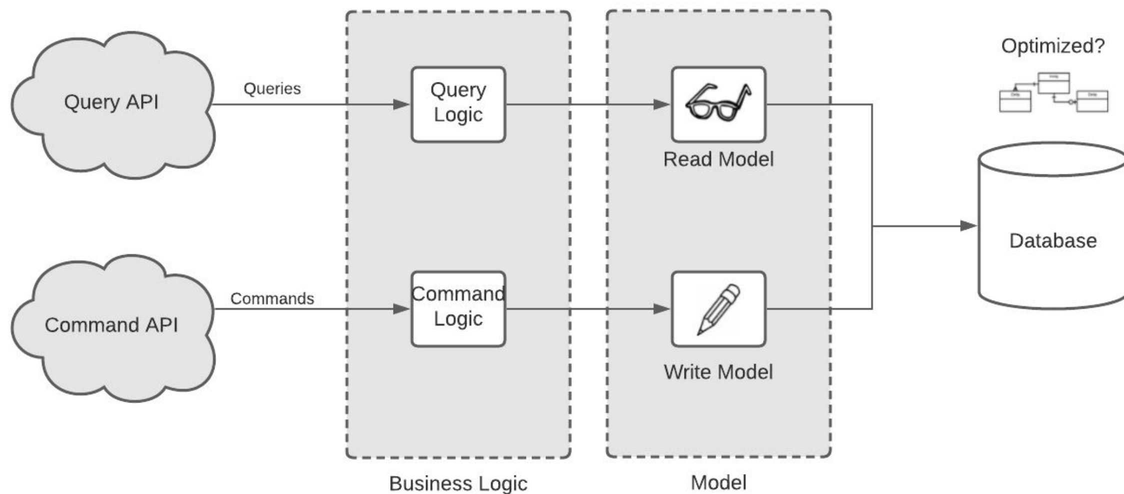
- Using replicated caching, each service has its own in-memory data that is kept in sync between services
- There is no external cache dependency
- Each cache instance communicates with another so that when an update is made to a cache, that update is immediately asynchronously propagated to other services using the same cache
- If the data becomes two large, a mechanism can be used so not all data is propagated to every cache and when a service is called, if the data is not there it can query the cache which will find it in the replicated cache



Red Hat

Pattern: Shared Data

- When the domain model is not complex
- Backing an applications services with the same daa store doesn't affect performance or introduce undesired complexity

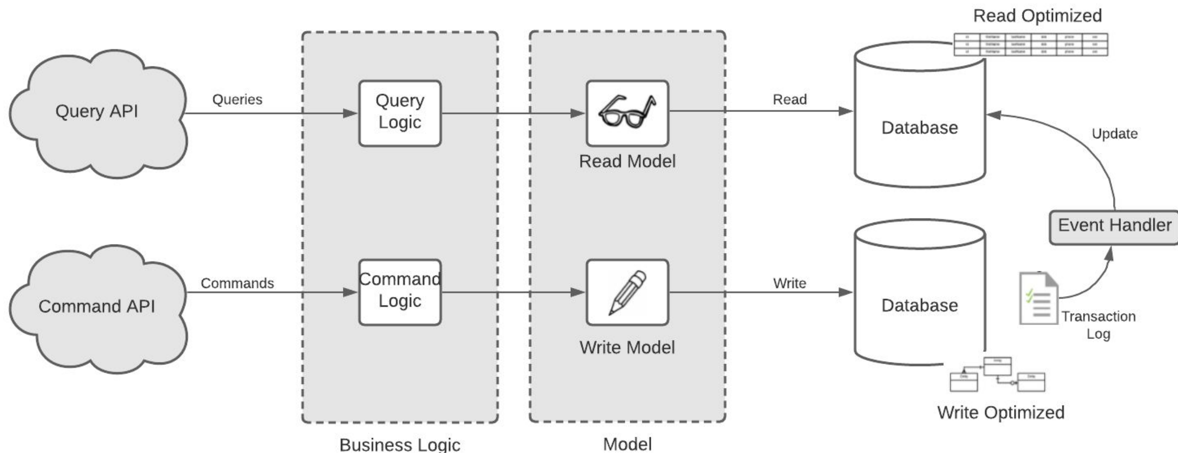


Red Hat

Pattern: Transaction Log Tailing

Tail the database transaction log and publish each event to a message broker

- **Usage:** When a change occurs in your application's data store, a notification must be published so that other applications can act upon it.



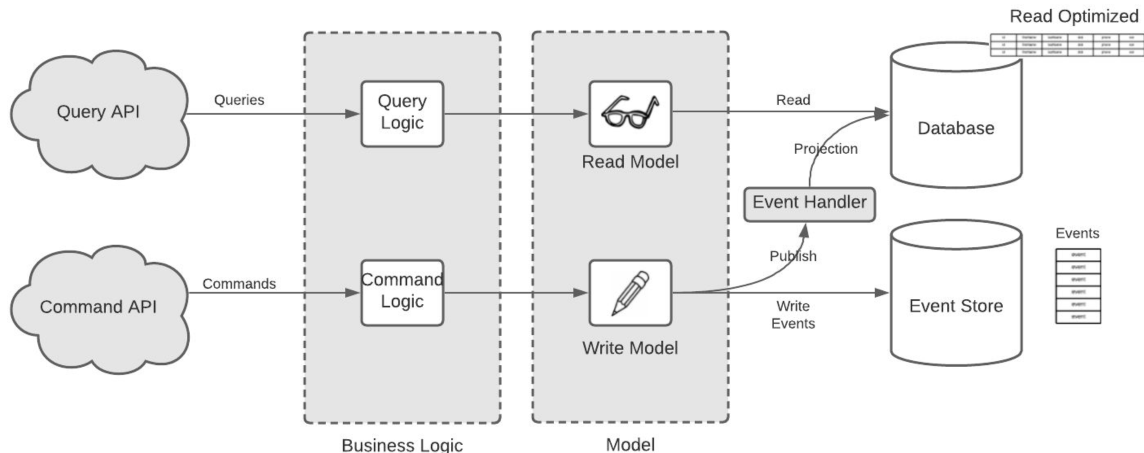
Transaction log is monitored to trigger updates | guaranteed accuracy

Red Hat

Pattern: Event Sourcing

All changes are stored as a sequence of events.

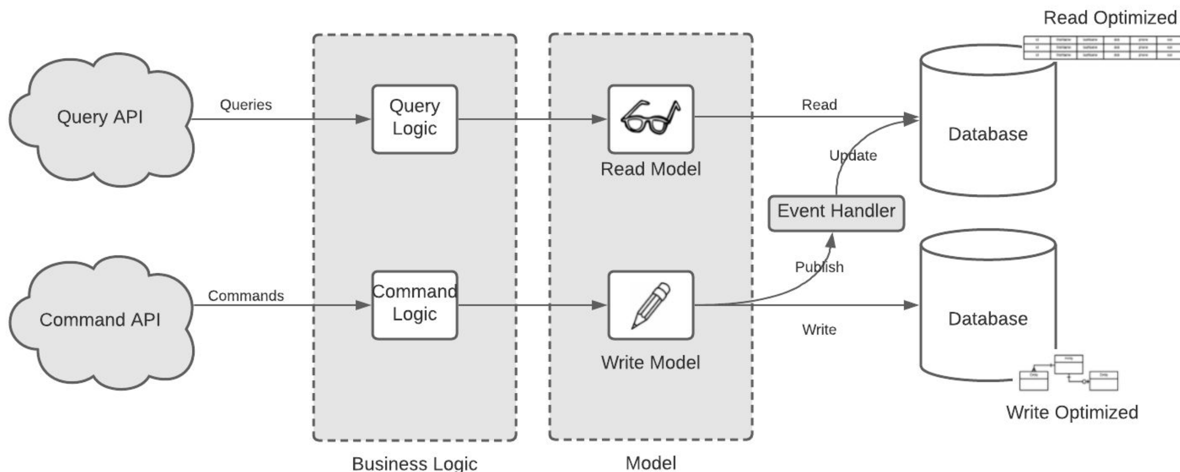
- **Usage:** You need to capture activity in your application - in the order that it occurs - so that the state of an object can be recreated or a series of actions can be replayed.



Writes stored as “log” of events | event handlers manage data propagation | object state can be re-constructed by replaying a series of events | eventual consistency

Pattern: Command Query Responsibility Segregation

- Separate data stores
- Event handlers manage data propagation
- Eventual consistency



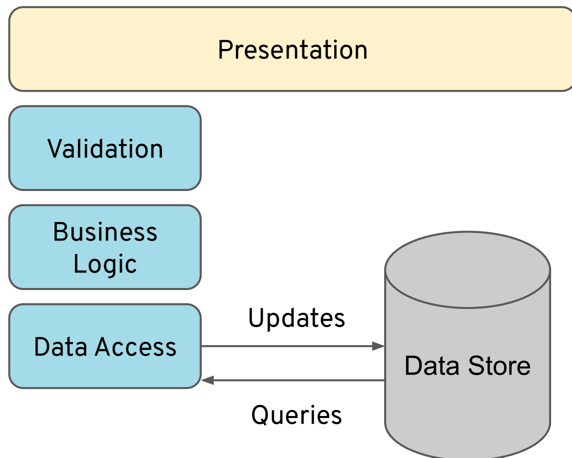
Red Hat

Pattern: Command Query Responsibility Segregation

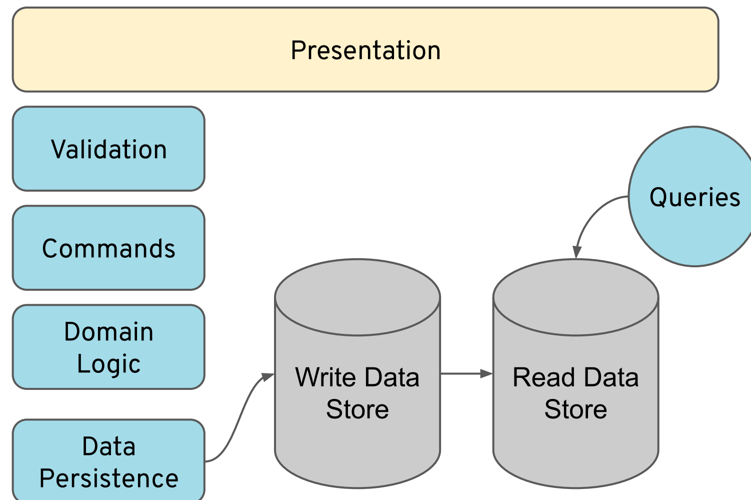
Implementing a query that retrieves data while avoiding chatty communication is challenging in a microservices architecture.

1. API Gateway for simple data aggregation
2. Command Query Response Segregation (CQRS) is less complex and more efficient than SQL join when multiple databases exist. With CQRS a separate table is created in a database for queries only.

Create, Read, Update Delete (CRUD) Monolith



CRUD following a CQRS strategy for Microservices



https://www.redhat.com/files/summit/session-assets/2018/Microservices-data-patterns-CQRS-and-event-sourcing-Distribution.pdf
https://developers.redhat.com/blog/2019/05/23/data-as-a-microservice-distributed-data-focused-integration/
https://docs.google.com/presentation/d/1ZGb_O0pdBC4WsEjYgAMucKTO4IRdjeynU3-zhM7Jnjo/edit?usp=sharing

Red Hat

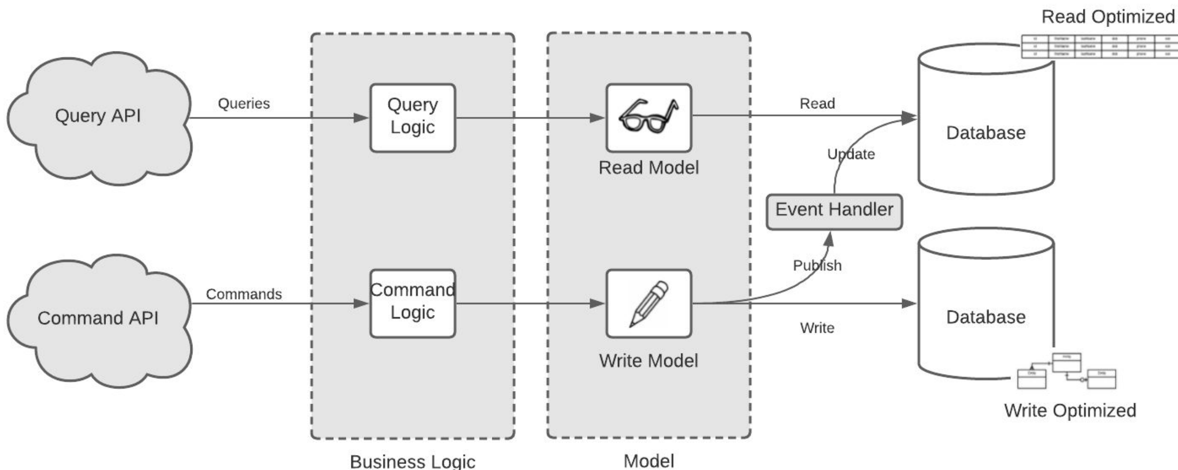
CQRS: Trade-offs

Benefits

- Loose coupling
- Independent scaling
- Optimized schemas
- Evolvable
- Query simplification
- Focus on user intent

Considerations

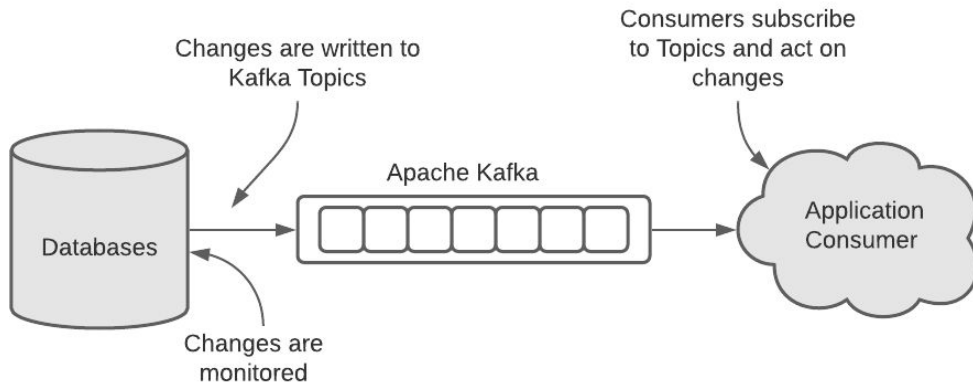
- More moving parts
- Distributed transactions
- Distributed data management, eventual consistency



Red Hat

Pattern: Simple Change Data Capture (CDC)

- Replicate data to databases
- Feed data into a data warehouse
- Update or invalidate a cache
- Feed data into an analytics system
- Feed data into a search engine
- Feed data into a monitoring system
- Propagate data to CQRS read models
- Propagate data within microservices architectures
- Aid with strangler pattern (mono-to-micro migration)

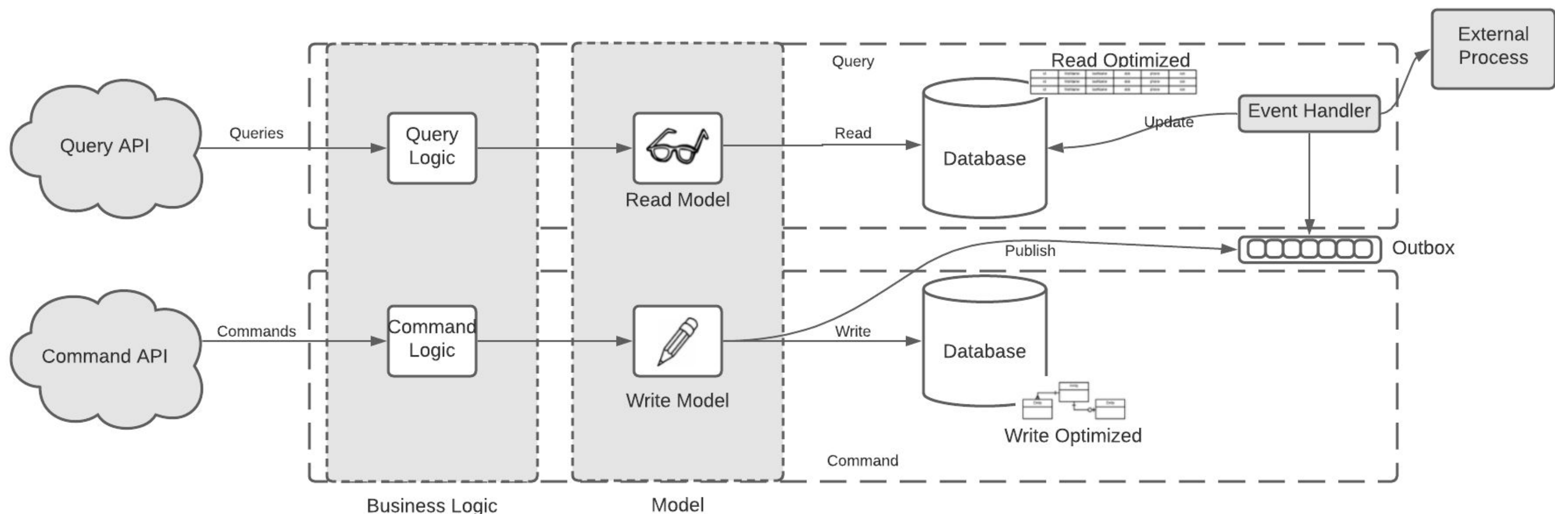


Change Data Capture (CDC) is a design pattern where changes to data in a database are detected and collected so that they can be acted upon by other applications.

Pattern: Outbox

A change event is published to an outbox.

- **Usage:** When a change event occurs in your application, a notification must be sent so that additional actions can be taken.



event handlers manage data propagation | events can also trigger other processes

https://debezium.io/blog/2020/02/10/event-sourcing-vs-cdc/

Trade-offs and Techniques

- What choices are you likely to make for caching and why?
- Consider a flexible caching tool such as JBoss Data Grid
- Caching can re-introduce coupling
- Avoid using a single database for integration
- How can change data capture positively impact your architecture
  - Consider using Debezium
  - May introduce some complexity such as involving Kafka but consider the trade-offs
- CQRS can be very complex
  - What are the benefits for your architecture at scale

Application Modernization:

Sagas

Red Hat

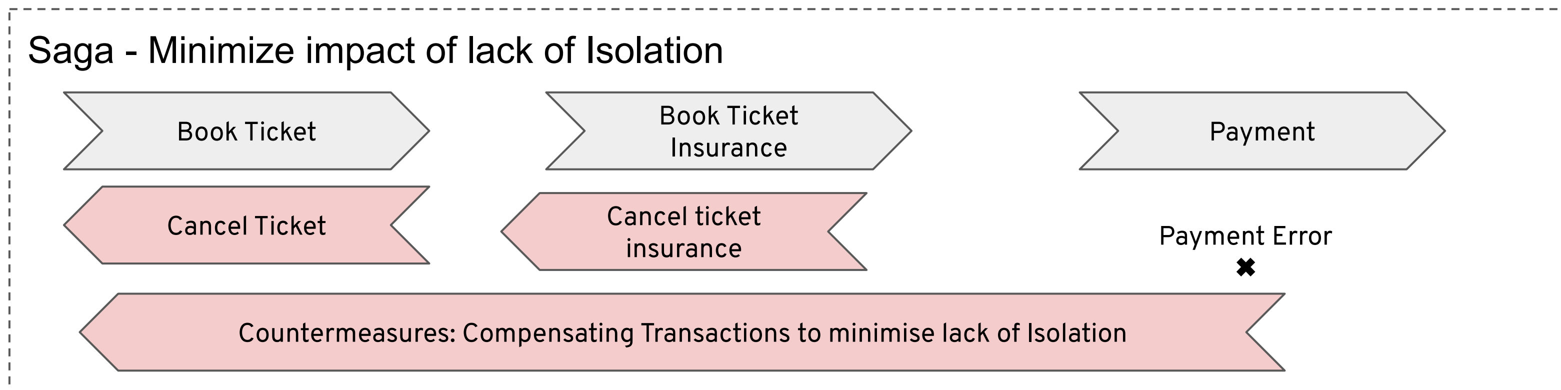
Distributed Transactions

Distributed Transactions typical in monolith transactions have **strong consistency and are ACID:**

- Atomicity - All or nothing transactions
- Consistency - Guarantees Committed Transaction State
- Isolation - Transactions are Independent
- Durability Commited Data is never lost

Microservices distributed databases have an **eventually consistency and lack Isolation (ACD)**

- Dirty Reads: Data being modified at read time
- Lost updates: Blind updates by two different transactions
- Non-repeatable reads: Inconsistent data return

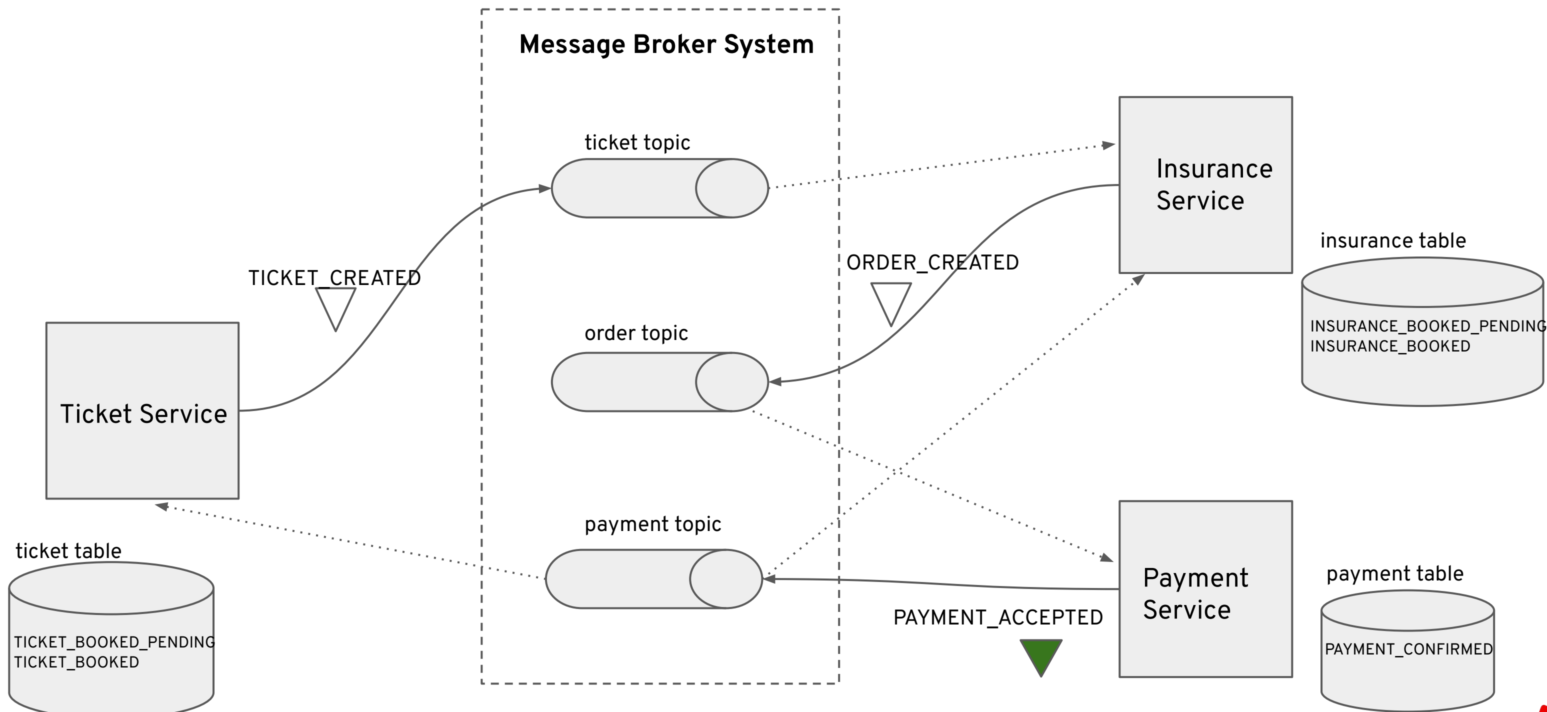


Two strategies for Saga, Choreography and Orchestration.

https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/
https://www.redhat.com/files/summit/session-assets/2019/T42224.pdf
https://docs.google.com/presentation/d/19Q6MaEeFJIlJzTr1Rc_cUehbDGeZzQVEZ70bmZT8KwY/edit?usp=sharing

Pattern: Saga | Choreographed

Choreography: The microservice is responsible for providing the logic to compensate

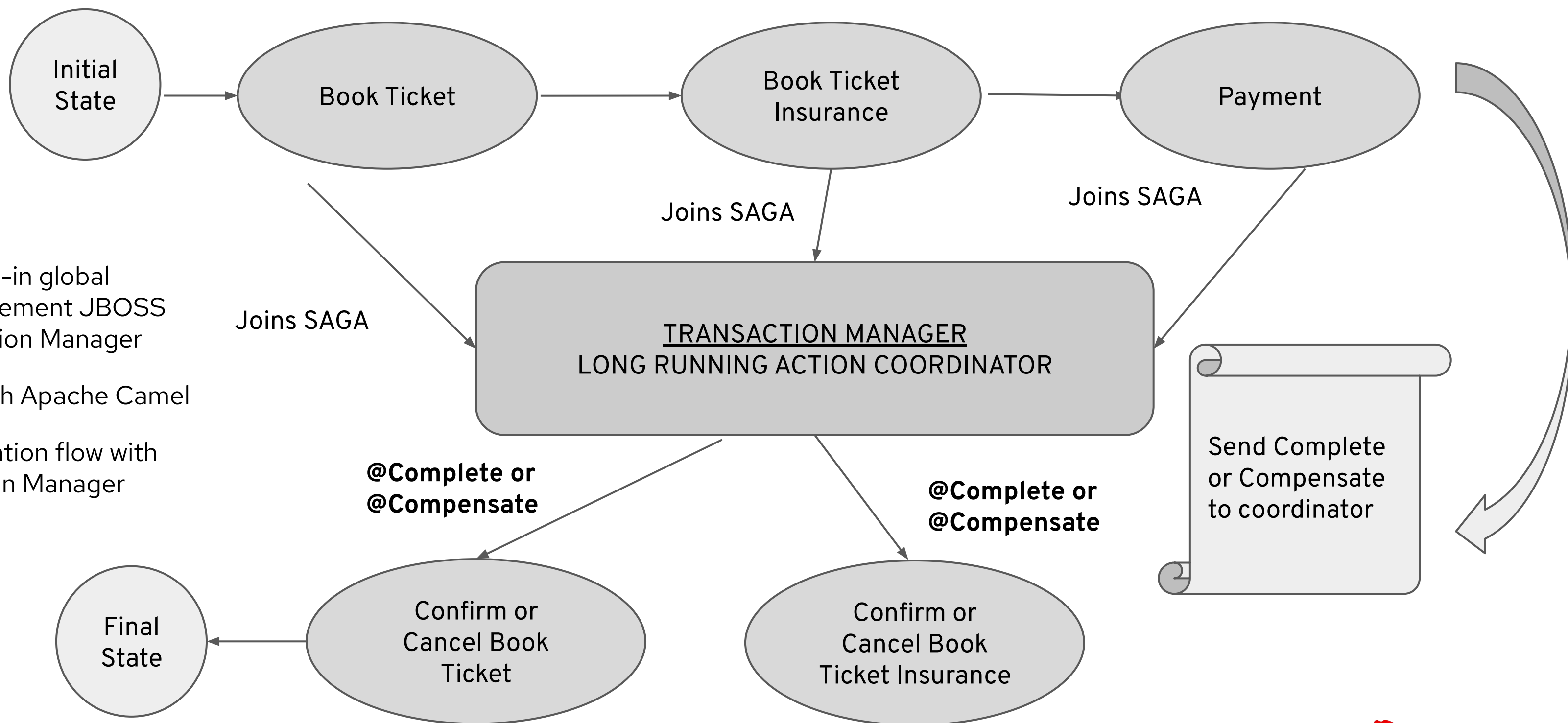


https://docs.google.com/presentation/d/19Q6MaEeFJlIJzTr1Rc_cUehbDGeZzQVEZ70bmZT8KwY/edit?usp=sharing

Red Hat

Pattern: Saga | Orchestrated

There is a central coordinator



1. Red Hat Fuse, built-in global transaction management JBOSS Narayana Transaction Manager
2. FUSE Saga EIP with Apache Camel
3. BPMN2 Compensation flow with Process Automation Manager

Red Hat

Sagas

- The idea of the saga pattern predates microservices, but there are really at least eight possible types of sagas

Pattern Name	Communication	Consistency	Coordination
Epic Saga (sao)	Synchronous	Atomic	Orchestrated
Phone Tag Saga (sac)	Synchronous	Atomic	Choreographed
Fairy Tale Saga (seo)	Synchronous	Eventual	Orchestrated
Time Travel Saga (sec)	Synchronous	Eventual	Choreographed
Fantasy Fiction Saga (aao)	Asynchronous	Atomic	Orchestrated
Horror Story (aac)	Asynchronous	Atomic	Choreographed
Parallel Saga (aeo)	Asynchronous	Eventual	Orchestrated

1. Software Architecture: The Hard Parts, Neal Ford, Mark Richards, Pramod Sadalage & Zhamak Dehghani

Red Hat

Trade-offs and Techniques

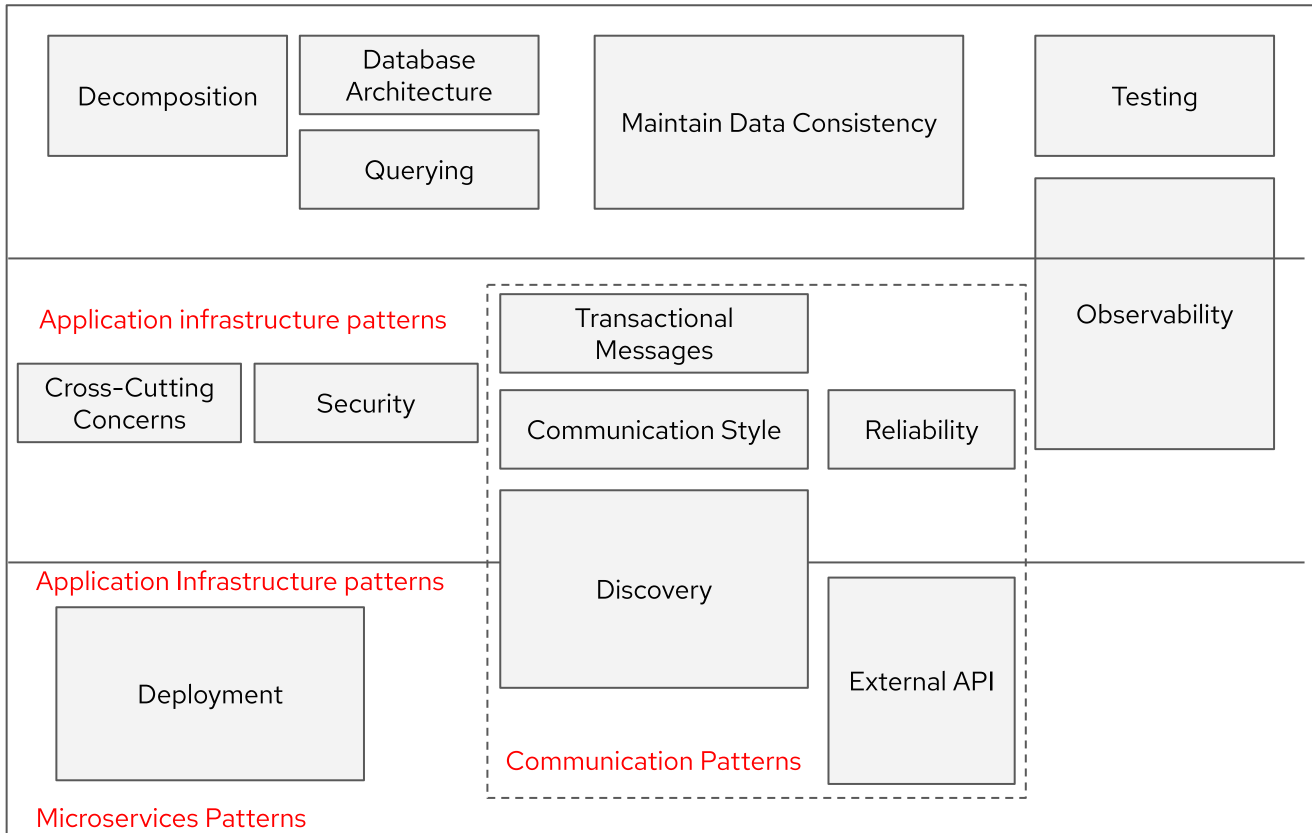
- Sagas are incredibly complicated
  - Perhaps try to avoid them until you need them
- Sagas will re-introduce coupling
- Not all saga types are equal
- Consider using an existing frame work such as
  - Netflix Conductor
  - Eventuate Tram

Red Hat

Conclusion

Red Hat

Microservices architecture



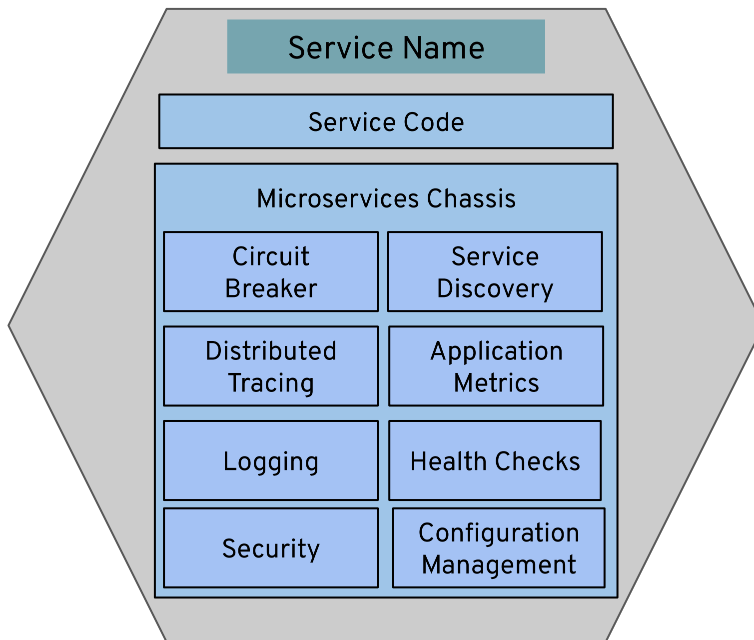
1. Microservices Patterns, Chris Richardson, Manning Publications

Red Hat

Loosely Coupled Architecture

Applications require a set of foundational capabilities to deal with cross-cutting concerns

Monolithic applications built these capabilities within the application



Microservices 1.0 leveraged language specific libraries to increase reuse (e.g. Netflix OSS)

Microservices 2.0 The platform provided those services with OpenShift (Kubernetes, ISTIO, Prometheus, EFK, OAuth, ...)

Complexity of an Existing Application to Migrate

	Easy	Moderate	Difficult
Code	Completely Isolated (single process)	Somewhat Isolated (multiple processes)	Self Modifying (e.g. Actor Model)
Configuration	One Configuration File	Several Configuration Files	Configuration Anywhere in Filesystem
Data	Data Saved in Single Place	Data Saved in Several Places	Data Anywhere in Filesystem
Secrets	Static Files	Network	Dynamic Generation of Certificates
Network	HTTP, HTTPS	TCP, UDP	IPSEC, Highly Isolated
Installation	Packages, Source	Installers (install.sh) and Understood Configuration	Installers (install.sh)
Licensing	Open Source	Proprietary	Restrictive & Proprietary
Type	Web based, stateless, Stateful (JBoss EAP), Storage with good data separation, Red Hat supported (PHP, Node, MySQL)	PCI Compliant, Big Data, Advanced routing, Non-HTTP/Websocket	Proprietary Software, ERP, CRM, Legacy (e.g. COBOL).

https://www.redhat.com/en/blog/architecting-containers-part-4-workload-characteristics-and-candidates-containerization

Red Hat

Free Download

Modernizing Enterprise Java

While containers, microservices, and distributed systems dominate discussions in the tech world, most applications in use today still run monolithic architectures that follow traditional development processes. This practical book helps developers examine long-established Java-based models and demonstrates how to bring these monolithic applications successfully into the cloud-native model with Kubernetes.

- Learn the basics of cloud-native applications and understand what parts of your organization's Java-based applications and platforms need to migrate and modernize.
- Understand how enterprise Java specifications can help you transition projects and teams.
- Build a cloud-native platform that supports effective development without falling into buzzword traps.
- Find a starting point for your migration projects by identifying candidates and staging them through modernization steps.
- Discover how to complement a traditional enterprise Java application with components on top of containers and Kubernetes.
- Go beyond and look toward the future of cloud-native, serverless, event-driven architectures.

https://developers.redhat.com/e-books/modernizing-enterprise-java

Free Download

OpenShift for Developers

Get a hands-on introduction to daily life as a developer crafting code on OpenShift, the open source container application platform from Red Hat. Creating and packaging your applications for deployment on modern distributed systems can be daunting. Too often, sophisticated infrastructure can complicate development. With this practical guide, you'll learn how to build, deploy, and manage a multitiered application on Red Hat OpenShift.

With the Kubernetes container orchestrator at its core, OpenShift simplifies and automates how you build, ship, and run code. You'll learn how to use OpenShift and the Quarkus Java framework to develop and deploy apps using proven enterprise technologies and practices that you can apply to code in any language.

- Learn the development techniques and tools for building and deploying on OpenShift.
- Use OpenShift to build, deploy, and manage the ongoing lifecycle of an n-tier application.
- Create a continuous integration and deployment pipeline to turn your source code changes into production rollouts.
- Automate scaling decisions with metrics and trigger application lifecycle events with webhooks.

https://developers.redhat.com/e-books/openshift-for-developers

Free Download

Knative Cookbook

Enterprise developers face several challenges when it comes to building serverless applications, such as integrating applications and building container images from source. With more than 60 practical recipes, this cookbook helps you solve these issues with Knative–the first serverless platform natively designed for Kubernetes.

With this cookbook, you'll learn to:

- Efficiently build, deploy, and manage modern serverless workloads
- Apply Knative in real enterprise scenarios, including advanced eventing
- Monitor your Knative serverless applications effectively
- Integrate Knative with CI/CD principles, such as using pipelines for faster, more successful production deployments
- Deploy a rich ecosystem of enterprise integration patterns and connectors in Apache Camel K as Kubernetes and Knative components.

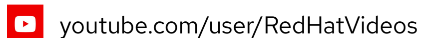
https://developers.redhat.com/books/knative-cookbook

Q & A

Red Hat

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.



Red Hat