

Welcome

Microservices Best Practices and Patterns

VENKATA RAMANA
jbossramana@gmail.com



Venkata Ramana

Over three decades of IT experience in Corporate Training, Software development and architectural designs involving web technologies, databases, SOA, Microservices & Cloud.

A Prayer of gratitude to Mother Nature:

All actions are being performed by the Mother Nature.

But in my ignorance, I may sometimes think that :

"I am the doer"

No.. Never

"Mother Nature is the doer"

A Prayer of gratitude to Mother Nature:

I have a right to perform my prescribed duties, but
I am not entitled to the fruits of my actions.

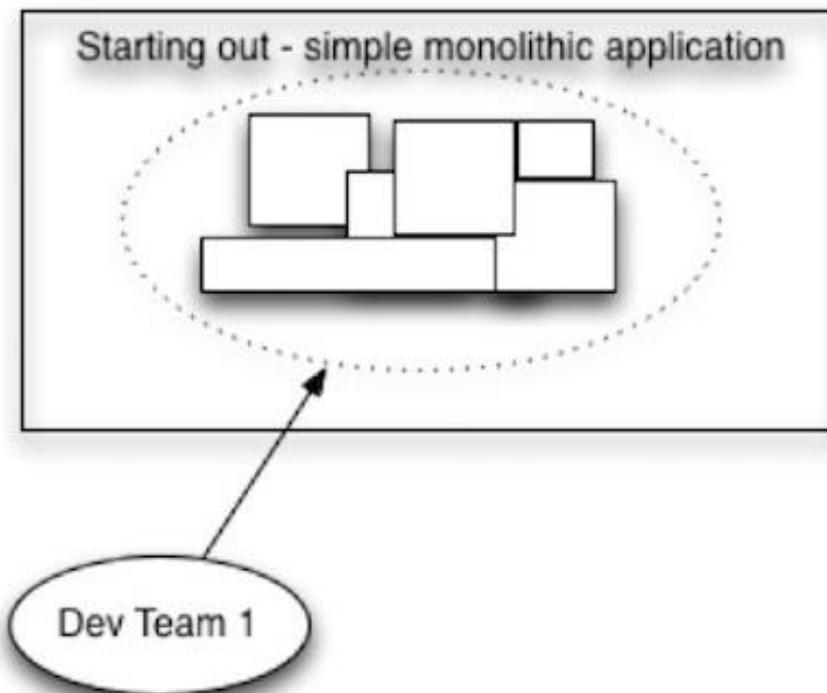
I never consider myself to be the cause of the
results of my activities, nor be attached to inaction.

Monolith Application

Monolith Application:

A single monolith would contain all the code for all the business activities an application performed.

We write a simple application, which is developed and managed by a single team.



Advantages of monolithic applications over microservices:

- 1. Simplicity:** Monolithic applications are generally simpler to develop, deploy, and manage compared to microservices architectures. With a monolithic approach, you have all components of the application tightly integrated into a single codebase, making it easier to understand and maintain.

- 2. Development speed:** Because all components of the application are in one codebase, development teams often find it faster to iterate and develop new features in a monolithic architecture. There's no need to manage communication between different services or deal with the complexities of distributed systems.

3. Easier debugging and testing: Debugging and testing a monolithic application can be simpler compared to microservices because all components run within the same process, making it easier to trace issues and perform end-to-end testing.

4. Performance: In some cases, monolithic applications can have better performance compared to microservices architectures, especially when communication overhead between services is a concern. Since all components are within the same process, method calls and data access can be more efficient.

5. Resource efficiency: Monolithic applications may require fewer resources (such as memory and processing power) compared to microservices architectures, as there is no overhead associated with running multiple instances of different services.

6. Simplified deployment: Deploying a monolithic application typically involves deploying a single artifact, which can be simpler compared to managing and orchestrating multiple services in a microservices architecture.

7. **Easier scalability:** Scaling a monolithic application can be simpler compared to microservices architectures, especially in cases where the entire application needs to be scaled uniformly. With microservices, scaling individual services requires more sophisticated orchestration and coordination.
8. **Team familiarity:** Many developers are familiar with monolithic architectures, as they have been the traditional approach for many years. This familiarity can lead to quicker onboarding for new team members and a larger pool of available talent.

Here are a few specific examples of use cases where a monolithic architecture might be suitable:

1. Content Management Systems (CMS):

- ✓ Many small to medium-sized websites or blogs utilize monolithic CMS platforms like WordPress or Drupal. These platforms provide a unified solution for content creation, management, and publishing without the need for complex distributed architectures.

2. E-commerce Websites:

- ✓ Small e-commerce businesses with relatively simple product catalogs and transactional workflows may opt for monolithic architectures. They can use platforms like Magento or Shopify, which offer integrated solutions for inventory management, order processing, and customer relationship management.

3. Internal Business Applications:

- ✓ Companies often build monolithic applications to handle internal processes such as employee management, payroll, inventory tracking, and project management. These applications typically have a well-defined scope and serve specific business needs without the need for extensive integration with external services.

4. Enterprise Resource Planning (ERP) Systems:

- ✓ Small or medium-sized enterprises may deploy monolithic ERP systems to streamline various business functions such as accounting, human resources, supply chain management, and customer relationship management. These systems provide integrated solutions for managing core business processes within a single application.

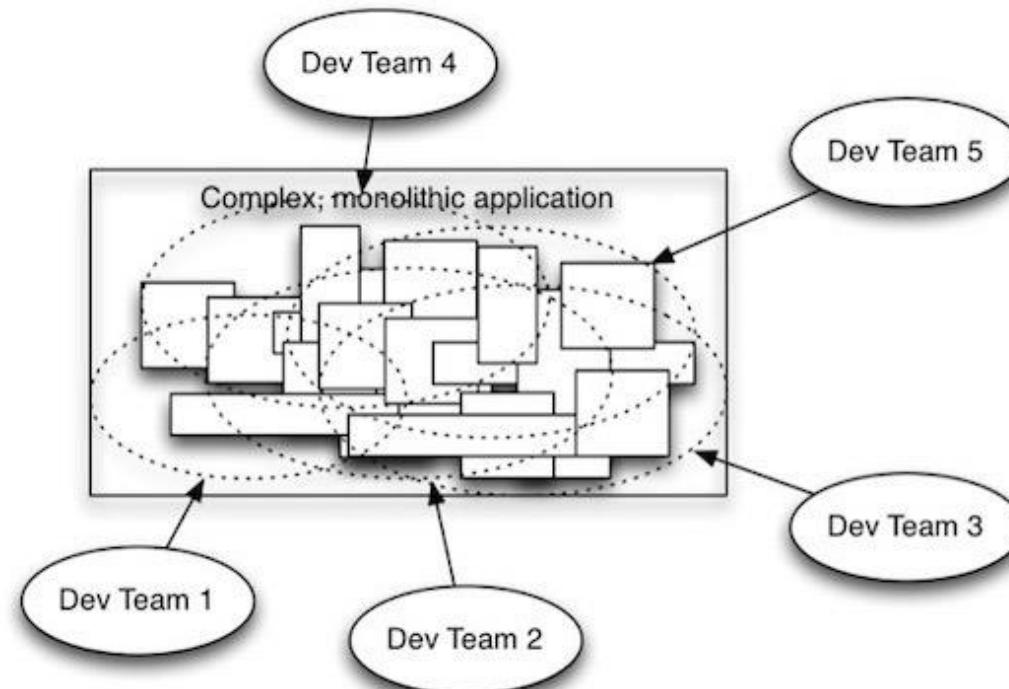
5. Learning Management Systems (LMS):

- ✓ Educational institutions or corporate training departments may use monolithic LMS platforms to deliver online courses, track student progress, and manage course content. These platforms provide a centralized solution for organizing educational materials, conducting assessments, and facilitating communication between instructors and learners.

Microservices Architecture

But what happens if your application turns out to be successful?
Users like it and begin to depend on it.

Traffic increases dramatically. And almost inevitably, users request improvements and additional features, so more developers are roped in to work on the growing application. Before too long, your application looks more like this:



Here are some scenarios when choosing microservices might be appropriate:

1. Large, Complex Applications: When dealing with large and complex applications, where different components have distinct functionalities and can be developed, deployed, and scaled independently, microservices can offer better modularity and maintainability.

2. Scalability Requirements: If your application needs to handle variable and potentially high loads, microservices allow you to scale individual components independently. This scalability is particularly advantageous for applications with varying usage patterns across different parts of the system.

3. Technology Diversity: Microservices allow you to choose the most suitable technology stack for each service based on its requirements. This flexibility is beneficial when different components have diverse needs in terms of programming languages, frameworks, databases, or other technologies.

4. Continuous Deployment and DevOps Practices: Microservices align well with continuous integration and continuous deployment (CI/CD) practices and DevOps culture. They enable teams to release updates to individual services more frequently, reducing the risk and impact of changes compared to monolithic applications.

5. Fault Isolation and Resilience: Microservices promote fault isolation, meaning that failures in one service are less likely to impact the entire system. This architectural pattern enhances system resilience and fault tolerance, as failures are contained within individual services.

6. Integration with Third-party Services: If your application needs to integrate with multiple third-party services or APIs, microservices can provide a more modular and flexible approach. Each service can be responsible for interacting with specific external systems, simplifying integration and reducing dependencies.

- ✓ It's important to carefully evaluate your project's requirements, team capabilities, and organizational constraints before deciding to adopt a microservices architecture.
- ✓ While microservices offer numerous benefits, they also introduce additional complexity in terms of development, deployment, monitoring, and operations.
- ✓ Therefore, it's crucial to ensure that the benefits outweigh the challenges and that your team has the necessary expertise and resources to successfully implement and manage a microservices-based system.

Microservice architecture is a method of developing software applications as a set of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

Single Responsibility

Each software module or a class should have one and only one reason to change

Liskov Substitution

You should be able to use any derived class instead of a base class without modification

Dependency Inversion

High level classes should not depend on low level classes instead both should depend upon abstraction

S O L I D

Open/Closed

A Software Class or module should be open for extension but closed for modification

Design Principles

Interface Segregation

Client should not be forced to use an interface which is not relevant to it

A class or module should have one, and only one, reason to be changed (i.e. rewritten).

As an example, consider a module that compiles and prints a report.

Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change.

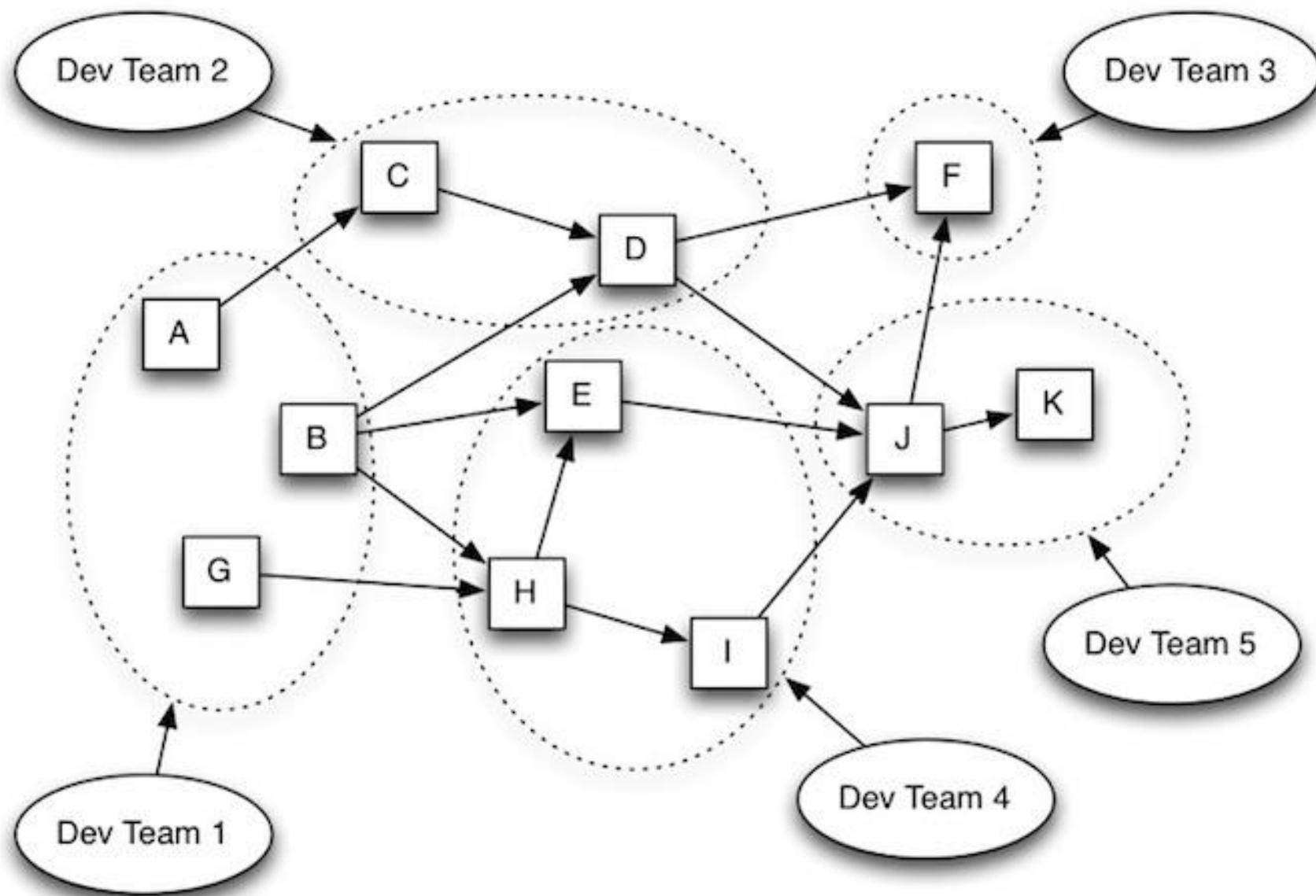
The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

Applications built using microservices possess certain characteristics.

In particular, they:

- ❑ Are fragmented into multiple modular, loosely coupled components, each of which performs a discrete function
- ❑ Have those individual functions built to align to business capabilities
- ❑ Can be distributed across clouds and data centres
- ❑ Treat each function as an independent service that can be changed, updated, or deleted without disrupting the rest of the application



Overview of Microservices Architecture:

Definition: Microservices architecture is an architectural style that structures an application as a collection of loosely coupled services.

Characteristics: Each service is responsible for a specific business function, can be developed and deployed independently, and communicates via lightweight protocols.

Contrasted with monolithic architecture: In contrast to monolithic architecture, where the entire application is built as a single, tightly-coupled unit, microservices architecture decomposes the application into smaller, independently deployable services.

Key Principles and Characteristics:

Decentralization: Services are developed, deployed, and managed independently, promoting autonomy and flexibility.

Scalability: Services can be scaled independently based on demand, allowing for efficient resource utilization.

Resilience: Fault isolation and redundancy enable the system to continue operating in the event of failures.

Polyglotism: Each service can be implemented using different programming languages, frameworks, or technologies based on its requirements.

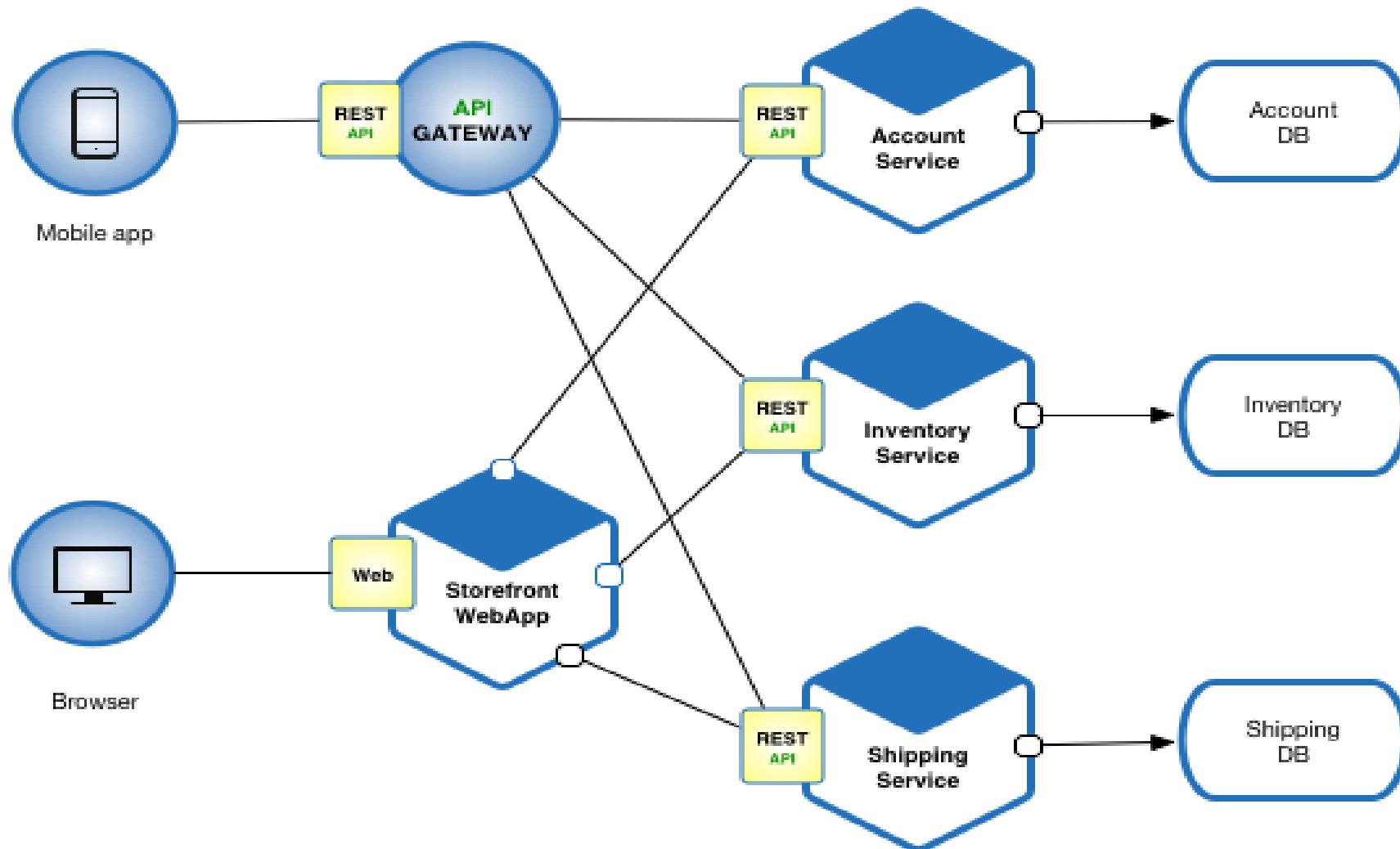
Microservices

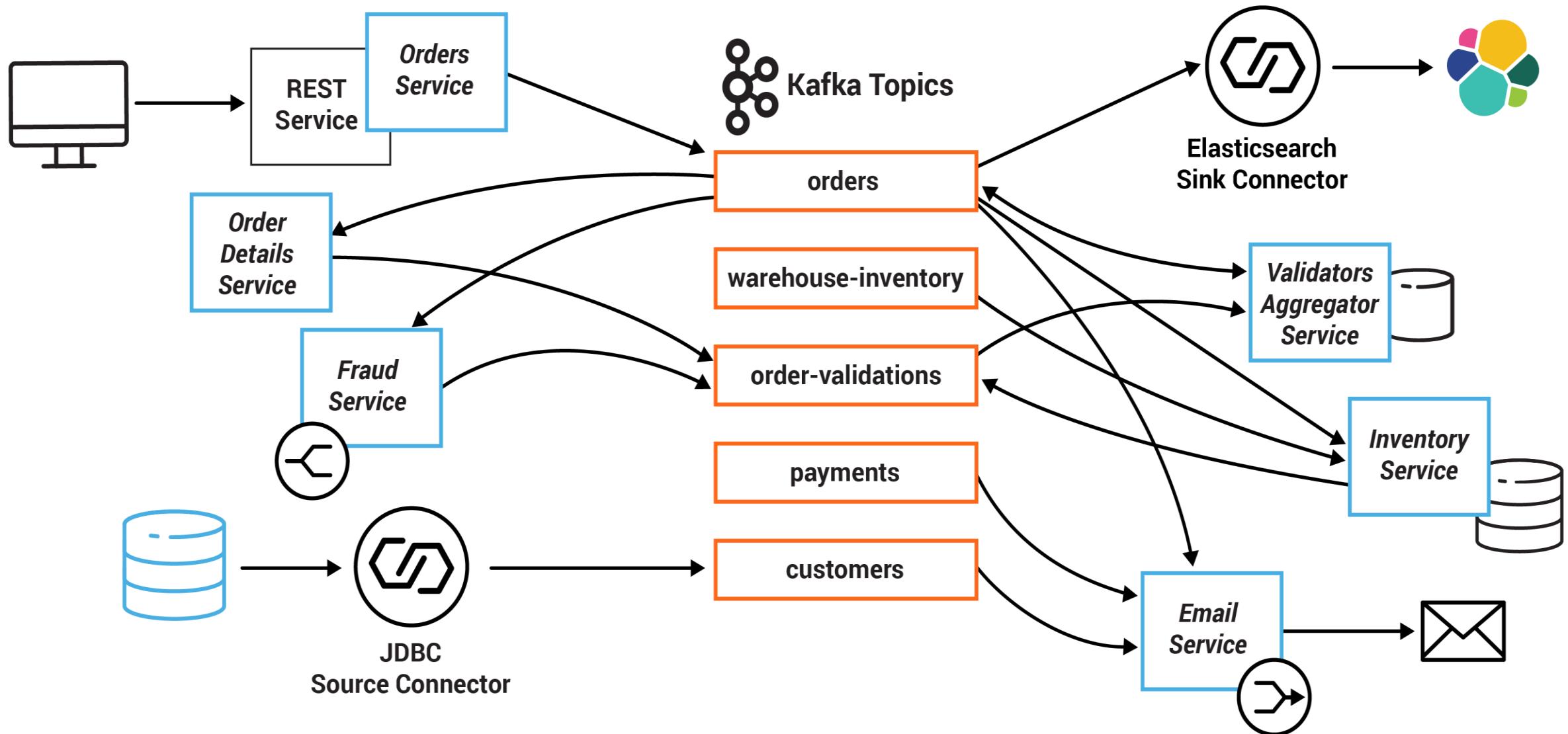
So what exactly is a microservice architecture?

According to Martin Fowler:

A microservice architecture consists of “set of independently deployable services” organized “around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.”

Microservice Architecture





Microservices are a service-oriented architectural pattern as well for defining distributed software architectures.

The pattern aims for better scalability, decoupling and control throughout the application development, testing and deployment cycle.

It relies on an inter-service communication protocol, which could be SOAP, REST and other technologies..

*The microservices style is usually organized around **business capabilities and priorities**.*

Unlike a traditional monolithic development approach—where different teams each have a specific focus on, say, UIs, databases, technology layers, or server-side logic—microservice architecture utilizes cross-functional teams.

The responsibilities of each team are to make specific products based on one or more individual services communicating via message bus. That means that when changes are required, there won't necessarily be any reason for the project, as a whole, to take more time or for developers to have to wait for budgetary approval before individual services can be improved.

Microservices Benefits

- Smaller code base is easy to maintain.
- Easy to scale as individual component.
- Technology diversity i.e. we can mix libraries, databases, frameworks etc.
- Fault isolation i.e. a process failure should not bring whole system down.
- Better support for smaller and parallel team.
- Independent deployment
- Deployment time reduce

Microservices Challenges

- Difficult to achieve strong consistency across services
- ACID transactions do not span multiple processes.
- Distributed System so hard to debug and trace the issues
- Greater need for end to end testing
(JUnit, Selenium ,Arquillian, Hoverfly, AssertJ)
- Required cultural changes in across teams like Dev and Ops working together even in same team.

cloud-native application



A cloud-native application is an application specifically designed and optimized to leverage cloud computing principles and technologies.

Cloud-native applications are developed, deployed, and operated with cloud principles in mind, taking advantage of the scalability, resilience, and flexibility offered by cloud environments.

Here are some key characteristics and principles of cloud-native applications:

Microservices Architecture:

Cloud-native applications are often built using a microservices architecture. Instead of monolithic architectures, where the entire application is built as a single, tightly-coupled unit, microservices architecture breaks the application into smaller, independent services.

Each service is focused on a specific business function and can be developed, deployed, and scaled independently.

Containerization:

Cloud-native applications are typically packaged and deployed using containerization technologies such as Docker.

Containers encapsulate the application along with its dependencies, making it portable and consistent across different environments.

Containerization enables developers to build once and run anywhere, whether it's on-premises or in the cloud.

Orchestration:

Container orchestration platforms like Kubernetes are commonly used in cloud-native environments to automate the deployment, scaling, and management of containerized applications.

Kubernetes provides features such as service discovery, load balancing, auto-scaling, and rolling updates, allowing for efficient management of distributed applications.

Infrastructure as Code (IaC):

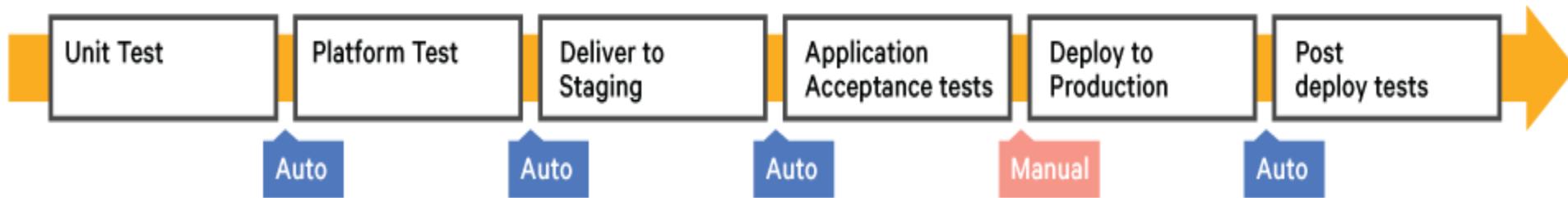
Cloud-native applications are often provisioned and managed using Infrastructure as Code (IaC) principles. Infrastructure components such as virtual machines, networks, and storage are defined and configured using code, allowing for automation, consistency, and repeatability in infrastructure provisioning and management.

DevOps Practices:

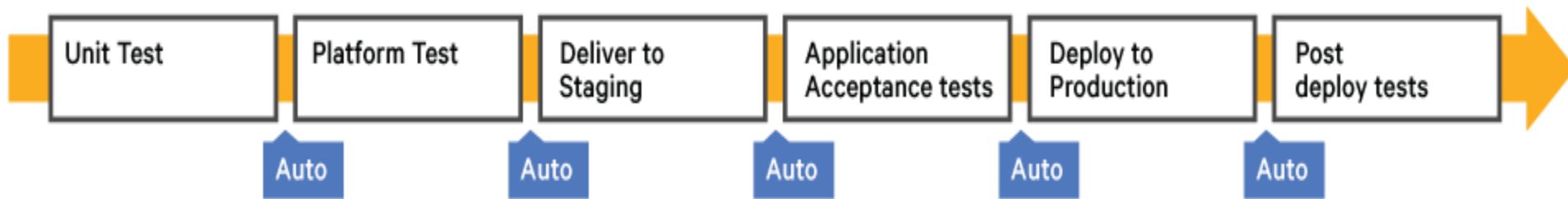
Cloud-native development embraces DevOps practices to streamline the software delivery lifecycle. Continuous integration, continuous delivery (CI/CD), automated testing, and monitoring are integral parts of cloud-native development workflows.

DevOps practices enable teams to deliver software faster, with higher quality, and increased agility.

Continuous Delivery



Continuous Deployment



Resilience and Fault Tolerance:

Cloud-native applications are designed to be resilient and fault-tolerant in the face of failures. They leverage techniques such as redundancy, auto-scaling, and distributed architecture to minimize downtime and ensure high availability.

Components within cloud-native applications are often decoupled and designed to fail independently without impacting the entire system.

Scalability and Elasticity:

Cloud-native applications are designed to scale horizontally to handle varying levels of traffic and workload. They can dynamically scale up or down based on demand, allowing for efficient resource utilization and cost optimization.

Auto-scaling capabilities provided by cloud platforms enable applications to automatically adjust capacity in response to changes in traffic patterns.

Cloud-Native Data Services:

Cloud-native applications leverage cloud-native data services such as managed databases, caching services, and data analytics platforms. These services provide scalable, highly available storage and processing capabilities that complement the scalability and flexibility of cloud-native applications.

Overall, cloud-native applications are designed to be agile, scalable, resilient, and cost-effective, leveraging cloud technologies to deliver value to users more efficiently and effectively than traditional approaches.

The **12 Factor App** methodology

“Cloud-native applications conform to a framework or “**contract**” designed to maximize resilience through predictable behaviours”

Cloud Native Platform

Developer



1. Application Framework

Contract – 12 Factor App

Dev+Ops

2. Container Runtime

Contract – BOSH Release

IT Ops

3. Infrastructure Automation



Contract – Cloud Provider Interface

IT Ops

4. Infrastructure



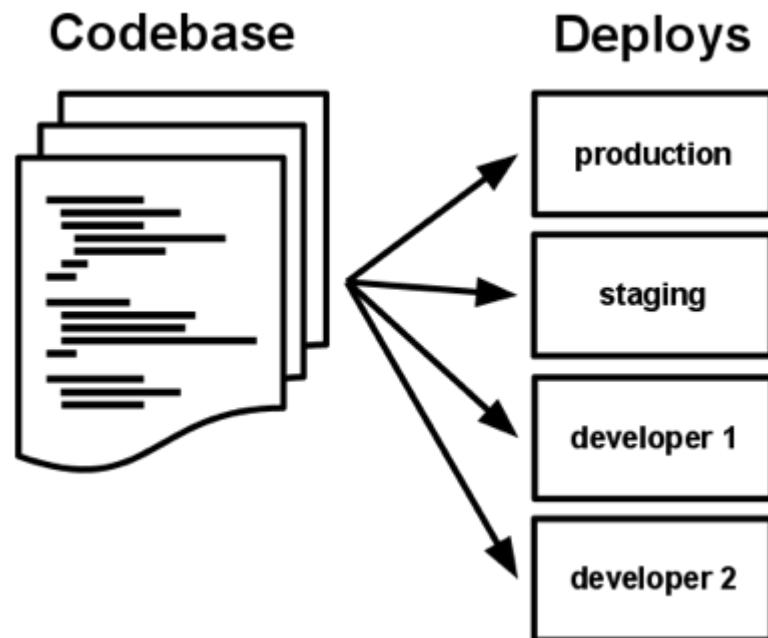
"Contracts" between Applications, opinionated frameworks like Spring Boot and Spring Cloud and opinionated Cloud Native Platforms like Cloud Foundry help significantly accelerate the development of Cloud Native applications

The twelve-factor app is a methodology for building software-as-a-service apps

Twelve-Factor Applications		
1. CODEBASE One codebase tracked in SCM, many deploy	2. DEPENDENCIES Explicitly declare isolate dependencies	3. CONFIGURATION Store config in the environment
4. BACKING SERVICES Treat backing services as attached resources	5. BUILD, RELEASE, RUN Strictly separate build and run stages	6. PROCESSES Execute app as stateless processes
7. PORT BINDING Export services via port binding	8. CONCURRENCY Scale out via the process model	9. DISPOSABILITY Maximize robustness & graceful shutdown
10. DEV/ PROD PARITY Keep dev, staging, prod as similar as possible	11. LOGS Treat logs as event stream	12. ADMIN PROCESSES Run admin / mgmt tasks as one-off processes

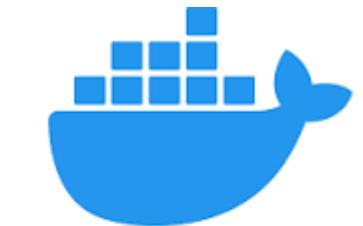
1. CODEBASE

One codebase tracked
in SCM, many deploy





pom.xml



Dockerfile

2. DEPENDENCIES
Explicitly declare
isolate dependencies

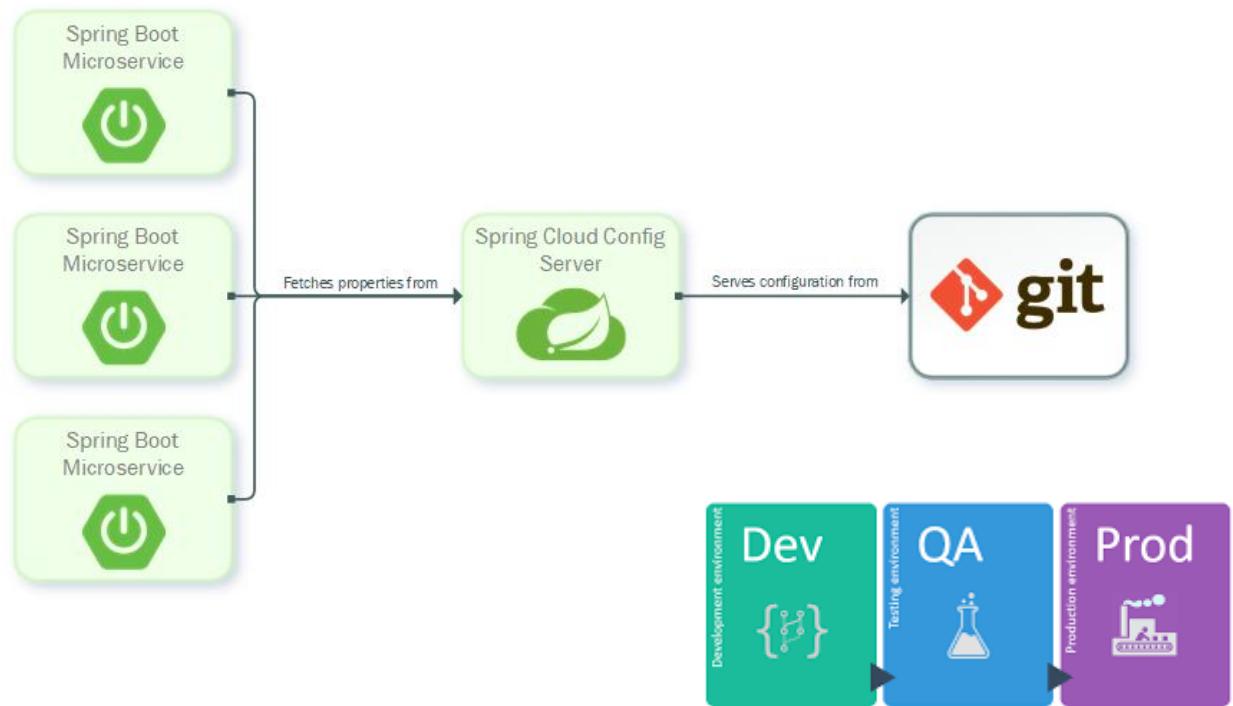


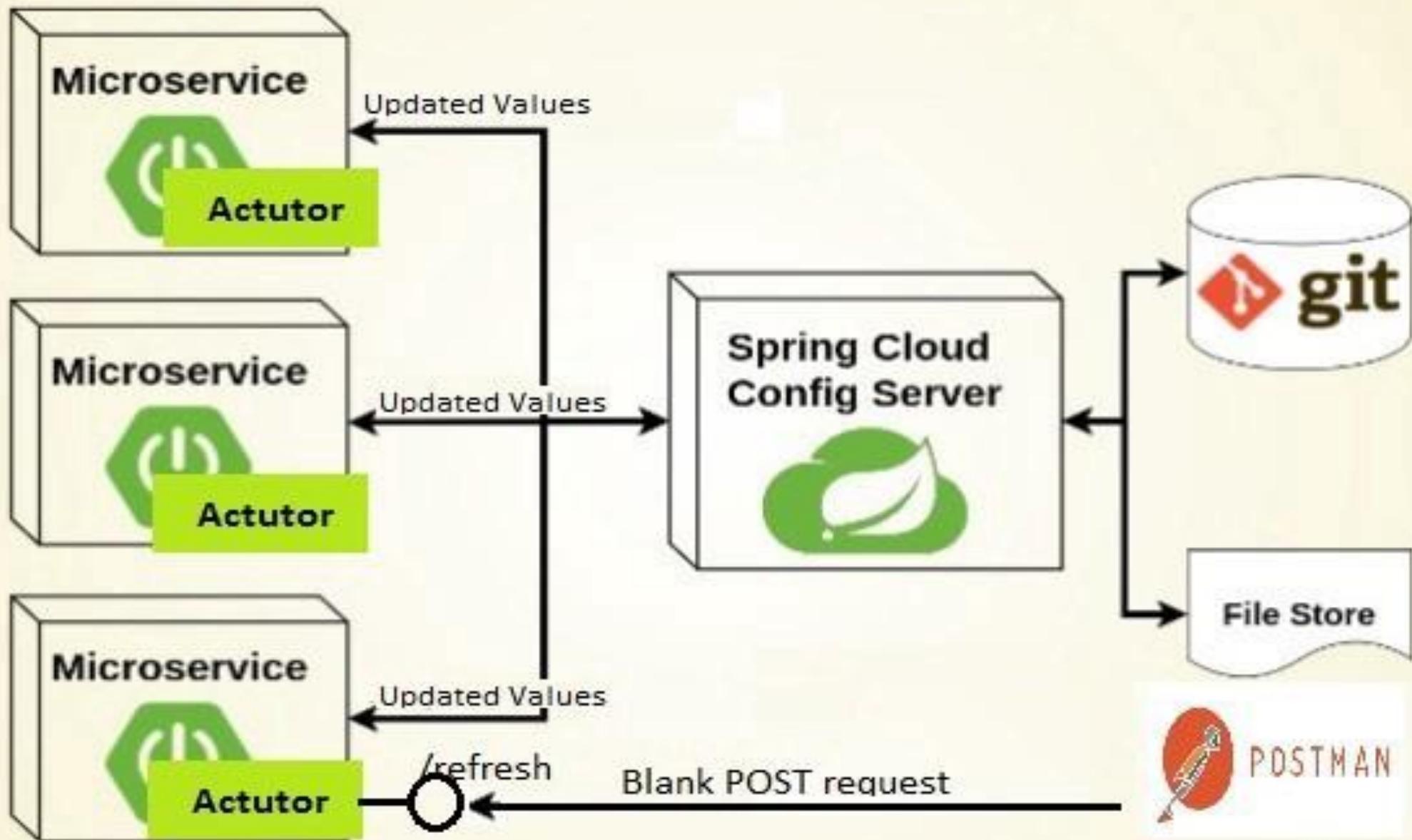
Pivotal **Cloud Foundry**

manifest.yml

3. CONFIGURATION

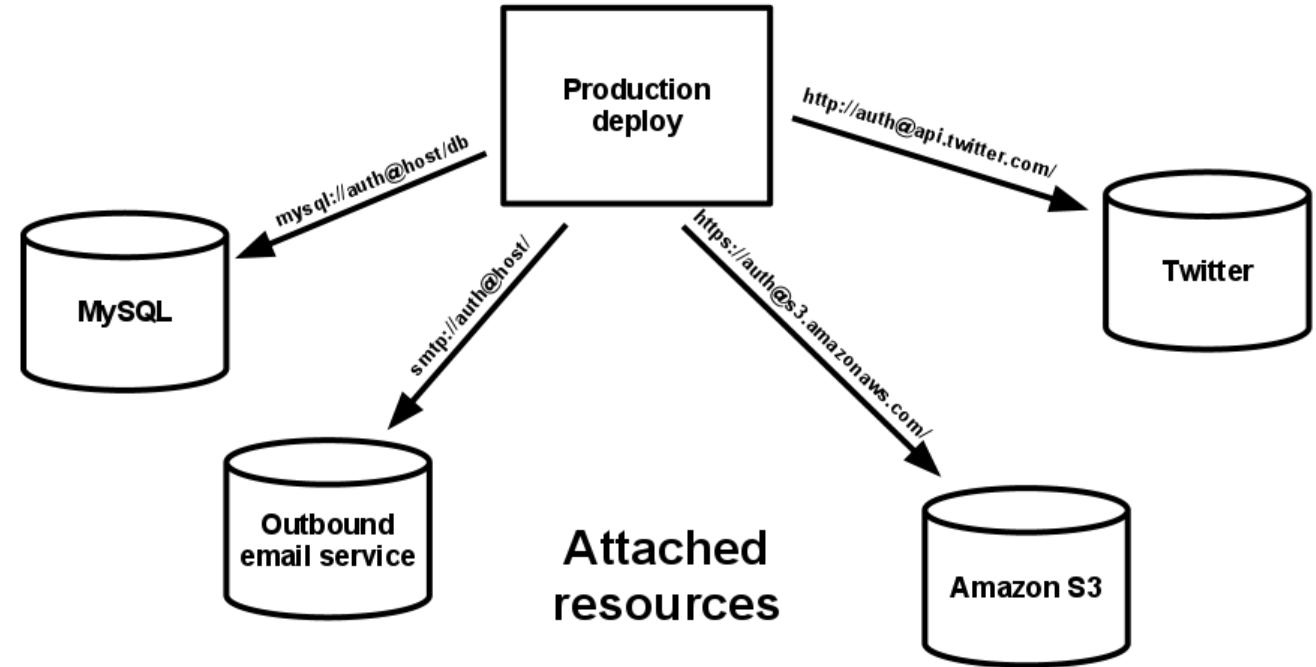
Store config in the environment





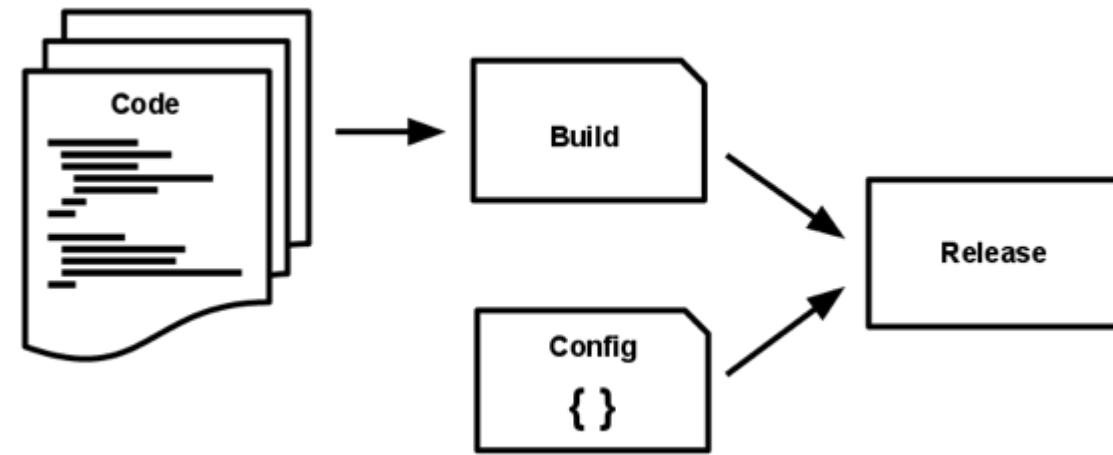
4. BACKING SERVICES

Treat backing services as attached resources



5. BUILD, RELEASE, RUN

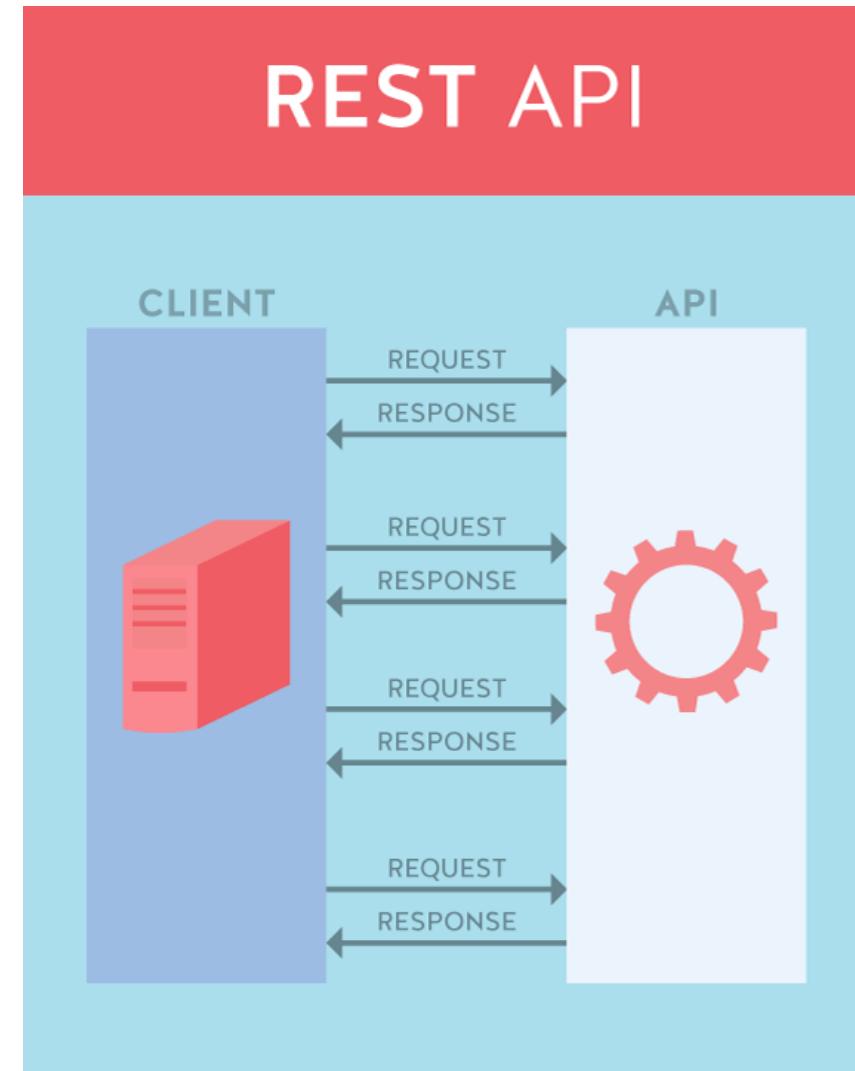
Strictly separate build
and run stages



Jenkins

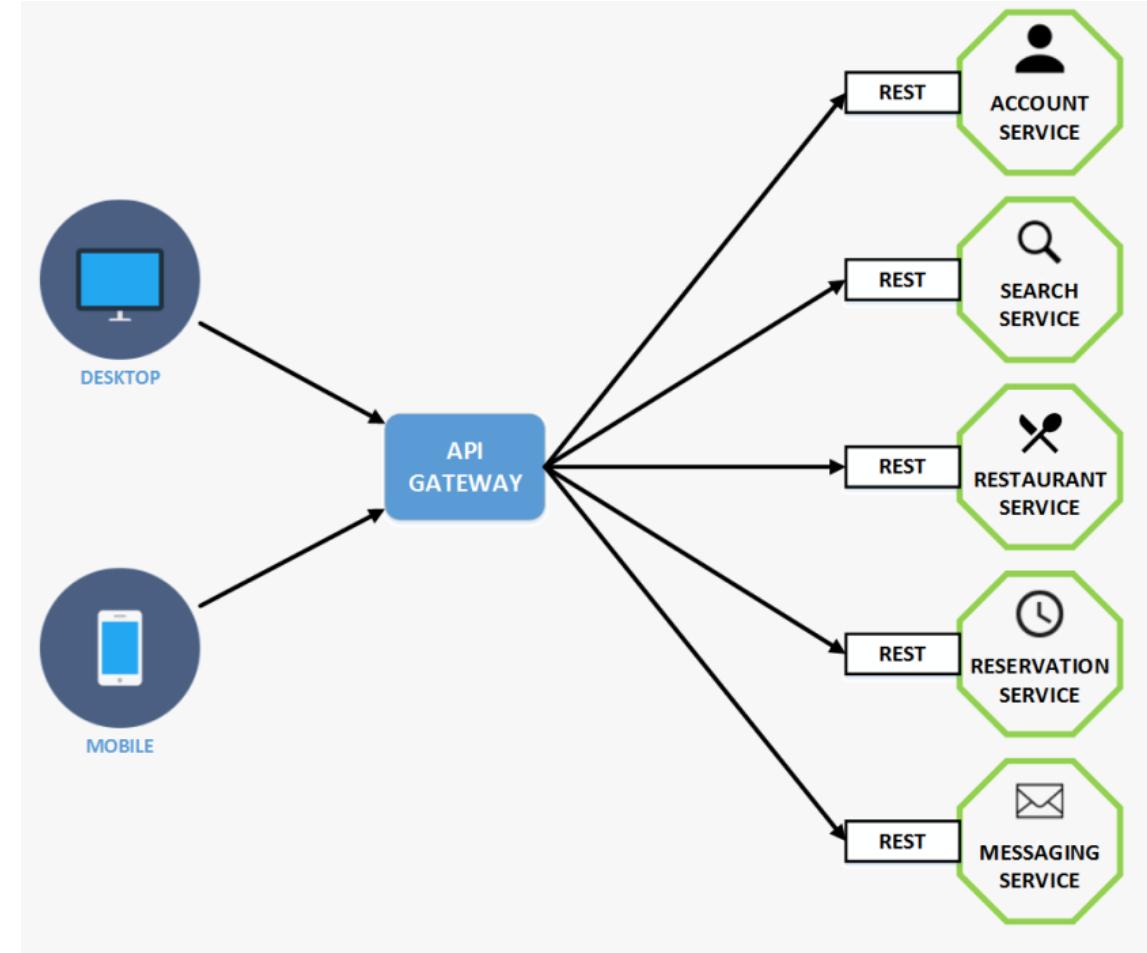
6. PROCESSES

Execute app as
stateless processes



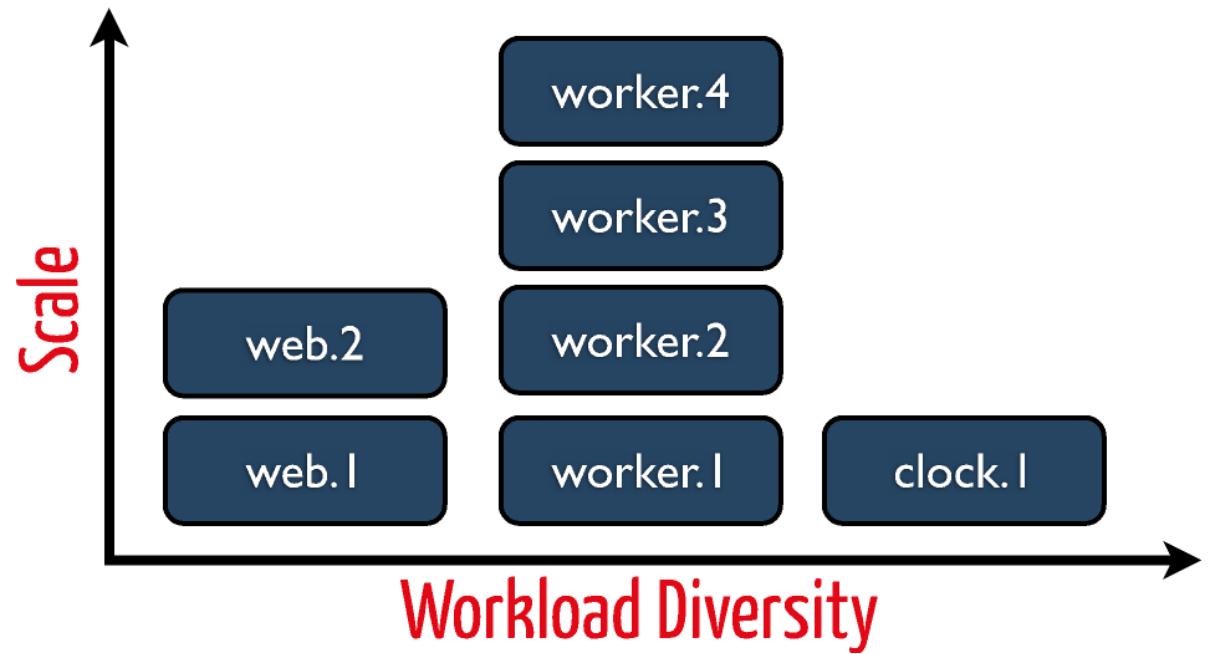
7. PORT BINDING

Export services via port binding



Processes

8. CONCURRENCY
Scale out via the process model



9. DISPOSABILITY

Maximize robustness
with fast startup
and graceful shutdown



10. DEV/ PROD PARITY

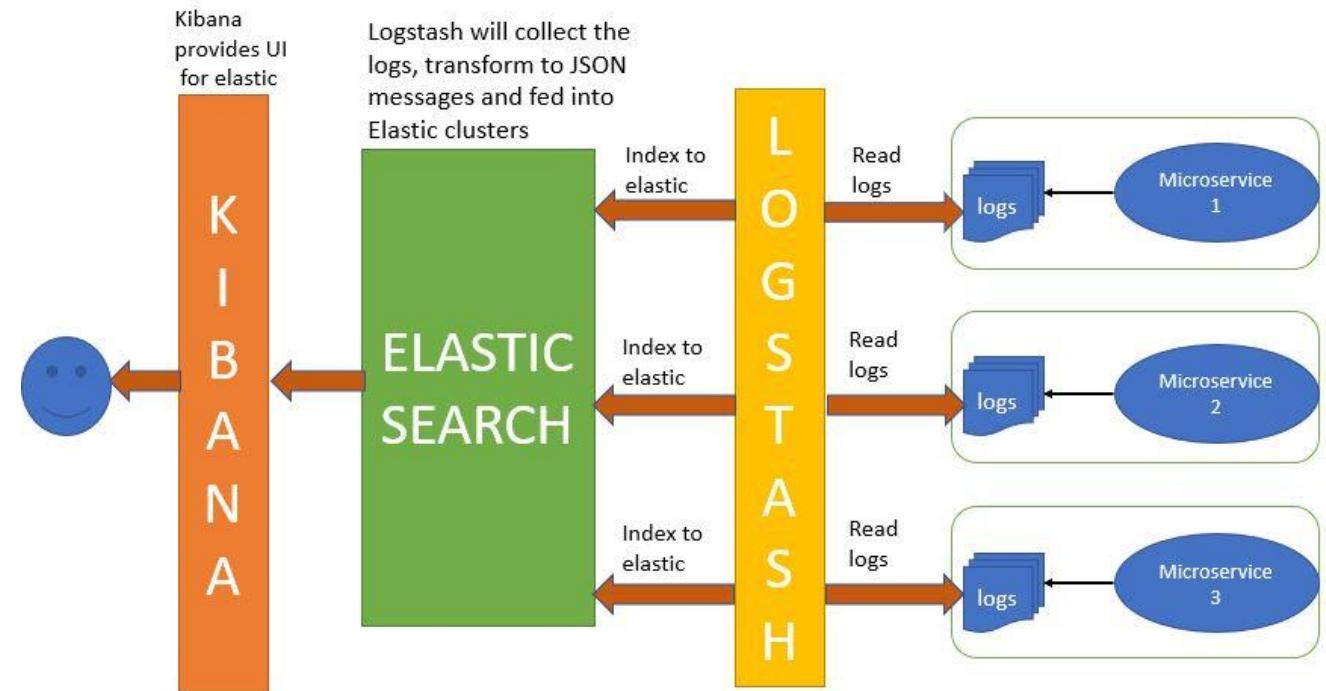
Keep dev, staging, prod as similar as possible

dev = staging = prod



11. LOGS

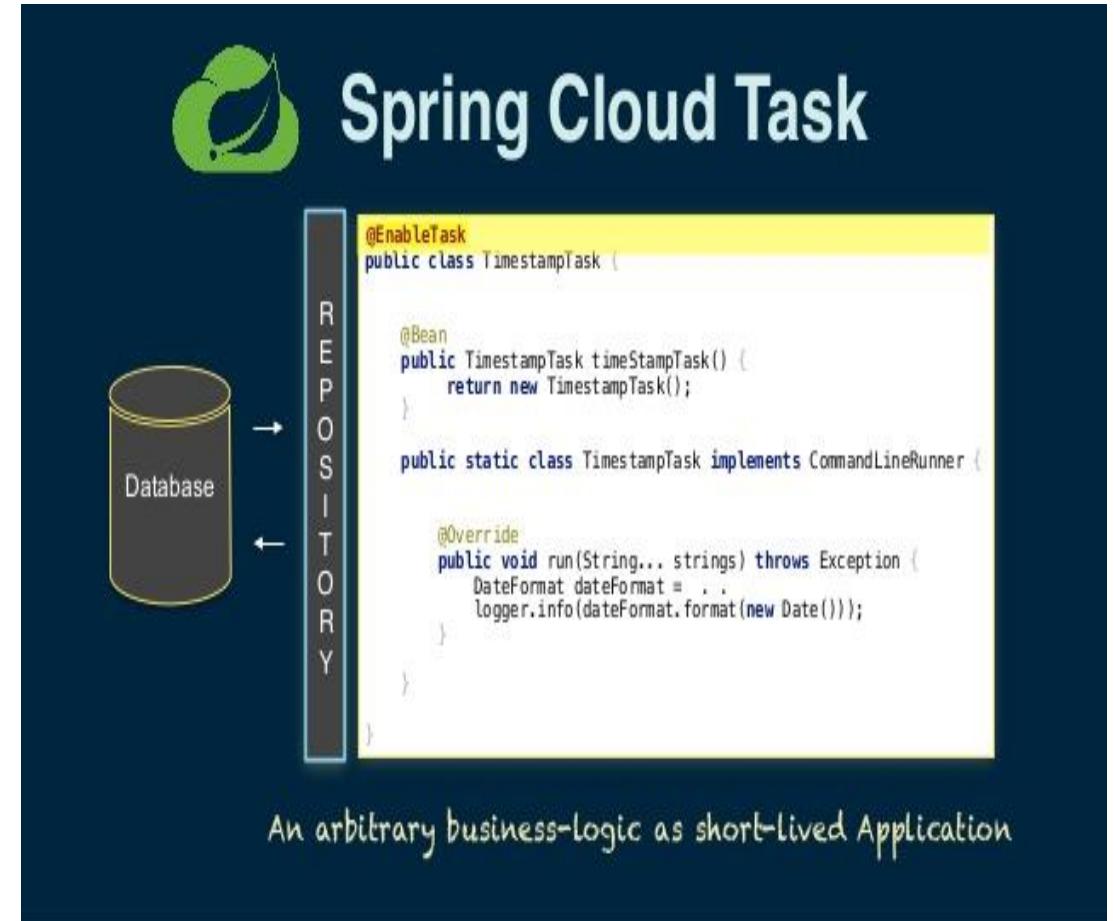
Treat logs as event stream

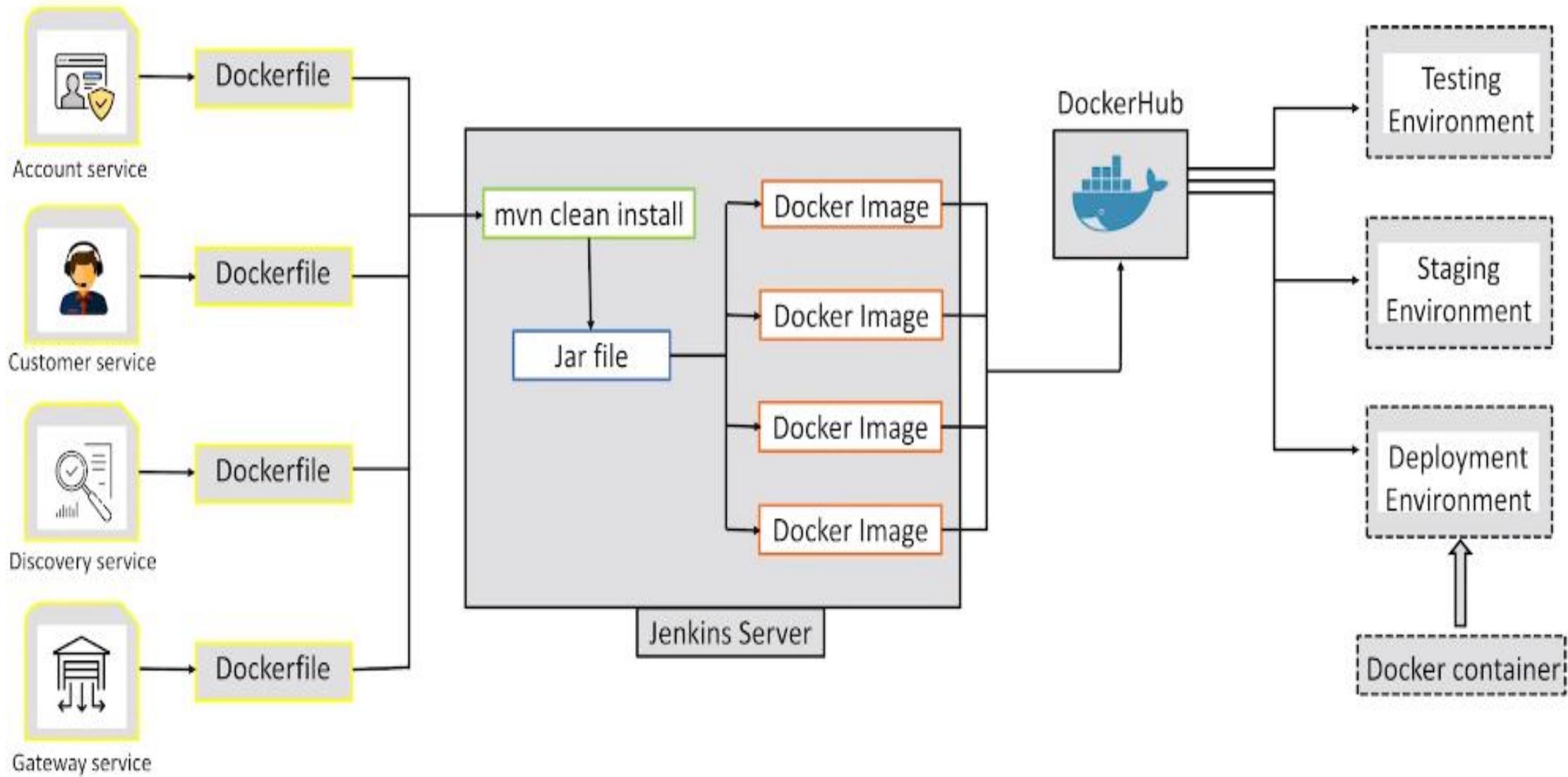


12. ADMIN PROCESSES

Run admin / mgmt tasks
as one-off processes

- Running database migrations
- Fetch analytical data to gather business insights
- Batch processing tasks





Communication Patterns in Microservices

Understanding Different Communication Patterns:

- ✓ Communication patterns define how services interact with each other within a microservices architecture.
- ✓ Two main communication patterns: Choreography and Event-Driven Architecture (EDA).
- ✓ Centralized vs. Decentralized: Choreography relies on decentralized communication between services, while EDA involves event-based communication.

Choreography and event-driven architecture (EDA)

Microservices Choreography

- Definition: A decentralized approach where each microservice makes decisions and communicates with other services directly.
- Characteristics:
 - Loose coupling: Services interact without relying on a central orchestrator.
 - Autonomy: Services are independent and make decisions based on local information.
- Benefits:
 - Scalability: Services can scale independently.
 - Flexibility: Changes in one service don't affect others directly.
- Challenges:
 - Complexity: Understanding interactions between services can be challenging.
 - Coordination: Ensuring consistency and reliability requires careful planning.

Event-Driven Architecture (EDA)

- Definition: An architecture where services communicate through events, often using a message broker.
- Characteristics:
 - Asynchronous communication: Services react to events as they occur.
 - Event-driven scalability: Handling events asynchronously allows for scalability.
- Benefits:
 - Loose coupling: Services are decoupled, leading to better isolation and independence.
 - Real-time processing: Enables real-time reactions to changes in the system.
- Challenges:
 - Eventual consistency: Ensuring consistency across services can be complex.
 - Event versioning: Managing changes to event schemas requires careful consideration.

Choreography:

- ✓ Choreography refers to a pattern where individual components (microservices, in the context of microservices architecture) interact with each other directly through the exchange of events.
- ✓ In a choreography-based system, each component is responsible for responding to events it receives and emitting events in response to its own actions.
- ✓ There's no centralized control or coordination mechanism; instead, the system's behavior emerges from the collective interactions of its components.
- ✓ Choreography promotes loose coupling and autonomy among components, enabling flexibility and scalability in distributed systems.

Event-Driven Architecture (EDA):

- ✓ Event-Driven Architecture is a broader architectural style that emphasizes the use of events as the primary means of communication and coordination between system components.
- ✓ EDA encompasses various patterns and practices for designing systems where events play a central role in representing state changes, business events, or meaningful occurrences.
- ✓ While choreography is one implementation of EDA, EDA can also involve other patterns such as event sourcing, event notification, and event-driven messaging.
- ✓ EDA can be applied at different levels of system architecture, including microservices, monolithic applications, and integration between disparate systems.

REST as a communication mechanism within a
choreographed microservices architecture

REST as a communication mechanism within a choreographed microservices architecture.

In a choreographed setup, each microservice communicates directly with other services to accomplish tasks, and RESTful HTTP APIs can serve as a means of communication between these services.

Decentralized Communication:

In a choreographed architecture, each microservice is responsible for making decisions and coordinating its activities with other services. RESTful HTTP APIs allow services to interact with each other in a decentralized manner.

Each service exposes its functionalities through REST endpoints, and other services can consume these endpoints to perform actions or retrieve data.

Loose Coupling:

RESTful APIs promote loose coupling between services. Each service is responsible for defining its own API contract (URI structure, HTTP methods, request/response formats), and other services interact with it based on this contract.

This loose coupling enables services to evolve independently without directly impacting each other.

HTTP as a Communication Protocol:

REST leverages HTTP as its communication protocol, which is widely supported and understood. HTTP provides a uniform interface for communication, making it easier to build and integrate services using RESTful APIs.

Services can leverage standard HTTP methods (GET, POST, PUT, DELETE) for CRUD operations and use status codes for indicating the outcome of requests.

Stateless Communication:

RESTful communication is inherently stateless, which aligns well with the principles of microservices architecture. Each HTTP request from a client contains all the information needed for the server to process it, without relying on any session state.

This statelessness simplifies the design and scalability of microservices.

Scalability and Performance:

RESTful APIs can be designed to support scalability and performance requirements. Services can be scaled horizontally to handle increased load by adding more instances, and RESTful communication allows for caching, asynchronous processing, and optimization techniques to improve performance.

REST API Best Practices

Resource Naming

- ✓ Resource naming is the foundation of a well-designed API.
- ✓ Use clear, descriptive nouns to represent resources (e.g., /user, /product).
- ✓ Avoid verbs in resource names; let HTTP methods dictate actions.
- ✓ Consistency in naming conventions across the API enhances readability and usability.

Using HTTP Methods Correctly

- HTTP methods (GET, POST, PUT, DELETE) define CRUD operations.
- GET: Retrieve data, POST: Create data, PUT: Update data, DELETE: Delete data.
- Proper usage of HTTP status codes (e.g., 200 for success, 404 for not found) enhances API reliability and interoperability.

Resource Endpoint Design

- ✓ Design clear and intuitive endpoints for resources.
- ✓ Utilize sub-resources for hierarchical data representation.
- ✓ Versioning endpoints (e.g., /v1/user) facilitates future updates without breaking existing clients.

Error Handling

- ✓ Clear and informative error messages aid developers in troubleshooting.
- ✓ Standardized error response format (e.g., JSON) for consistency.
- ✓ Proper use of HTTP status codes (e.g., 400 for client errors, 500 for server errors) communicates issues effectively.

Pagination and Filtering

- ✓ Pagination manages large datasets by limiting the number of returned results.
- ✓ Filtering allows clients to query specific data based on parameters.
- ✓ Standardized query parameters (e.g., limit, offset, sort) enhance API consistency and usability.

Input Validation

- ✓ Validate incoming data to ensure consistency and security.
- ✓ Sanitize user input to prevent injection attacks.
- ✓ Implement validation for data types, formats, and constraints (e.g., email format, password strength).

API Documentation

- ✓ Comprehensive documentation is essential for API adoption and integration.
- ✓ Utilize tools like Swagger or OpenAPI for automatic documentation generation.
- ✓ Include detailed descriptions of endpoints, parameters, request/response formats, and error handling.

- ✓ Following REST API best practices is crucial for building scalable, maintainable, and secure APIs.
- ✓ Consistent resource naming, correct HTTP method usage, clear endpoint design, robust error handling, efficient pagination and filtering, thorough input validation, and comprehensive documentation are key factors for success.
- ✓ Continuous learning and improvement in API development practices are essential for staying ahead in the ever-evolving tech landscape.

Microservice Patterns

Microservice Patterns:

- API gateway
- Service registry
- Circuit breaker
- Messaging
- Database per Service
- Access Token
- Saga
- Event Sourcing & CQRS

Microservices Tooling Supports

1. Using Spring for creating Microservices
 - ❑ Setup new service by using Spring Boot
 - ❑ Expose resources via a RestController
 - ❑ Consume remote services using RestTemplate/Feign

2. Adding Spring Cloud and Discovery server

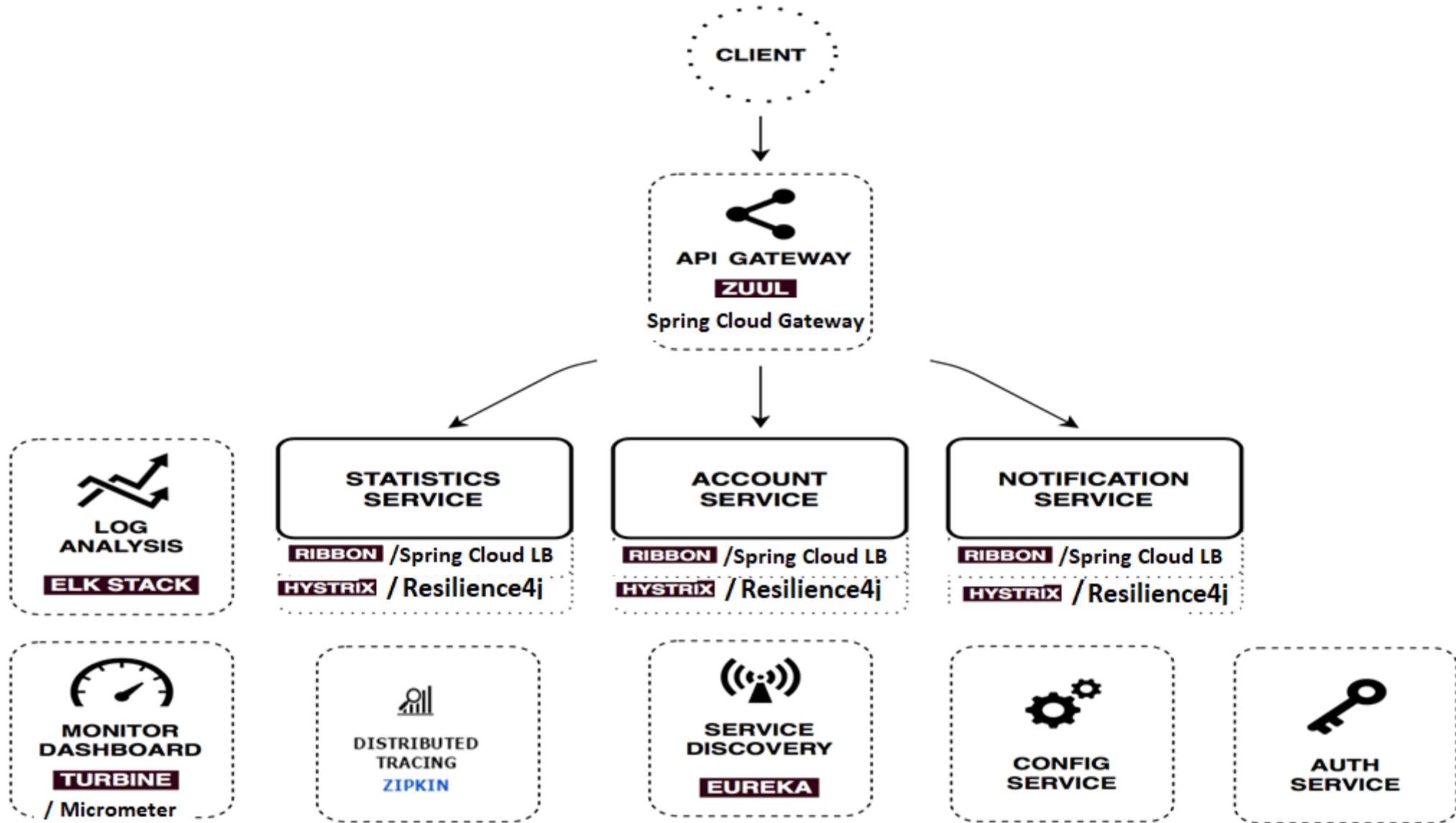
- ❑ It is building blocks for Cloud and Microservices
- ❑ It provides microservices infrastructure like provide use services such as Service Discovery, Configuration server and Monitoring.
- ❑ It provides several other open source projects like Netflix OSS
- .
- ❑ It provides PaaS like Cloud Foundry, AWS etc.,
- ❑ It uses Spring Boot style starters

spring-cloud-netflix

This project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components.

The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon)



Service Discovery:

Eureka instances can be registered and clients can discover the instances using Spring-managed beans

Circuit Breaker:

Hystrix/Resilience4j clients can be built with a simple annotation-driven method decorator
Embedded Hystrix/Micrometer dashboard with declarative Java configuration

Declarative REST Client:

Feign creates a dynamic implementation of an interface decorated with JAX-RS or Spring MVC annotations

Client Side Load Balancer:

Ribbon / Spring Cloud Load Balancer

External Configuration:

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

Router and Filter:

Automatic registration of **Zuul/Spring Cloud Gateway** filters, and a simple convention over configuration approach to reverse proxy creation

ELK – Elasticsearch, Logstash, Kibana:

three different tools usually used together.

They are used for searching, analyzing, and visualizing log data in a real time.

Zipkin is a distributed tracing system

It helps gather timing data needed to troubleshoot latency problems in microservice architectures.

It manages both the collection and lookup of this data.

Problem

How do clients of a service (in the case of Client-side discovery) and/or routers (in the case of Server-side discovery) know about the available instances of a service?



Solution

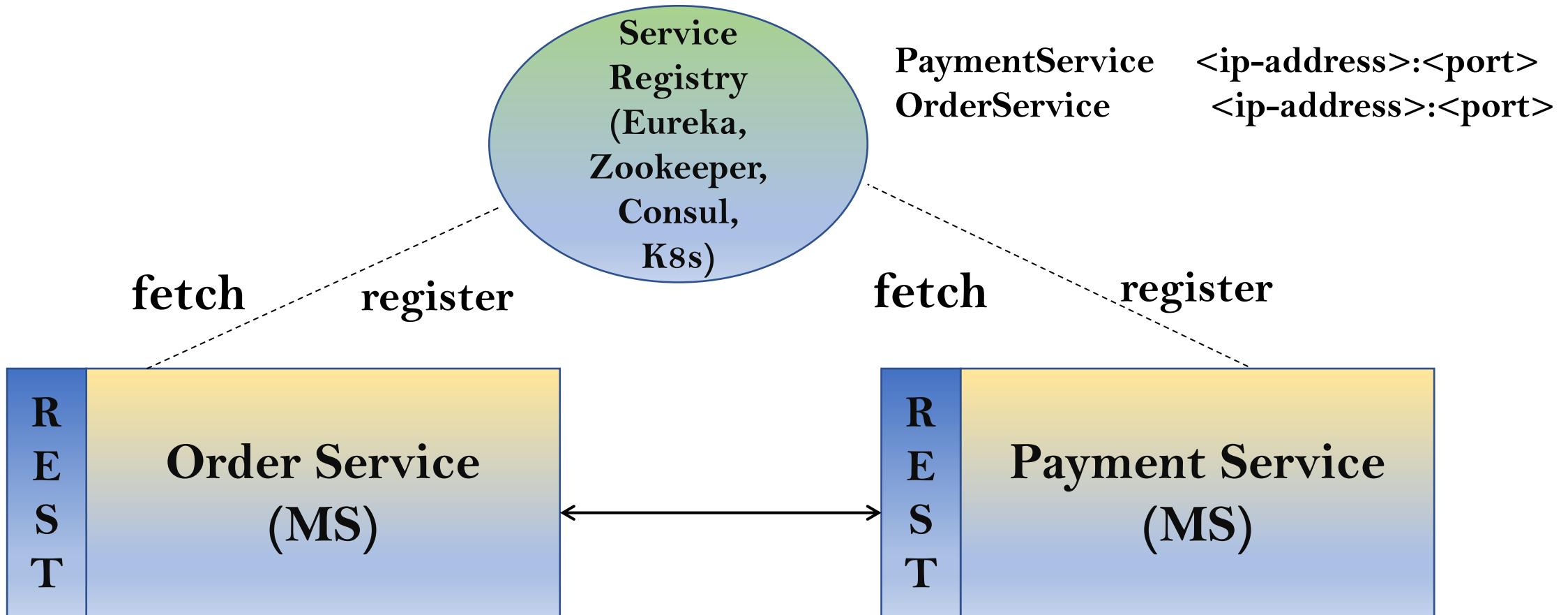
Implement a service registry, which is a database of services, their instances and their locations.

Service instances are registered with the service registry on startup and deregistered on shutdown.

Client of the service and/or routers query the service registry to find the available instances of a service.

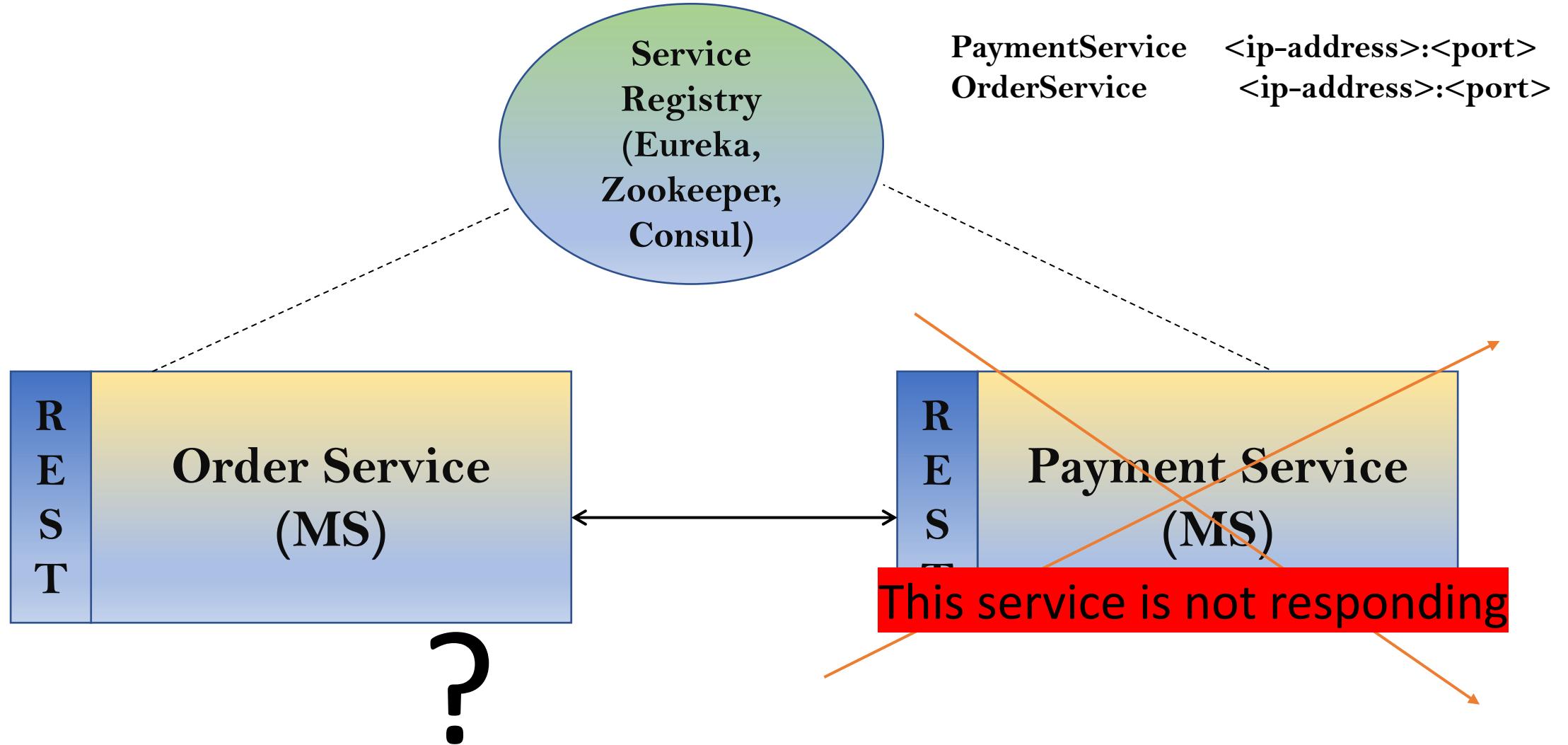
A service registry might invoke a service instance's health check API to verify that it is able to handle requests

Service registry Pattern



Problem

How to prevent a network or service failure from cascading to other services?



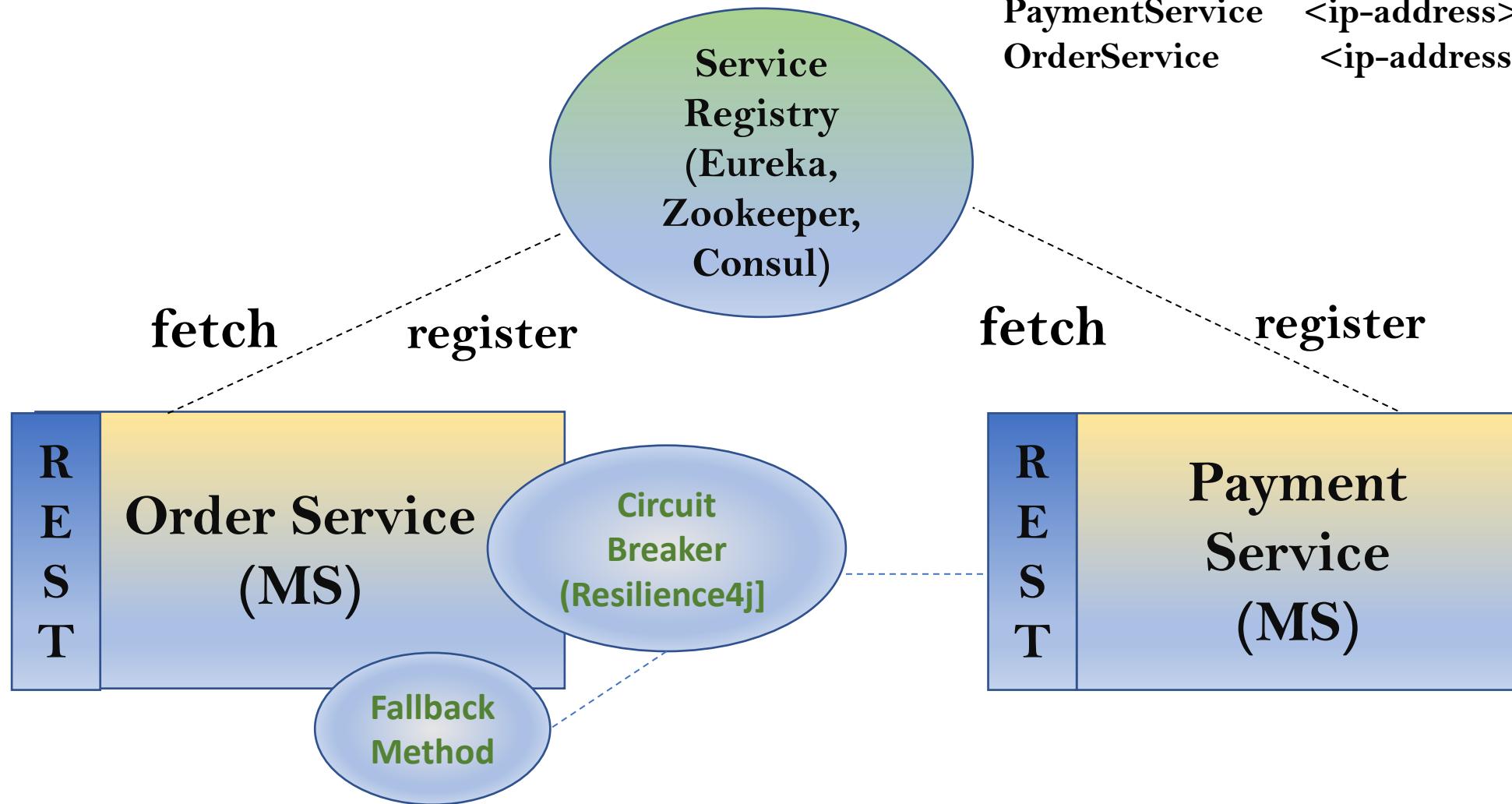
Solution

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker.

When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately.

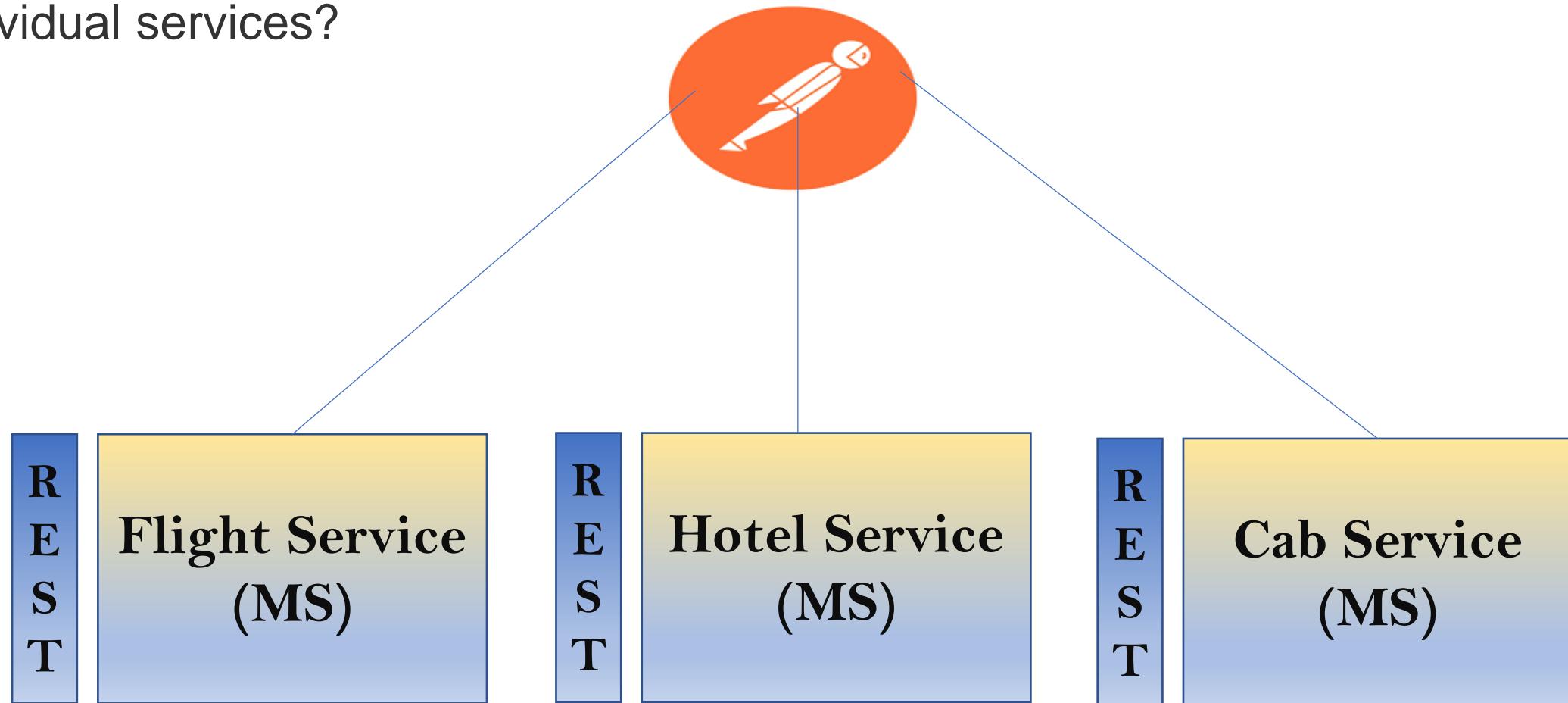
After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

Circuit Breaker Pattern



Problem

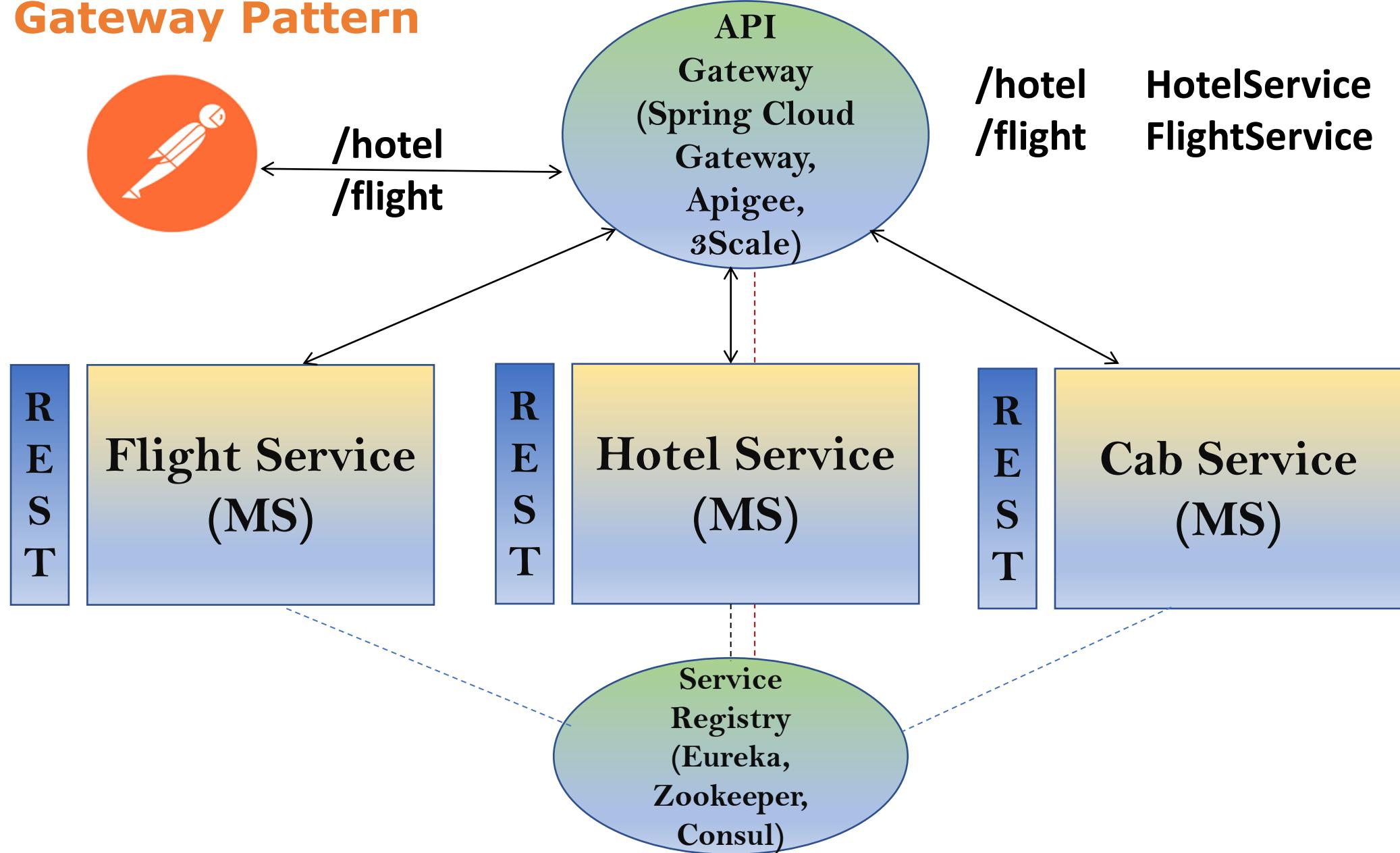
How do the clients of a Microservices-based application access the individual services?



Solution

Implement an API gateway that is the single entry point for all clients. The API gateway handles

API Gateway Pattern



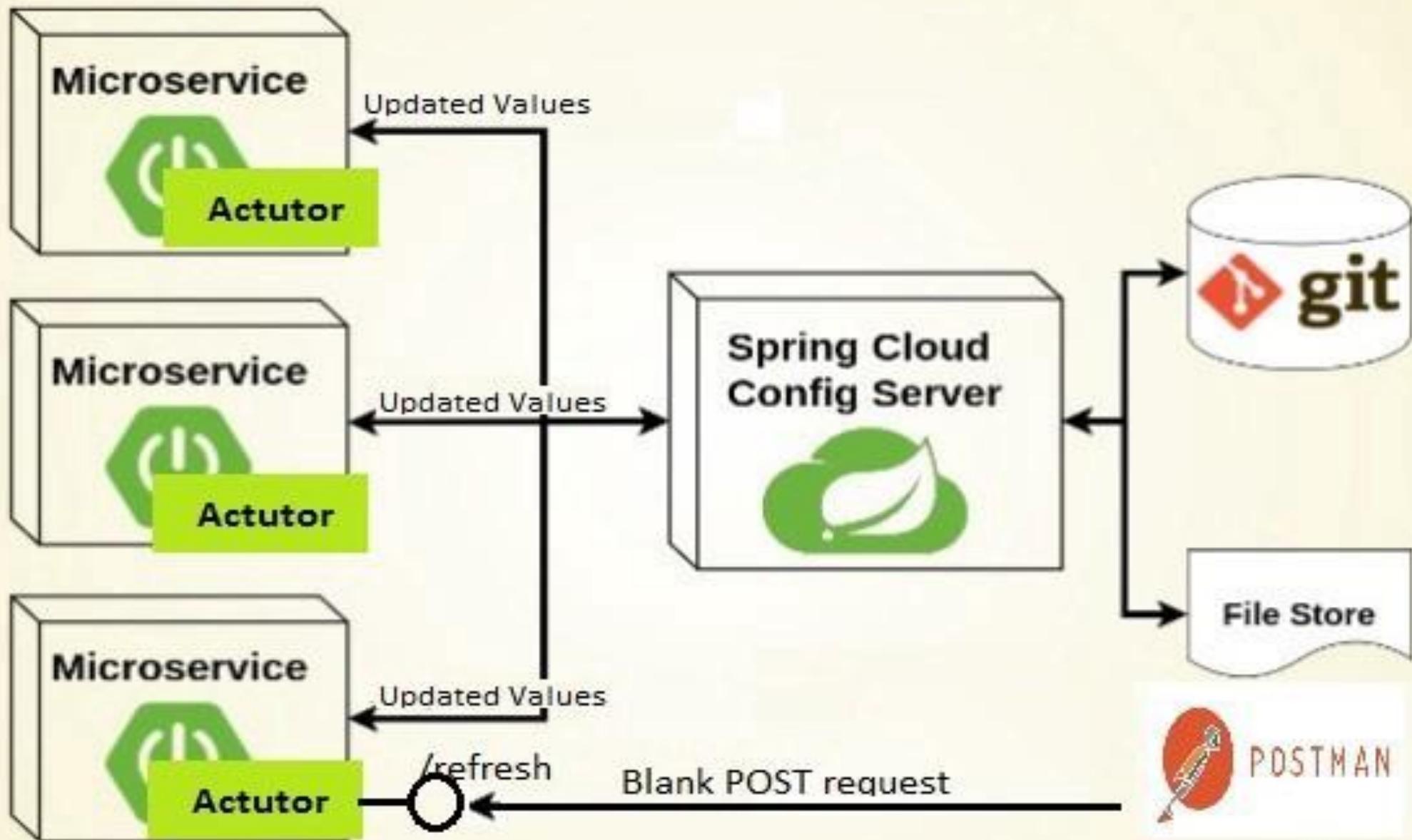
Pattern: Externalized configuration

Problem

How to enable a service to run in multiple environments without modification?

Solution

Externalize all application configuration including the database credentials and network location. On startup, a service reads the configuration from an external source, e.g. OS environment variables, etc.



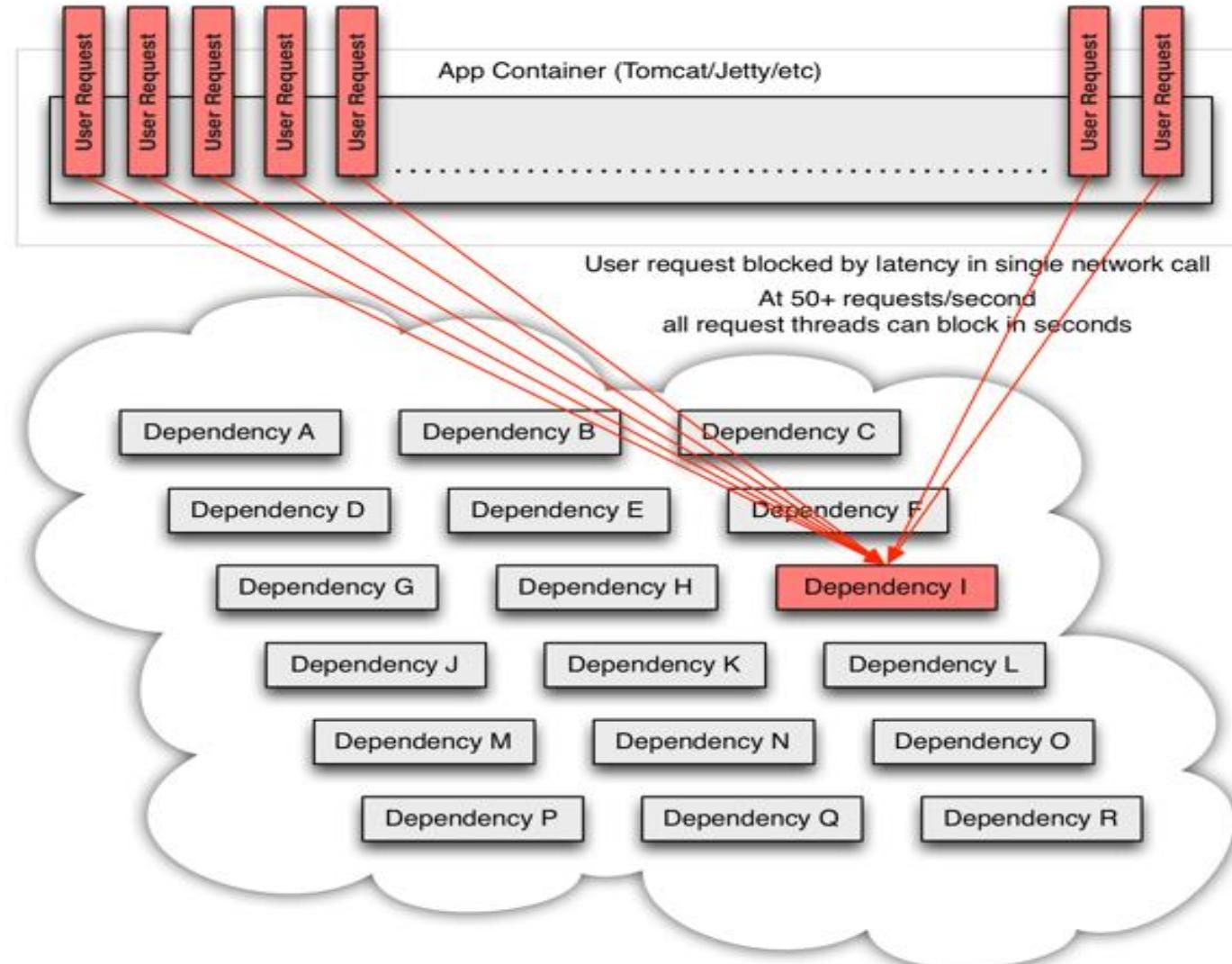
Container design patterns (Distributed Systems)

- The sidecar design pattern
- The ambassador design pattern
- The adapter design pattern
- The leader election design pattern
- The work queue design pattern
- The scatter/gather design pattern

Latency and Fault Tolerance for Distributed Systems

(Circuit Breaker Pattern – Hystrix / Resilience4j)

With high volume traffic a single backend dependency becoming latent can cause all resources to become saturated in seconds on all servers.



These issues are even worse when network access is performed through a third-party client — a “black box” where implementation details are hidden and can change at any time, and network or resource configurations are different for each client library and often difficult to monitor and change.

All of these represent failure and latency that needs to be isolated and managed so that a single failing dependency can’t take down an entire application or system.

Pattern: Circuit Breaker

Problem

How to prevent a network or service failure from cascading to other services?

Solution

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker.

When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately.

After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

CURRENT	REPLACEMENT
Hystrix	Resilience4j
Hystrix Dashboard / Turbine	Micrometer + Monitoring System
Ribbon	Spring Cloud Loadbalancer
Zuul 1	Spring Cloud Gateway
Archaius 1	Spring Boot external config + Spring Cloud Config

Resilience4j

Resilience4j has been inspired by Netflix Hystrix but is designed for Java 8 and functional programming.

It is lightweight compared to Hystrix as it has the Vavr library as its only dependency.

Vavr (Javaslang) core is a functional library for Java. It helps to reduce the amount of code and to increase the robustness.

A first step towards functional programming is to start thinking in immutable values.

Vavr provides immutable collections and the necessary functions and control structures to operate on these values.

Resilience4j

```
@Bulkhead(name = "bulkHeadPostToES")
public String postCallProxyES (String endpoint, String payload) {
    return this.initiateCallEsProxy (endpoint, payload);
}
```

```
bulkhead:
  backends:
    bulkHeadPostToES:
      maxWaitDuration: 20ms
      maxConcurrentCalls: 20

thread-pool-bulkhead:
  backends:
    bulkHeadPostToES:
      maxThreadPoolSize: 1
      coreThreadPoolSize: 1
      queueCapacity: 1
```

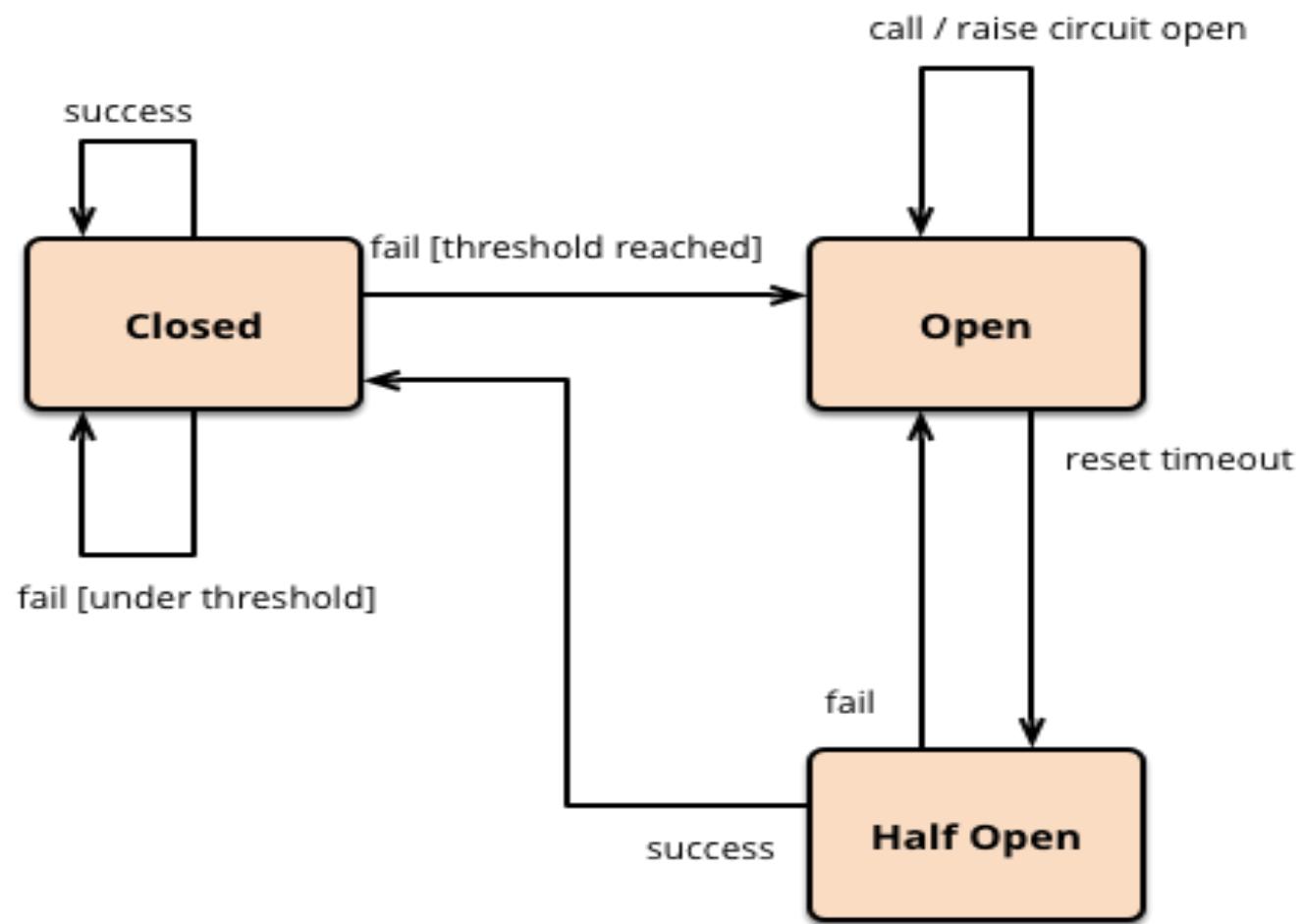
CircuitBreaker

When a service invokes another service, there is always a possibility that it may be down or having high latency.

This may lead to exhaustion of the threads as they might be waiting for other requests to complete.

This pattern functions in a similar fashion to an electrical Circuit Breaker:

- When a number of consecutive failures cross the defined threshold, the Circuit Breaker trips.
- For the duration of the timeout period, all requests invoking the remote service will fail immediately.
- After the timeout expires the Circuit Breaker allows a limited number of test requests to pass through.
- If those requests succeed the Circuit Breaker resumes normal operation.
- Otherwise, if there is a failure the timeout period begins again.



RateLimiter

Rate Limiting pattern ensures that a service accepts only a defined maximum number of requests during a window.

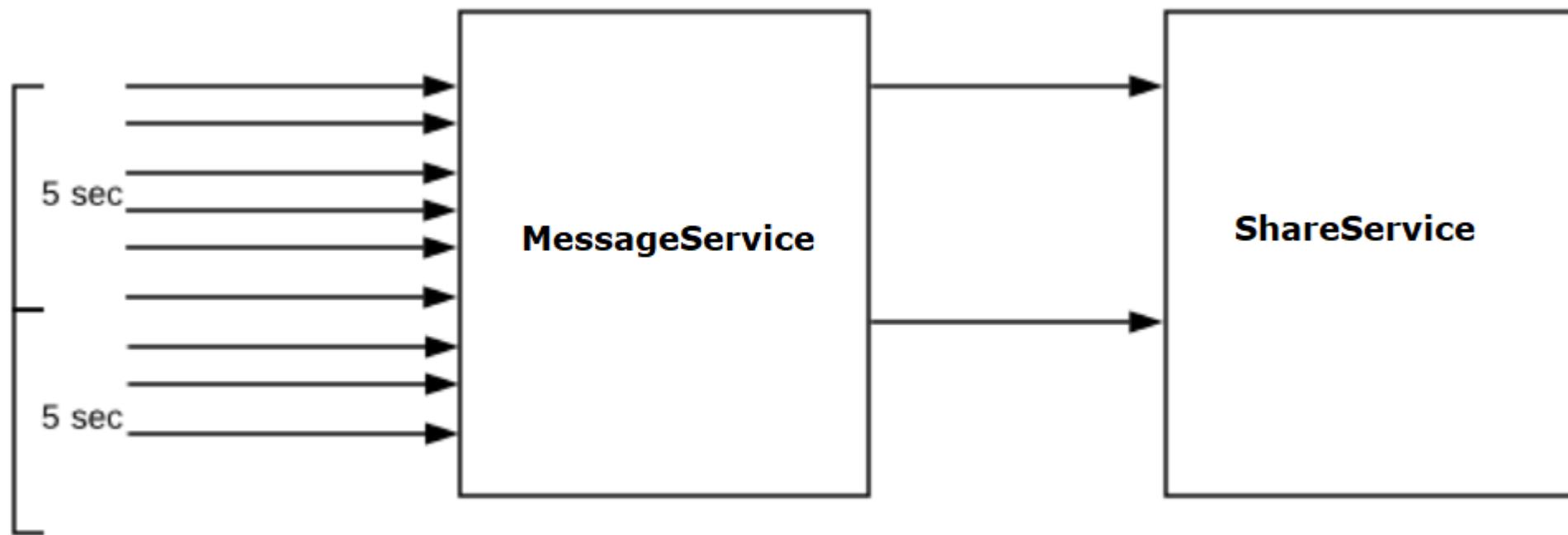
This ensures that underline resources are used as per their limits and don't exhaust.

Retry

Retry pattern enables an application to handle transient failures while calling to external services.

It ensures retrying operations on external resources a set number of times. If it doesn't succeed after all the retry attempts, it should fail and response should be handled gracefully by the application.

RateLimiter



Bulkhead

Bulkhead ensures the failure in one part of the system doesn't cause the whole system down. It controls the number of concurrent calls a component can take. This way, the number of resources waiting for the response from that component is limited.

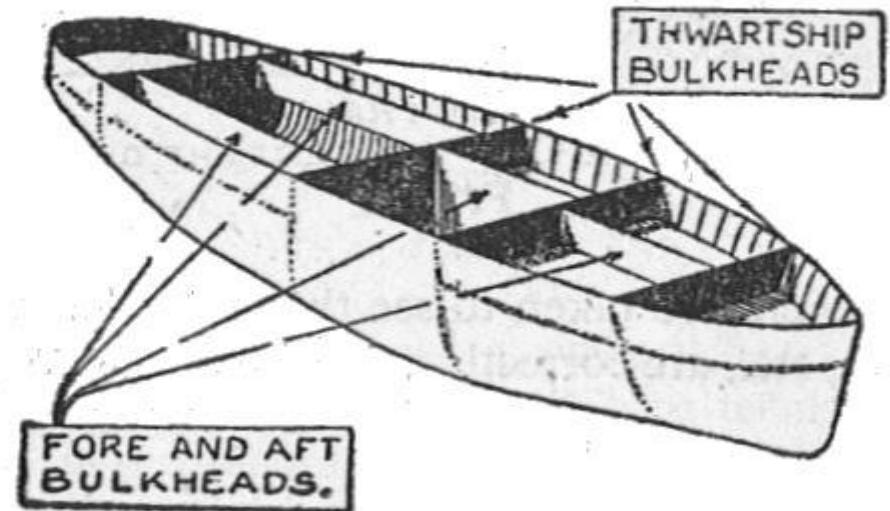
There are two types of bulkhead implementation:

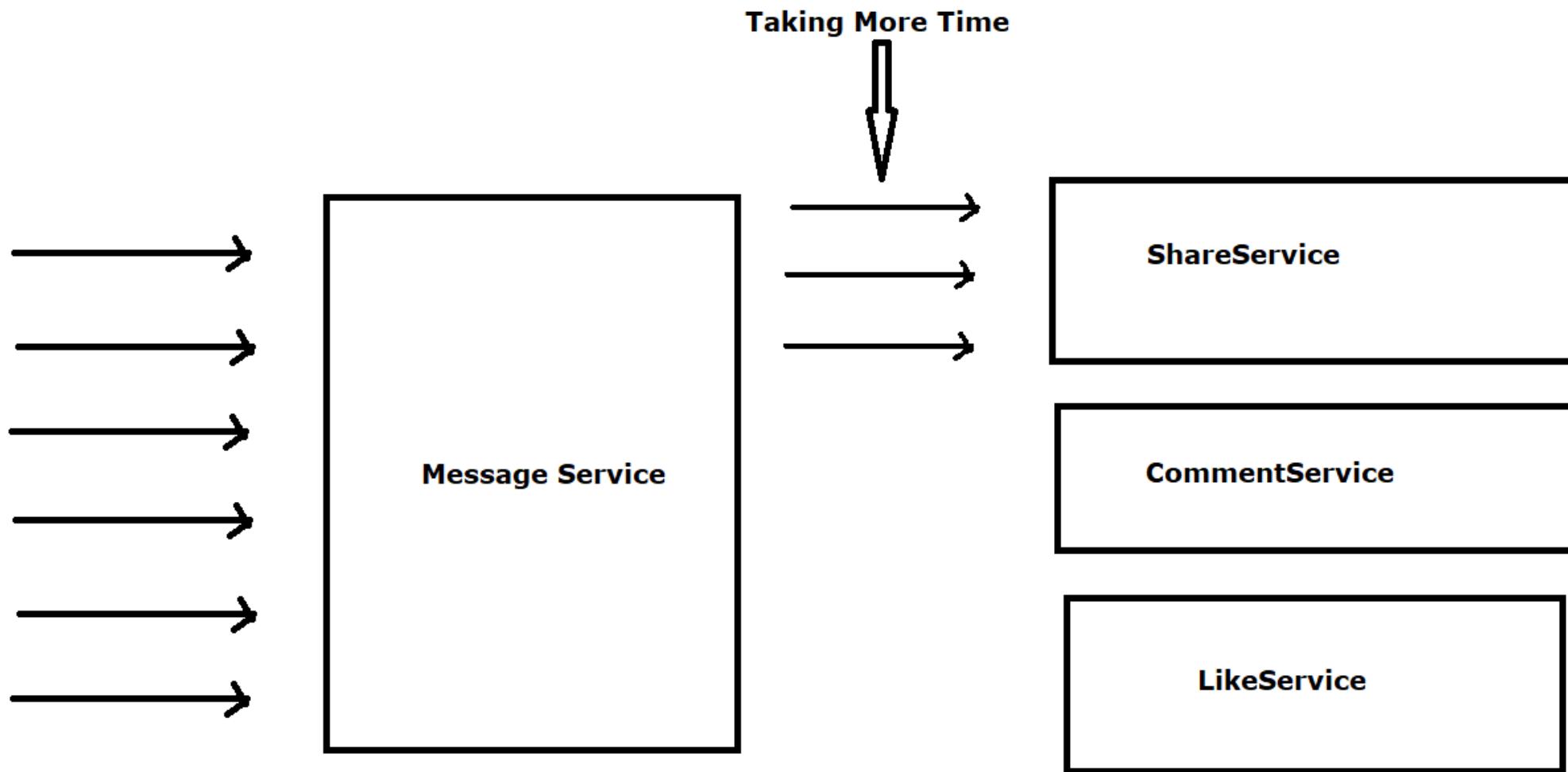
The semaphore isolation approach limits the number of concurrent requests to the service. It rejects requests immediately once the limit is hit. The thread pool isolation approach uses a thread pool to separate the service from the caller and contain it to a subset of system resources.

The thread pool approach also provides a waiting queue, rejecting requests only when both the pool and queue are full. Thread pool management adds some overhead, which slightly reduces performance compared to using a semaphore, but allows hanging threads to time out.

A ship is split into small multiple compartments using Bulkheads. Bulkheads are used to seal parts of the ship to prevent entire ship from sinking in case of flood.

Similarly failures should be expected when we design software. The application should be split into multiple components and resources should be isolated in such a way that failure of one component is not affecting the other.





CircuitBreaker

Config property	Default Value	Description
failureRateThreshold	50	Configures the failure rate threshold in percentage. When the failure rate is equal or greater than the threshold the CircuitBreaker transitions to open and starts short-circuiting calls.
slowCallRateThreshold	100	Configures a threshold in percentage. The CircuitBreaker considers a call as slow when the call duration is greater than <code>slowCallDurationThreshold</code> . When the percentage of slow calls is equal or greater the threshold, the CircuitBreaker transitions to open and starts short-circuiting calls.
slowCallDurationThreshold	60000 [ms]	Configures the duration threshold above which calls are considered as slow and increase the rate of slow calls.
permittedNumberOfCallsInHalfOpenState	10	Configures the number of permitted calls when the CircuitBreaker is half open.

slidingWindowType	COUNT_BASED	<p>Configures the type of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed.</p> <p>Sliding window can either be count-based or time-based.</p> <p>If the sliding window is COUNT_BASED, the last <code>slidingWindowSize</code> calls are recorded and aggregated.</p> <p>If the sliding window is TIME_BASED, the calls of the last <code>slidingWindowSize</code> seconds recorded and aggregated.</p>
slidingWindowSize	100	<p>Configures the size of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed.</p>

minimumNumberOfCalls	10	<p>Configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate.</p> <p>For example, if minimumNumberOfCalls is 10, then at least 10 calls must be recorded, before the failure rate can be calculated.</p> <p>If only 9 calls have been recorded the CircuitBreaker will not transition to open even if all 9 calls have failed.</p>
waitDurationInOpenState	60000 [ms]	<p>The time that the CircuitBreaker should wait before transitioning from open to half-open.</p>

Create and configure a Bulkhead

You can provide a custom global BulkheadConfig. In order to create a custom global BulkheadConfig, you can use the BulkheadConfig builder. You can use the builder to configure the following properties.

Config property	Default value	Description
maxConcurrentCalls	25	Max amount of parallel executions allowed by the bulkhead
maxWaitDuration	0	Max amount of time a thread should be blocked for when attempting to enter a saturated bulkhead.

RateLimiter

Config property	Default value	Description
timeoutDuration	5 [s]	The default wait time a thread waits for a permission
limitRefreshPeriod	500 [ns]	The period of a limit refresh. After each period the rate limiter sets its permissions count back to the limitForPeriod value
limitForPeriod	50	The number of permissions available during one limit refresh period

Retry

Config property	Default value	Description
maxAttempts	3	The maximum number of retry attempts
waitDuration	500 [ms]	A fixed wait duration between retry attempts
intervalFunction	numOfAttempts -> waitDuration	A function to modify the waiting interval after a failure. By default the wait duration remains constant.
retryOnResultPredicate	result -> false	Configures a Predicate which evaluates if a result should be retried. The Predicate must return true, if the result should be retried, otherwise it must return false.
retryOnExceptionPredicate	throwable -> true	Configures a Predicate which evaluates if an exception should be retried. The Predicate must return true, if the exception should be retried, otherwise it must return false.
retryExceptions	empty	Configures a list of error classes that are recorded as a failure and thus are retried.
ignoreExceptions	empty	Configures a list of error classes that are ignored and thus are not retried.

```
RetryConfig config = RetryConfig.custom()
    .maxAttempts(2)
    .waitDuration(Duration.ofMillis(1000))
    .retryOnResult(response -> response.getStatus() == 500)
    .retryOnException(e -> e instanceof WebServiceException)
    .retryExceptions(IOException.class, TimeoutException.class)
    .ignoreExceptions(BusinessException.class,
                      OtherBusinessException.class)
    .build();
```

resilience4j has the ability to add multiple fault tolerance features into one call. It is more configurable and amount of code needs to be written is less with right amount of abstractions.

```
@CircuitBreaker(name = BACKEND, fallbackMethod = "fallback")
@RateLimiter(name = BACKEND)
@Bulkhead(name = BACKEND)
@Retry(name = BACKEND, fallbackMethod = "fallback")
@TimeLimiter(name = BACKEND)
Public String postCallProxyES(...) {
    ...
}
```

The default Resilience4j Aspects order is the following:

Retry (CircuitBreaker (RateLimiter (TimeLimiter (Bulkhead (Function)))))

Bulkhead

Limit number of concurrent calls at a time.

Ex: Allow 5 concurrent calls at a time

In above example, first 5 calls will start processing in parallel while any further calls will keep waiting. As soon as one of those 5 in-process calls is finished, next waiting calls will be immediately eligible to be executed.

Rate Limiter

Limit number total calls in given period of time

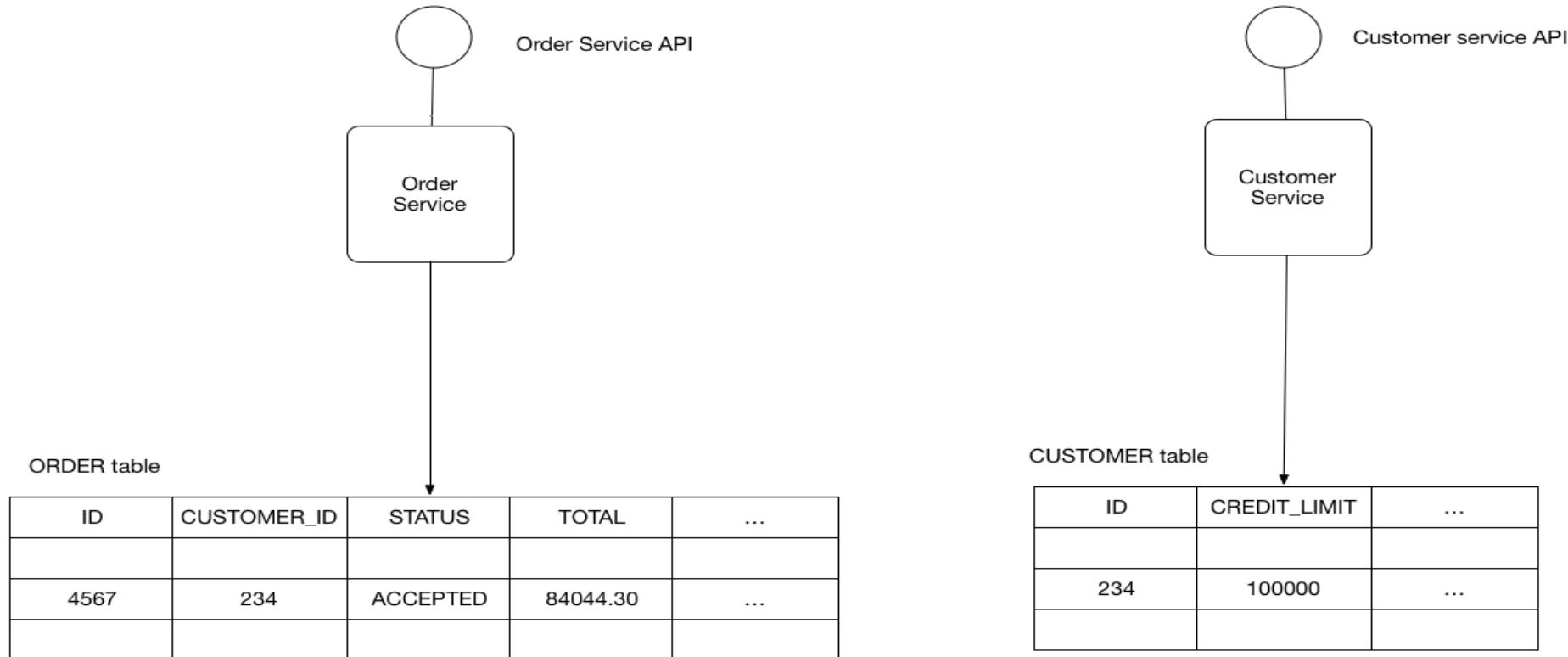
Ex: Allow 5 calls every 2 second.

In above example, 2 second window starts & first 5 calls will start processing (may be parallel or not parallel). Those 5 calls might finish before 2 seconds, but next waiting calls will NOT be immediately eligible to be executed. After 2 seconds window is over, next 2 seconds window will start & then only next waiting calls will be eligible to be executed.

Problem

What's the database architecture in a microservices application?

Pattern: Shared database



Pattern: Shared database

The benefits of this pattern are:

A developer uses familiar and straightforward ACID transactions to enforce data consistency

A single database is simpler to operate

Pattern: Shared database

The drawbacks of this pattern are:

Development time coupling - a developer working on, for example, the OrderService will need to coordinate schema changes with the developers of other services that access the same tables. This coupling and additional coordination will slow down development.

Runtime coupling - because all services access the same database they can potentially interfere with one another. For example, if long running CustomerService transaction holds a lock on the ORDER table then the OrderService will be blocked.

Single database might not satisfy the data storage and access requirements of all services.

Pattern: Database per service

Resulting context

Using a database per service has the following benefits:

Helps ensure that the services are loosely coupled.

Changes to one service's database does not impact any other services.

Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo

Using a database per service has the following drawbacks:

Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided. Moreover, many modern (NoSQL) databases don't support them.

Implementing queries that join data that is now in multiple databases is challenging.

Complexity of managing multiple SQL and NoSQL databases

There are various patterns/solutions for implementing transactions and queries that span services:

Implementing transactions that span services - use the **Saga** pattern.

Implementing queries that span services:

API Composition - the application performs the join rather than the database.

Command Query Responsibility Segregation (CQRS) - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each service publishes when it updates its data.

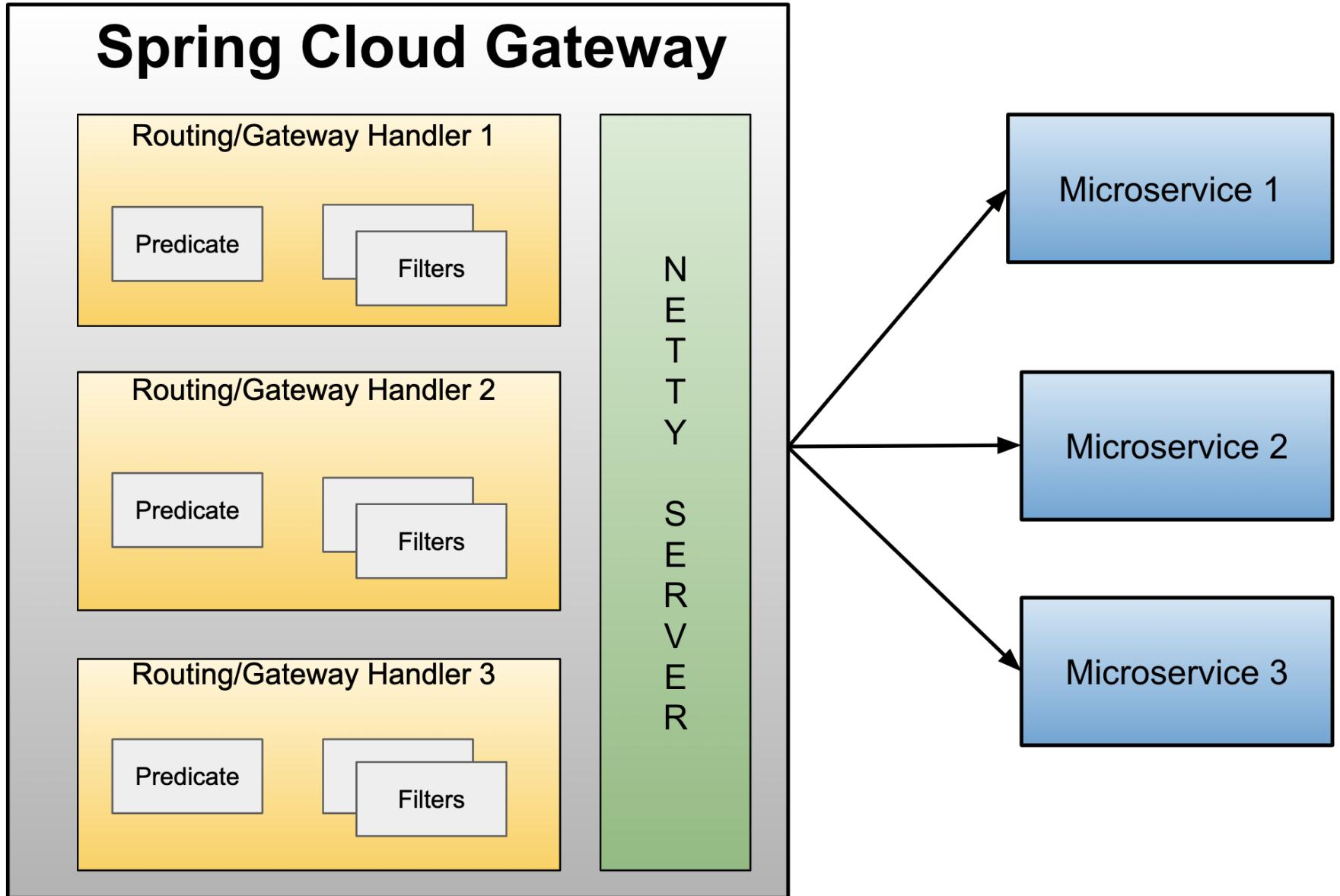
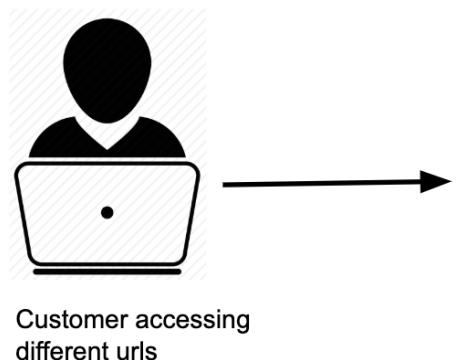
Spring Cloud Gateway

Spring Cloud Gateway is API Gateway implementation by Spring Cloud team on top of Spring reactive ecosystem.

It provides a simple and effective way to route incoming requests to the appropriate destination using Gateway Handler Mapping.

And Spring Cloud Gateway uses Netty server to provide non-blocking asynchronous request processing.

Spring Cloud Gateway



Spring Cloud Gateway consists of 3 main building blocks:

Route: Think of this as the destination that we want a particular request to route to. It comprises of destination URI, a condition that has to satisfy — Or in terms of technical terms, Predicates, and one or more filters.

Predicate: This is literally a condition to match. i.e. kind of “if” condition. If requests has something

— e.g. path=blah or request header contains foo-bar etc. In technical terms, it is Java 8 Function Predicate

Filter: These are instances of Spring Framework WebFilter.

This is where you can apply your magic of modifying request or response.

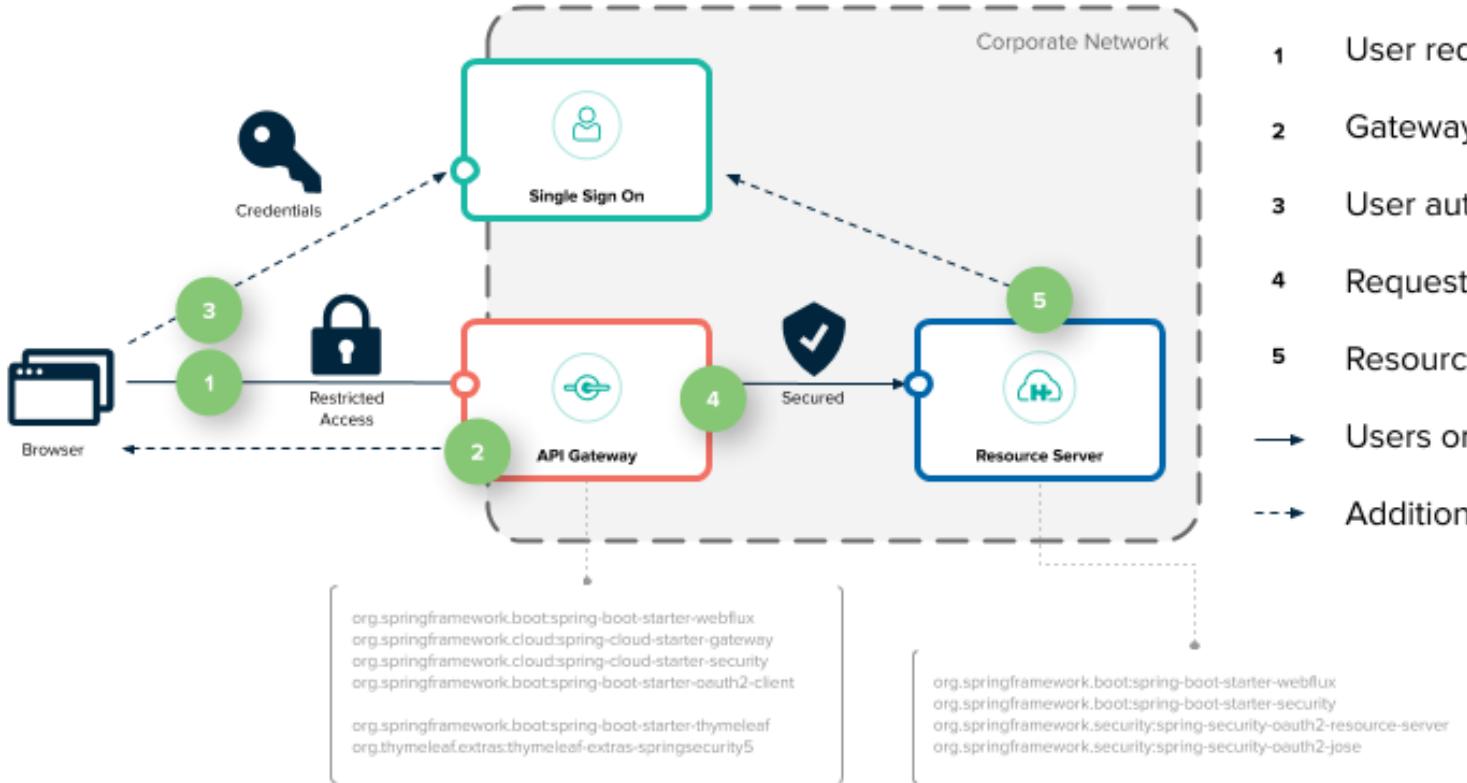
There are quite a lot of out of box WebFilter that framework provides

There are 2 different types of filters.

Pre Filters — if you want to add or change request object before we pass it down to destination service, you can use these filters.

Post Filters — if you want to add or change response object before we pass it back to client, you can use these filters.

Securing Services with Spring Cloud Gateway



- 1 User requests **/resource**
 - 2 Gateway redirects browser to SSO
 - 3 User authenticates & authorises the app
 - 4 Request & JWT sent to the Resource Server
 - 5 Resource Server checks user's JWT token
- Users original intended path
- ↔ Additional OAuth / JOSE flows

OAUTH 2.0 OVERVIEW

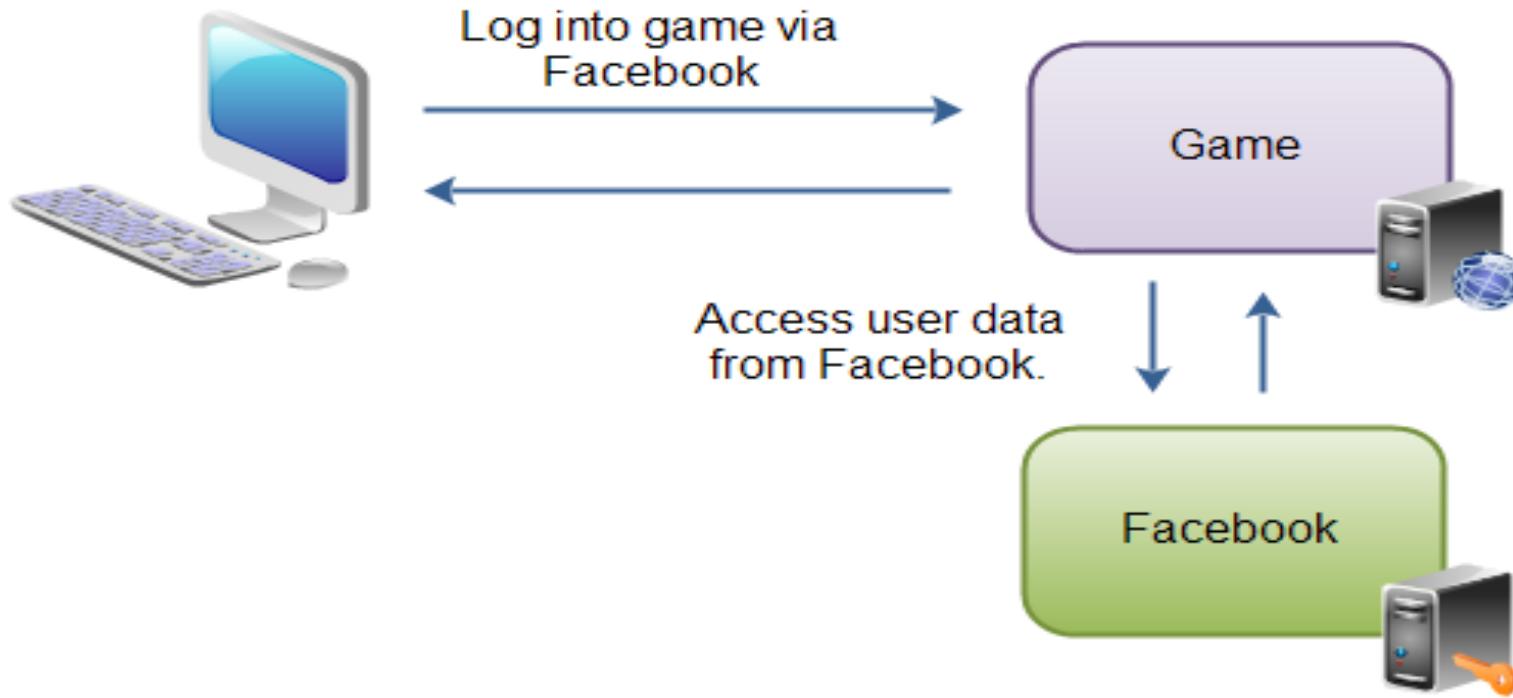
OAuth is an open standard for access delegation, commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords.

This mechanism is used by companies such as Amazon, Google, Facebook, Microsoft and Twitter to permit the users to share information about their accounts with third party applications or websites.

OAuth 2.0 is an open authorization protocol specification defined by IETF OAuth WG (Working Group) which enables applications to access each other's data.

The prime focus of this protocol is to define a standard where an application, say gaming site, can access the user's data maintained by another application like facebook, google or other resource server.

OAuth 2.0 is a replacement for OAuth 1.0, which was more complicated. OAuth 1.0 involved certificates etc. OAuth 2.0 is more simple. It requires no certificates at all, just SSL / TLS.



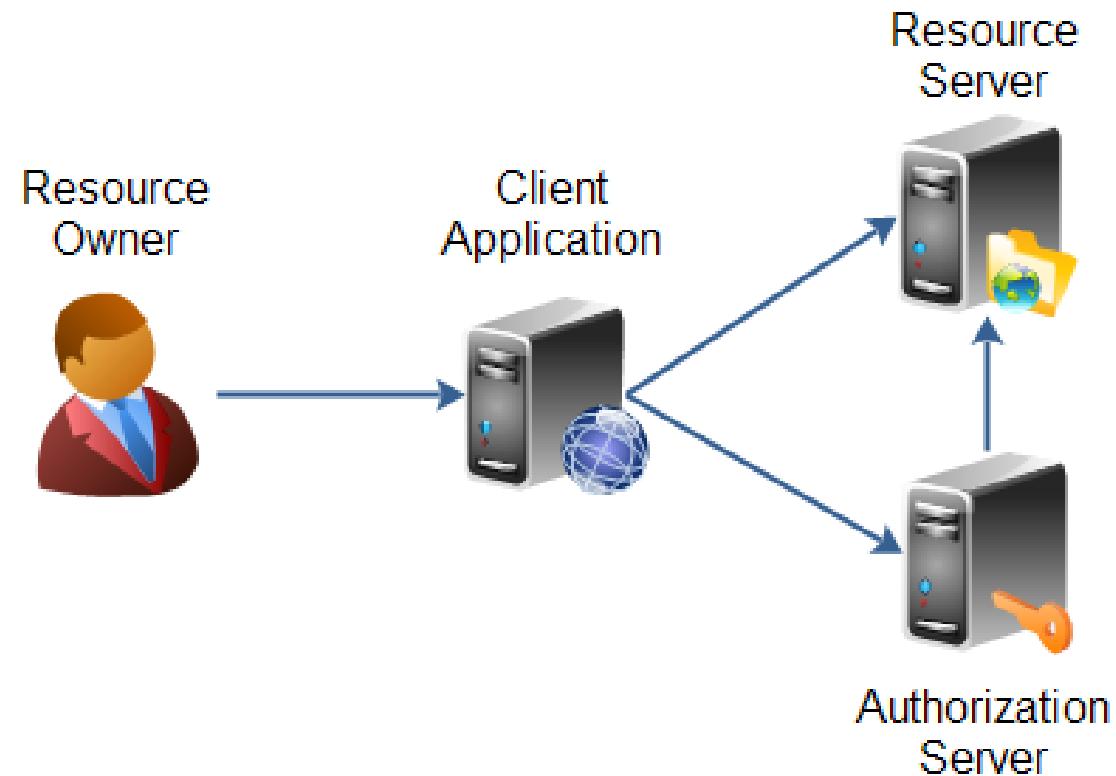
The user accesses the game web application.

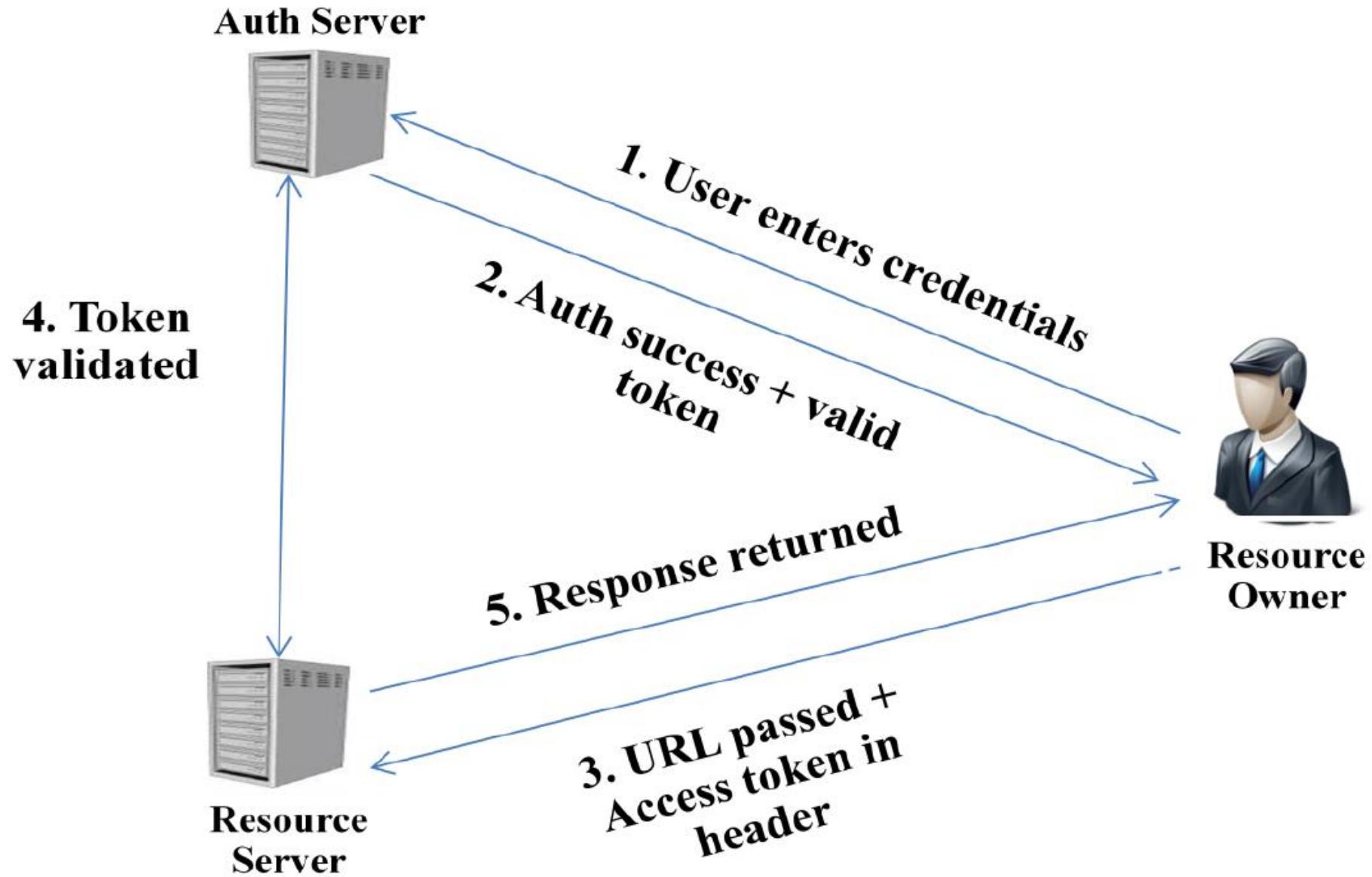
The game web application asks the user to login to the game via Facebook.

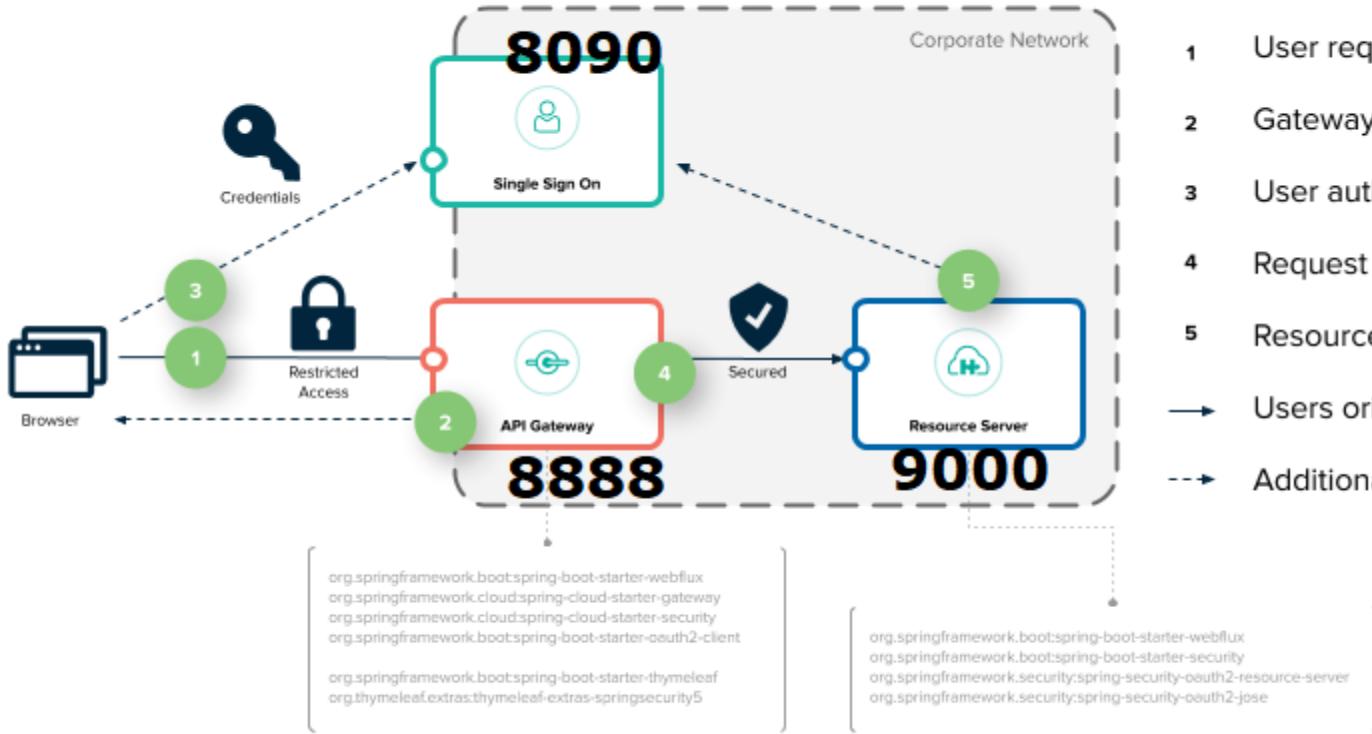
The user logs into Facebook, and is sent back to the game. The game can now access the users data in Facebook, and call functions in Facebook on behalf of the user (e.g. posting status updates).

OAuth 2.0 defines the following roles of users and applications:

- Resource Owner
- Resource Server
- Client Application
- Authorization Server







- 1 User requests **/resource**
 - 2 Gateway redirects browser to SSO
 - 3 User authenticates & authorises the app
 - 4 Request & JWT sent to the Resource Server
 - 5 Resource Server checks user's JWT token
- Users original intended path
- > Additional OAuth / JOSE flows

- ❑ The resource owner is the person or application that owns the data that is to be shared.

For instance, a user on Facebook or Google could be a resource owner. The resource they own is their data. The resource owner could also be an application. The OAuth 2.0 specification mentions both possibilities.

- ❑ The resource server is the server hosting the resources. For instance, Facebook or Google is a resource server (or has a resource server).

- ❑ The client application is the application requesting access to the resources stored on the resource server. The resources, which are owned by the resource owner. A client application could be a game requesting access to a users Facebook account.
- ❑ The authorization server is the server authorizing the client app to access the resources of the resource owner. The authorization server and the resource server can be the same server, but it doesn't have to.

Steps involved in User Authentication

1. User enters credentials which are passed over to Authorization Server in Http Authentication header in encrypted form. The communication channel is secured with SSL.
2. Authorization server authenticates the user with the credentials passed and generates a token for limited time and finally returns it in response.
3. The client application calls API to resource server, passing the token in http header or as a query string.
4. Resource server extracts the token and authorizes it with Authorization server.
5. Once the authorization is successful, a valid response is sent to the caller.

Client ID, Client Secret and Redirect URI

Before a client application can request access to resources on a resource server, the client application must first register with the authorization server associated with the resource server.

The registration is typically a one-time task. Once registered, the registration remains valid, unless the client app registration is revoked.

At registration the client application is assigned a client ID and a client secret (password) by the authorization server. The client ID and secret is unique to the client application on that authorization server.

Authorization Grant

The authorization grant is given to a client application by the resource owner, in cooperation with the authorization server associated with the resource server.

The OAuth 2.0 specification lists four different types of authorization grants. Each type has different security characteristics. The authorization grant types are:

- Authorization Code
- Implicit
- Resource Owner Password Credentials
- Client Credentials

Authorization Code

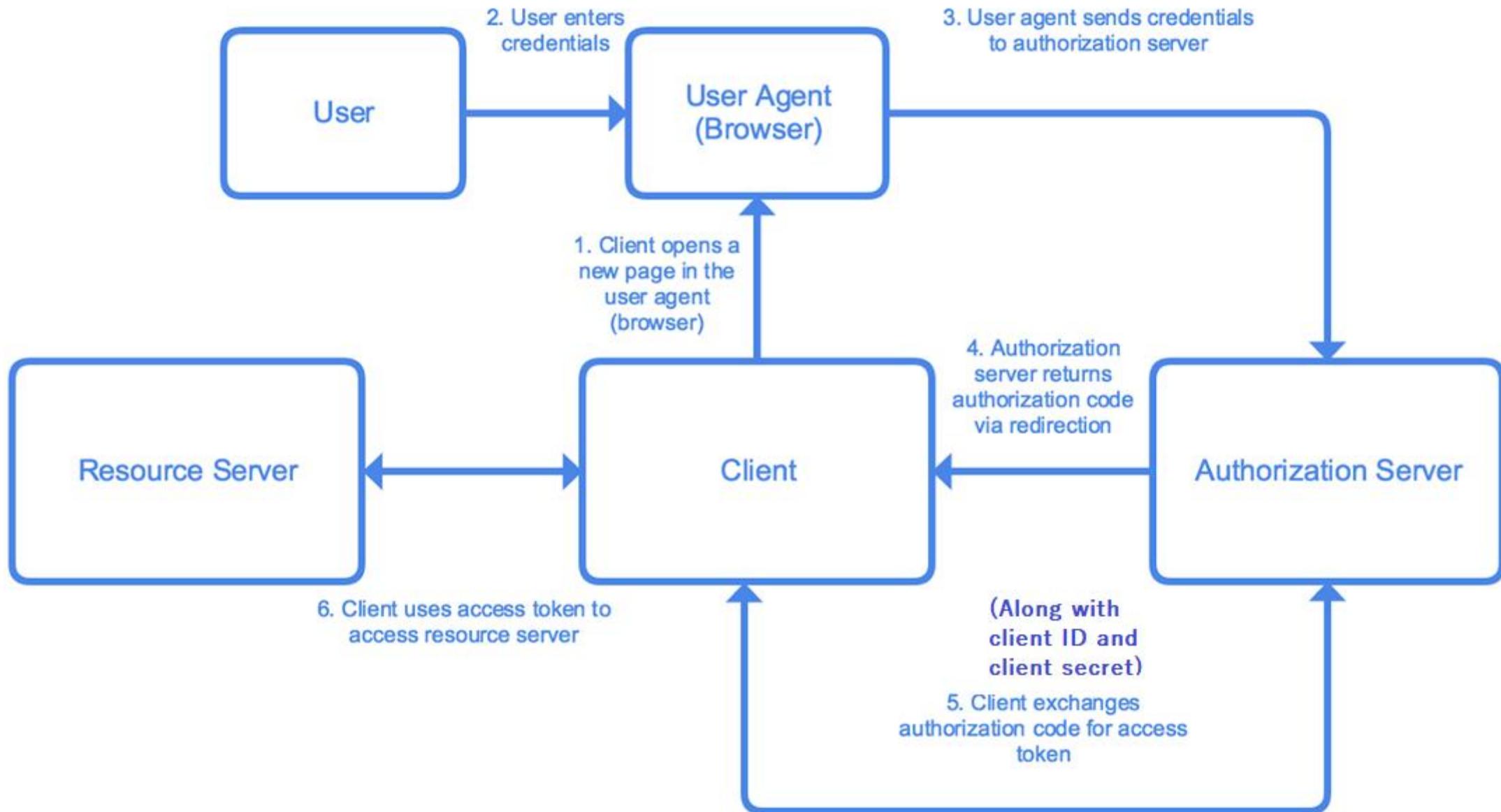
An authorization grant using an authorization code works like this

- 1) The resource owner (user) accesses the client application.
- 2) The client application tells the user to login to the client application via an authorization server (e.g. Facebook, Twitter, Google etc.).
- 3) To login via the authorization server, the user is redirected to the authorization server by the client application. The client application sends its client ID along to the authorization server, so the authorization server knows which application is trying to access the protected resources.



- 4) The user logs in via the authorization server. After successful login the user is asked if he wants to grant access to his resources to the client application. If the user accepts, the user is redirected back to the client application.
- 5) When redirected back to the client application, the authorization server sends the user to a specific redirect URI, which the client application has registered with the authorization server ahead of time. Along with the redirection, the authorization server sends an authorization code, representing the authorization.

- 6) When the redirect URI in the client application is accessed, the client application connects directly to the authorization server. The client application sends the authorization code along with its own client ID and client secret.
- 7) If the authorization server can accept these values, the authorization server sends back an access token.
- 10) The client application can now use the access token to request resources from the resource server. The access token serves as both authentication of the client, resource owner (user) and authorization to access the resources.



Implicit

An implicit authorization grant is similar to an authorization code grant, except the access token is returned to the client application already after the user has finished the authorization. The access token is thus returned when the user agent is redirected to the redirect URI.

This of course means that the access token is accessible in the user agent, or native application participating in the implicit authorization grant. The access token is not stored securely on a web server.



The client application can only send its client ID to the authorization server. If the client were to send its client secret too, the client secret would have to be stored in the user agent or native application too. That would make it vulnerable to hacking.

Implicit authorization grant is mostly used in a user agent or native client application. The user agent or native application would receive the access token from the authorization server.

Implicit Flow

Use Case: Browser

Client Secret: Confidentiality can not be guaranteed

Threats Implicit Flow

Resource owners might issue a token to a malicious client
(e.g. via phishing)

Attackers might steal token via other mechanisms

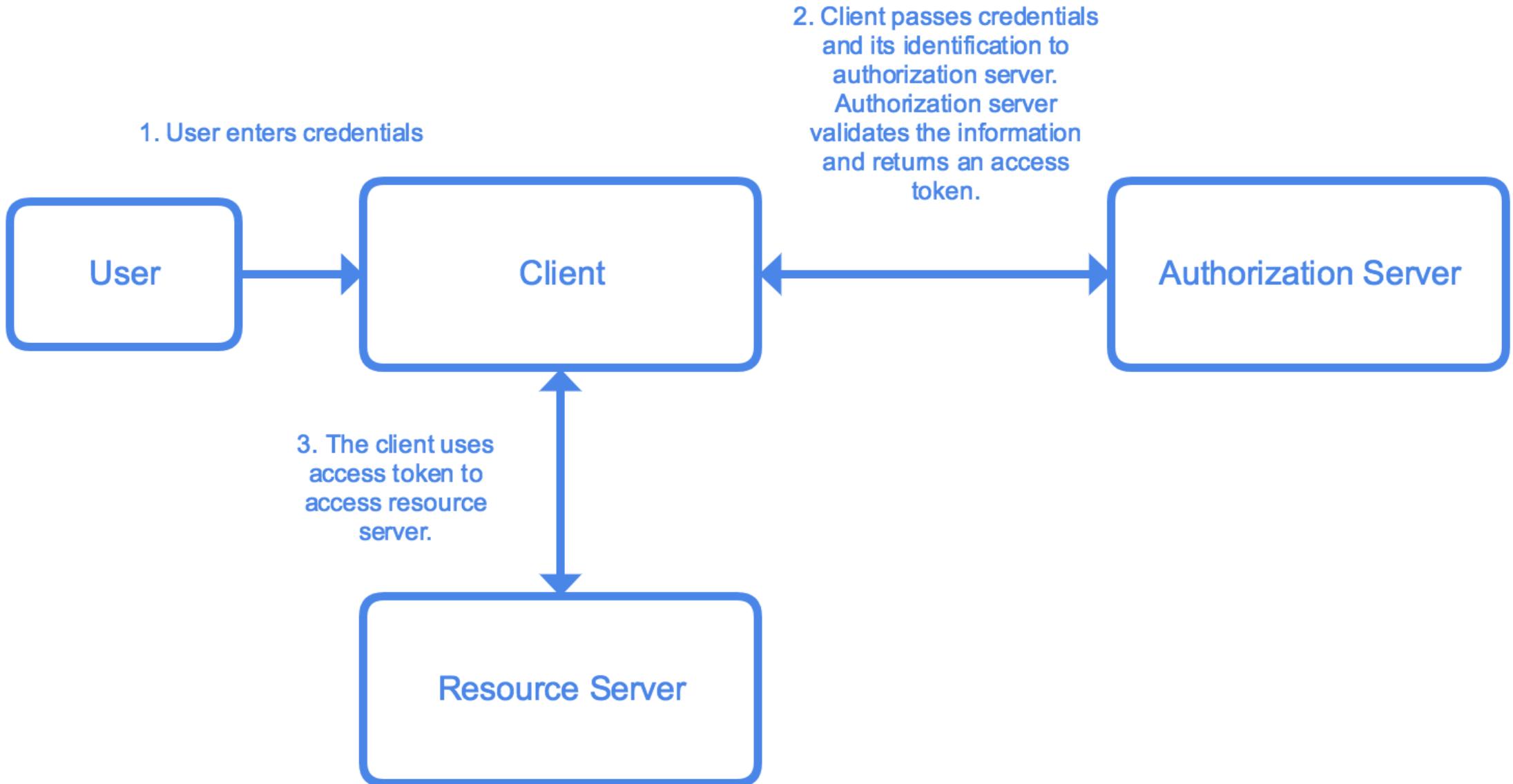
Resource Owner Password Credentials

The resource owner password credentials authorization grant method works by giving the client application access to the resource owners credentials. For instance, a user could type his Twitter user name and password (credentials) into the client application. The client application could then use the user name and password to access resources in Twitter.

Using the resource owner password credentials requires a lot of trust in the client application. You do not want to type your credentials into an application you suspect might abuse it.

The resource owner password credentials would normally be used by user agent client applications, or native client applications.

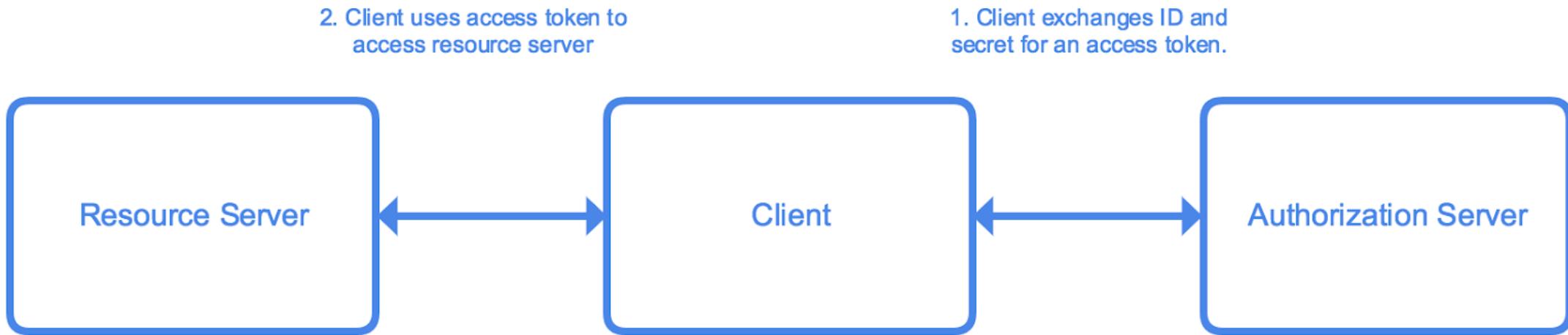
EX: Device Operating System



Client Credentials

Client credential authorization is for the situations where the client application needs to access resources or call functions in the resource server, which are not related to a specific resource owner (e.g. user).

For instance, obtaining a list of venues from Foursquare. This does not necessarily have anything to do with a specific Foursquare user.



Implicit and Client Credentials are flows typically reserved for special types of clients. More specifically,

Client Type	Flow
Single-page Javascript Web Applications (for example, Google Fonts)	Implicit
Non-interactive programs for machine-to-machine communications (for example, background services and daemons)	Client Credentials

As for other clients, depending on their trustworthiness, they can use the following flows:

Client Type	Flow
Highly trusted apps (first-party apps)	Authorization Code or Resource Owner Password Credentials
Less trusted apps (third-party apps requesting access to your platform)	Authorization Code

OAuth 2.0 Authorization Code Requests and Responses

The authorization code grant consists of 2 requests and 2 responses in total.

An authorization request + response, and a token request + response.

Authorization Request

The authorization request is sent to the authorization endpoint to obtain an authorization code. Here are the parameters used in the request

response_type	Required. Must be set to code
client_id	Required. The client identifier as assigned by the authorization server, when the client was registered.
redirect_uri	Optional. The redirect URI registered by the client.
scope	Optional. The possible scope of the request.
state	Optional (recommended). Any client state that needs to be passed on to the client request URI.

Authorization Response

The authorization response contains the authorization code needed to obtain an access token.

code	Required. The authorization code.
state	Required, if present in request. The same value as sent by the client in the state parameter, if any.

Authorization Error Response

If an error occurs during authorization, two situations can occur.

The first is, that the client is not authenticated or recognized. For instance, a wrong redirect URI was sent in the request. In that case the authorization server must not redirect the resource owner to the redirect URI. Instead it should inform the resource owner of the error.

The second situation is that client is authenticated correctly, but that something else failed.

error	Required. Must be one of a set of predefined error codes. See the specification for the codes and their meaning.
error_description	Optional. A human-readable UTF-8 encoded text describing the error. Intended for a developer, not an end user.
error_uri	Optional. A URI pointing to a human-readable web page with information about the error.
state	Required, if present in authorization request. The same value as sent in the state parameter in the request.

Token Request

Once an authorization code is obtained, the client can use that code to obtain an access token. Here is the access token request parameters:

client_id	Required. The client application's id.
client_secret	Required. The client application's client secret .
grant_type	Required. Must be set to authorization_code .
code	Required. The authorization code received by the authorization server.
redirect_uri	Required, if the request URI was included in the authorization request. Must be identical then.

Token Response

The response to the access token request is a JSON string containing the access token plus some more information:

```
{ "access_token": "...", "token_type": "...", "expires_in": "...", "refresh_token": "..." }
```

The `access_token` property is the access token as assigned by the authorization server.

The `token_type` property is a type of token assigned by the authorization server.

The `expires_in` property is a number of seconds after which the access token expires, and is no longer valid. Expiration of access tokens is optional.

The `refresh_token` property contains a refresh token in case the access token can expire. The refresh token is used to obtain a new access token once the one returned in this response is no longer valid.

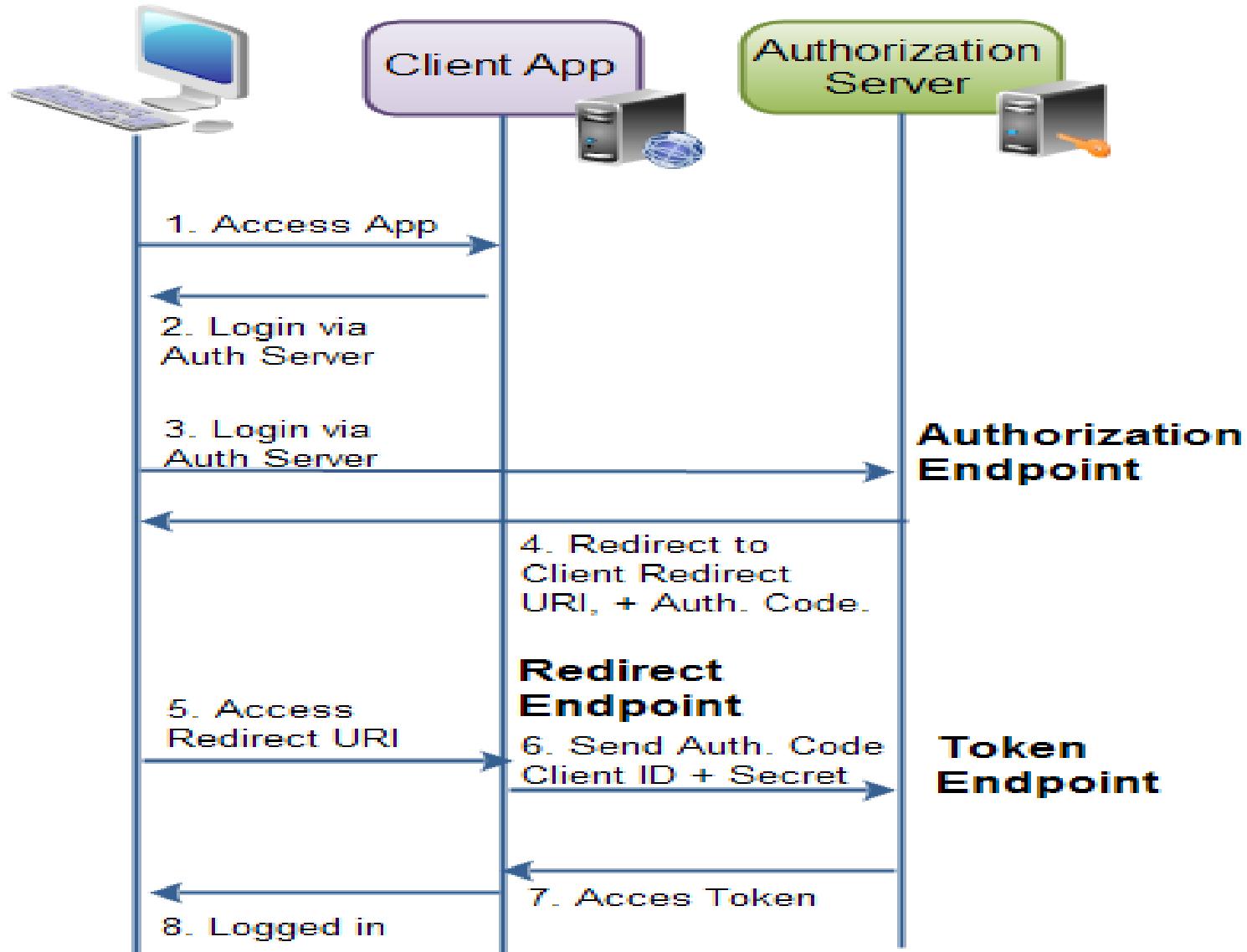
OAuth 2.0 Endpoints

OAuth 2.0 defines a set of endpoints. An endpoint is typically a URI on a web server. For instance, the address of a Java servlet, JSP page, PHP page, ASP.NET page etc.

The endpoints defined are:

- Authorization Endpoint
- Token Endpoint
- Redirection Endpoint

The authorization endpoint and token endpoint are both located on the authorization server. The redirection endpoint is located in the client application.



Authorization Endpoint

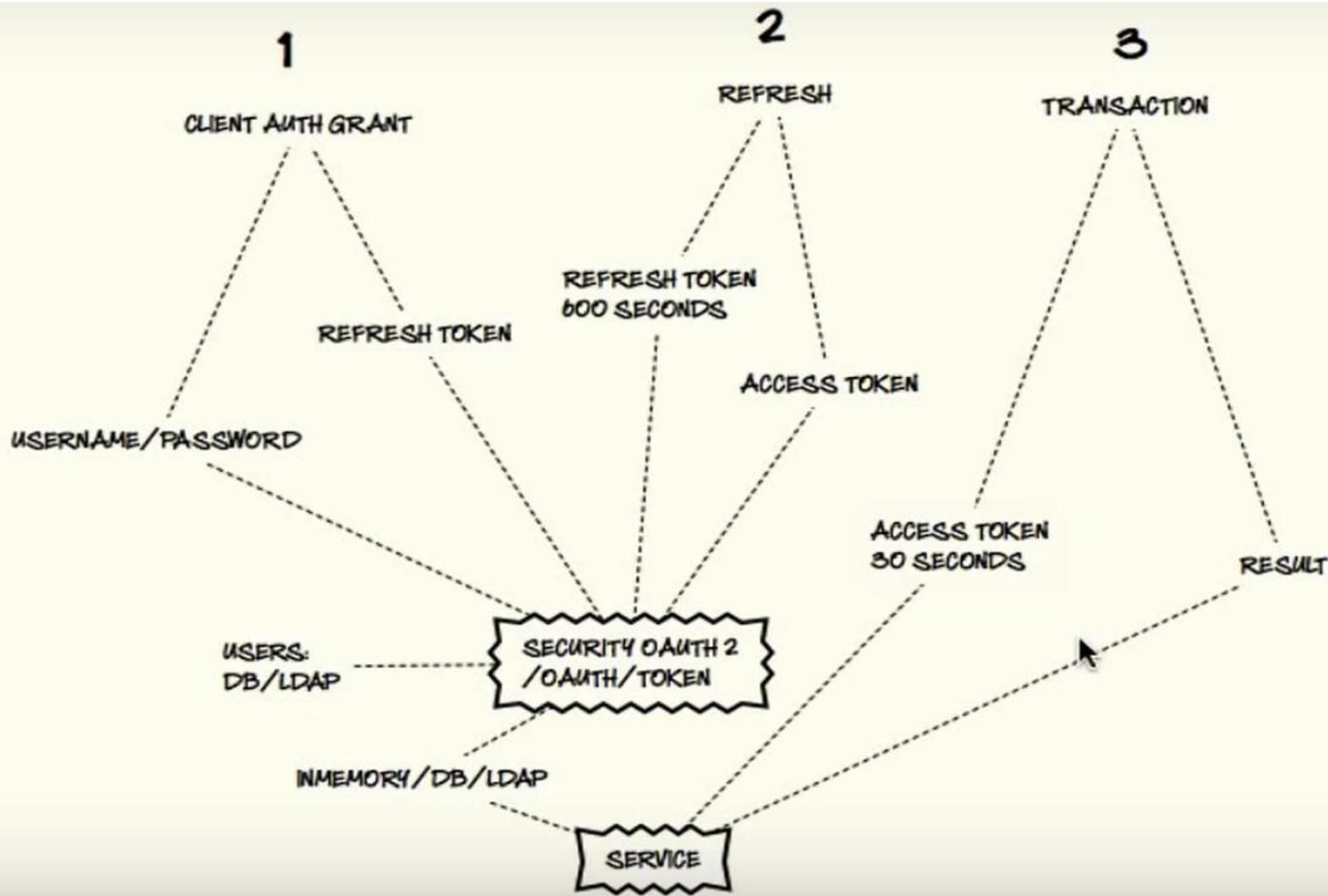
The authorization endpoint is the endpoint on the authorization server where the resource owner logs in, and grants authorization to the client application.

Token Endpoint

The token endpoint is the endpoint on the authorization server where the client application exchanges the authorization code, client ID and client secret, for an access token.

Redirect Endpoint

The redirect endpoint is the endpoint in the client application where the resource owner is redirected to, after having granted authorization at the authorization endpoint.



A **Refresh Token** is a special kind of token that can be used to obtain a renewed access token request of a new Access Token/Refresh Token pair

```
$ curl -X POST -H 'Authorization: Basic dGVzdGNsaWVudDpzZWNyZXQ=' -d  
'grant_type=password&username=test&password=test' localhost:3000/oauth/token
```

```
{  
  "token_type": "bearer",  
  "access_token": "eyJ0eXAiOiJKV1QiLCJhQ3_DYKxxP2rFnD37Ip4",  
  "expires_in": 20,  
  "refresh_token": "fdb8fdbecf1d03ce5e6125c067733c0d51de209c"  
}
```

We can use the Refresh Token to get a new Access Token by using the token endpoint

```
curl -X POST -H 'Authorization: Basic dGVzdGNsaWVudDpzZWNyZXQ=' -d  
'refresh_token=fdb8fdbecf1d03ce5e6125c067733c0d51de209c&grant_type=refresh_token'  
localhost:3000/oauth/token
```

grant_type=authorization_code

This is example of code request:

`http://localhost:8081/spring-security-oauth-server/oauth/authorize?response_type=code&client_id=myclient&redirect_uri=http://localhost:8080/auth`

And this is example of token request inside implementation of rest /auth

`http://localhost:8081/spring-security-oauth-server/oauth/token?client_id=myclient&client_secret=123&grant_type=authorization_code&code=byZc1r&redirect_uri=http://localhost:8080/auth`

"Understanding the Saga Pattern in Microservices Architecture
(Managing Long-Lived Transactions in Distributed Systems)

Introduction to Distributed Systems:

- Definition: A distributed system is a collection of autonomous computing elements that communicate and coordinate their actions through a network.
- Challenges: Long-lived transactions spanning multiple services, consistency, fault tolerance, scalability.

Need for a Solution:

- Traditional ACID transactions may not be suitable for distributed systems due to their rigid consistency requirements.
- Introduction to the saga pattern as an alternative approach.

Challenges with Traditional Transactions:

- ACID properties (Atomicity, Consistency, Isolation, Durability) are difficult to achieve in distributed environments.
- Long-lived transactions can lead to performance issues, scalability limitations, and increased complexity.

Need for an Alternative:

- An approach that allows for eventual consistency, fault tolerance, and scalability.
- Introduction to the saga pattern as a solution to these challenges.

Overview of Saga Pattern

Definition:

- The saga pattern is a design pattern for managing long-lived transactions in distributed systems.
- It decomposes a transaction into a series of smaller, more manageable steps.

Key Principles:

- Long-lived transactions: Transactions that span multiple services and can last for an extended period.
- Eventual consistency: Instead of ensuring immediate consistency, the system aims for consistency over time.
- Compensating actions: Actions executed to rollback or undo the effects of previously executed steps in case of failure.

Saga Execution

Saga Structure:

- A saga consists of a series of steps or operations performed by different services.
- Each step is encapsulated within its own local transaction boundary.

Coordination:

- Services communicate via events to coordinate the execution of saga steps.
- Events trigger the execution of subsequent steps and notify other services about the outcome of each step.

Compensating Actions

Importance:

- Compensating actions are essential for ensuring the consistency of the system in the face of failures.
- They are executed in response to step failures to rollback or undo the effects of previously executed steps.

Design Considerations:

- Compensating actions should be idempotent and designed to bring the system back to a consistent state.

Event-Driven Communication

Role of Events:

- Events play a crucial role in saga-based communication.
- They are used to trigger saga execution, notify other services about step outcomes, and initiate compensating actions.

Benefits:

- Decoupling of services: Services are loosely coupled, enabling independent development and scalability.
- Asynchronous communication: Enables non-blocking, asynchronous communication between services.

Orchestration vs. Choreography

Orchestration:

- In orchestration, a central coordinator service manages the execution of the saga.
- The coordinator orchestrates the execution of saga steps, communicates with services, and handles failures.

Choreography:

- In choreography, there is no central coordinator.
- Services communicate directly with each other via events, coordinating their actions without a single point of control.

Distributed Saga State Management

Challenges:

- Managing saga state across multiple services can be complex.
- Each service involved in the saga must maintain its local state and coordinate with other services to track the overall saga progress.

Solutions:

- Event-driven architecture: Events are used to propagate state changes and coordinate saga execution.
- Saga persistence: Saga state can be persisted using mechanisms such as event sourcing or distributed transaction logs.

Saga Persistence

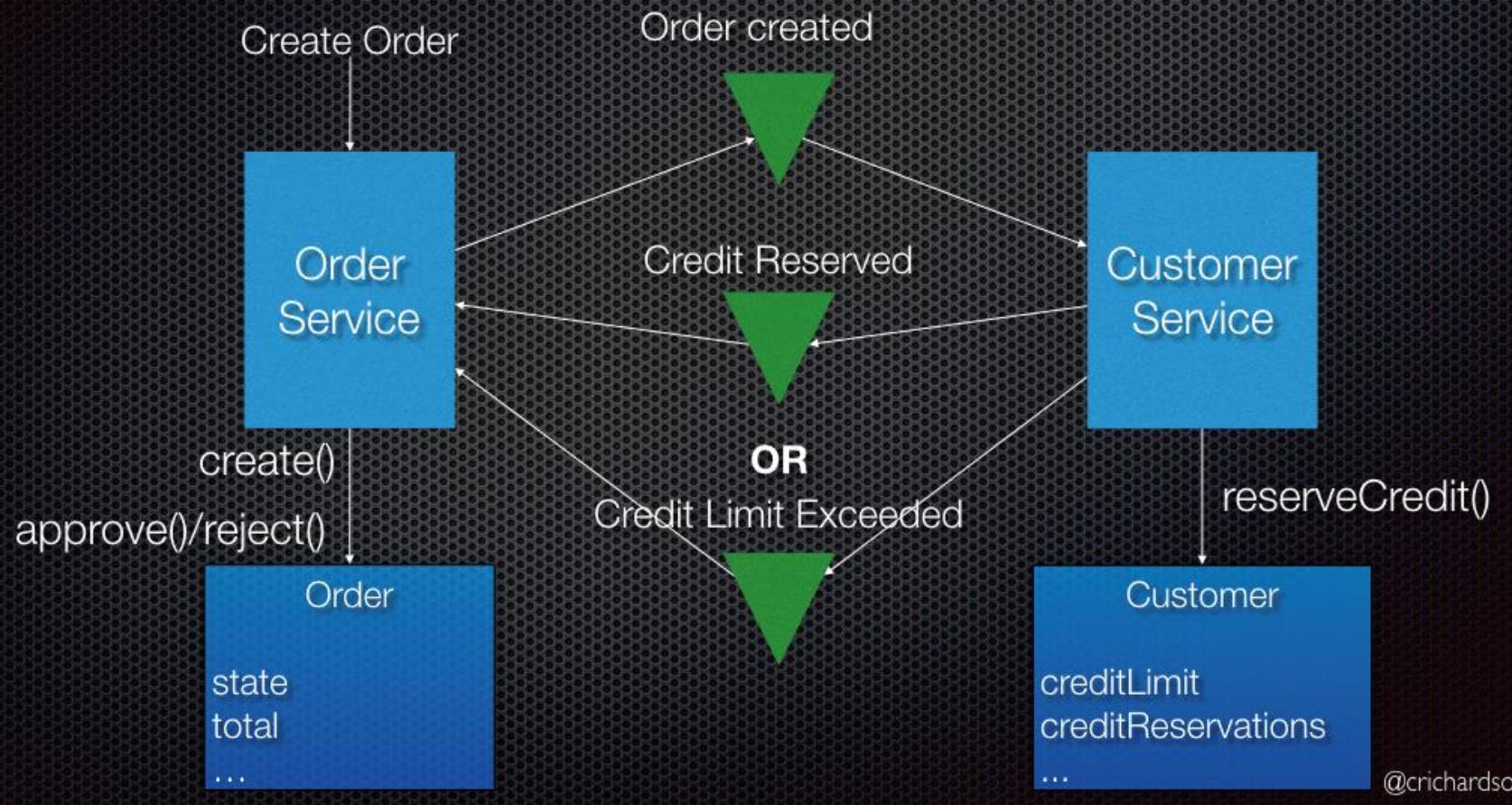
Need for Persistence:

- Saga persistence ensures that saga state is durable and can survive failures or system crashes.

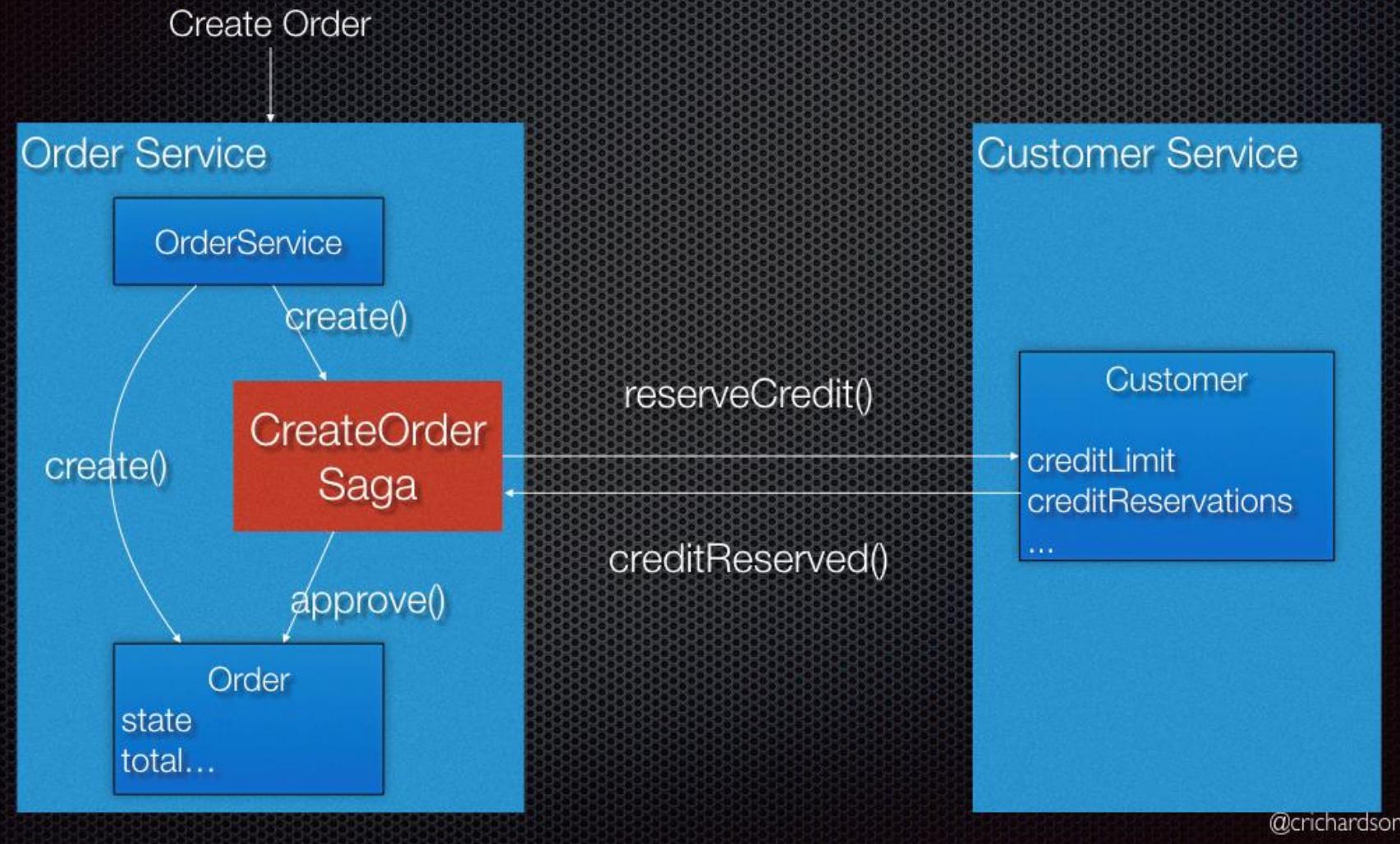
Persistence Mechanisms:

- Event sourcing: Storing the sequence of events that led to the current state of the saga.
- Distributed transaction logs: Logging saga state changes to ensure durability and fault tolerance.

Option #1: Choreography-based coordination using events



CreateOrderSaga orchestrator



Domain Driven Design

One of the biggest challenges of microservices is to define the boundaries of individual services.

The general rule is that a service should do "one thing" — but putting that rule into practice requires careful thought.

Microservices should be designed around business capabilities, not horizontal layers such as data access or messaging. In addition, they should have loose coupling and high functional cohesion.

Microservices are loosely coupled if you can change one service without requiring other services to be updated at the same time

A microservice is cohesive if it has a single, well-defined purpose, such as managing user accounts or tracking delivery history.

A service should encapsulate domain knowledge and abstract that knowledge from clients.

For example, a client should be able to schedule a drone without knowing the details of the scheduling algorithm or how the drone fleet is managed.

Domain-driven design (DDD) provides a framework that can get you most of the way to a set of well-designed microservices. DDD has two distinct phases, strategic and tactical.

In strategic DDD, you are defining the large-scale structure of the system. Strategic DDD helps to ensure that your architecture remains focused on business capabilities.

Tactical DDD provides a set of design patterns that you can use to create the domain model. These patterns include entities, aggregates, and domain services. These tactical patterns will help you to design microservices that are both loosely coupled and cohesive.

Understanding the Domain:

DDD focuses on understanding the problem domain and capturing its intricacies in a shared language and model. This understanding is fundamental to designing effective software solutions.

By starting with DDD, you gain insights into the domain's complexities, business rules, and key concepts, which serve as a solid foundation for architectural decisions.

Bounded Contexts and Sub-Domains:

DDD helps identify bounded contexts and sub-domains within the problem space, each with its own distinct models and language.

By delineating bounded contexts, you can define clear boundaries for your system components, which later align with architectural boundaries in Clean or Hexagonal Architecture.

Strategic Design Patterns:

DDD provides strategic design patterns like Bounded Context, Ubiquitous Language, and Context Mapping, which guide the organization and integration of domain models.

These patterns influence high-level architectural decisions and help ensure that architectural choices align with business requirements.

Tactical Design Patterns:

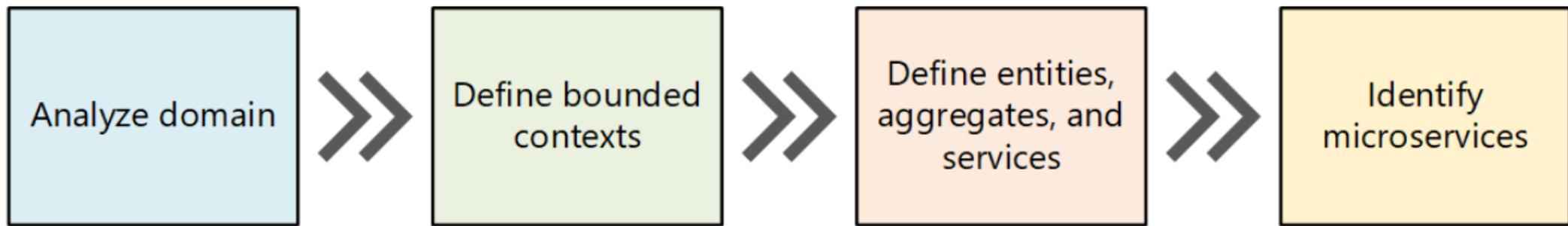
DDD offers tactical design patterns like Entity, Value Object, Aggregate, and Repository, which facilitate the implementation of domain logic within the application.

Understanding these patterns aids in designing the internal structure of components, which can then inform architectural decisions regarding layers, dependencies, and interactions.

Alignment with Business Needs:

Starting with DDD allows you to prioritize the alignment of your software with the business needs and objectives.

By focusing on the domain first, you ensure that your architectural choices are driven by business requirements and domain knowledge rather than technical considerations alone.



Domain Driven Design

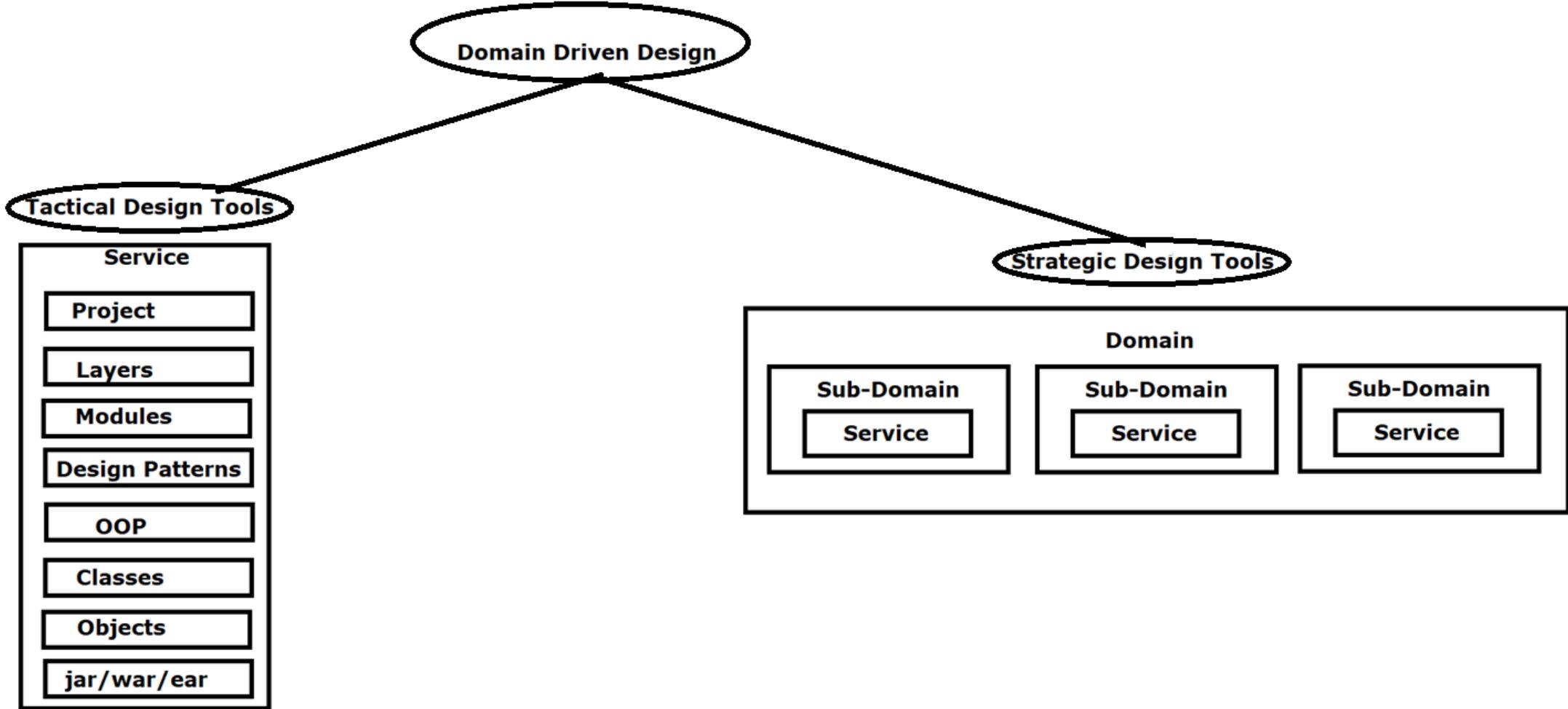
- It is a way of looking the software from top to down.
- When we are **developing a software our focus should not be** primarily on technology, it should be primarily **on business** or whatever activity we are trying to assist with the software, the domain.
- Specifically we approach that by trying to develop models of that domain and make our software conformed to that.

For most software projects, the primary focus should be on the domain and domain logic

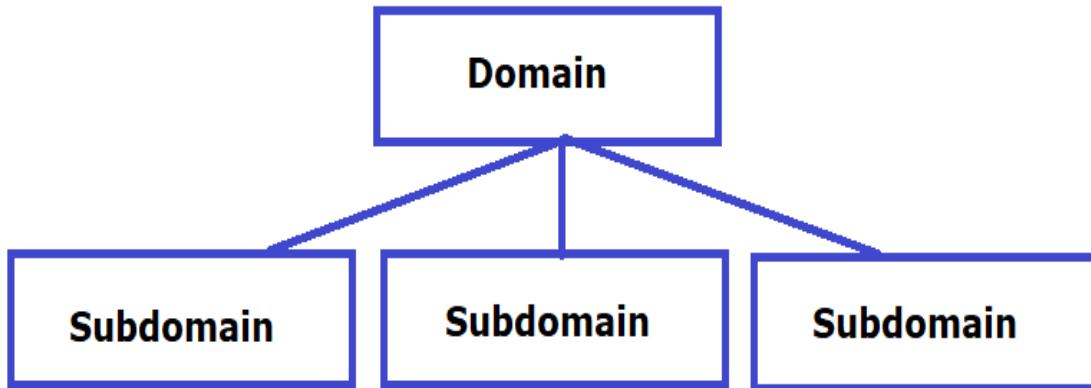
Domain-driven design is an approach to software development for complex needs by connecting the implementation to an evolving model.

Domain-driven design is predicated on the following goals:

- ❑ placing the project's primary **focus on the core domain and domain logic**;
- ❑ basing complex **designs on a model of the domain**;
- ❑ initiating a creative **collaboration between technical and domain experts** to iteratively refine a conceptual model that addresses particular domain problems.



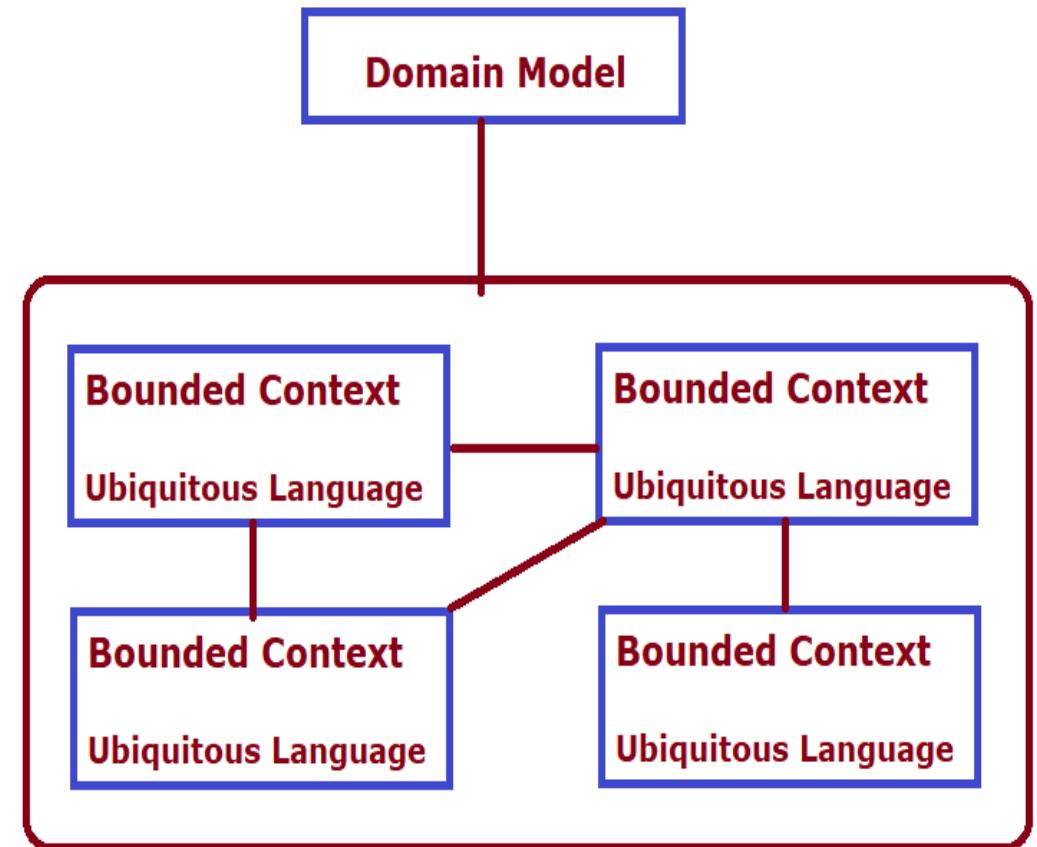
Problem Space



Types of Subdomain:

- Core Sub-Domain
- Supporting Sub-Domain
- Generic Subdomains

Solution Space



Object Oriented Design : Think in terms of Objects

Strategic Design : Think in terms of Contexts

Core Domain

The core domain is so **critical and fundamental to the business** that it gives you a competitive advantage and is a foundational concept behind the business.

Ex : Order processing, loan processing, payment service etc.,

Supporting Subdomain

These types of pieces are also necessary because they **help perform ancillary or, well, supporting functions** related directly to what the business does.

Ex : system manages all the banners, landing pages, and other creative materials produced by our design studio

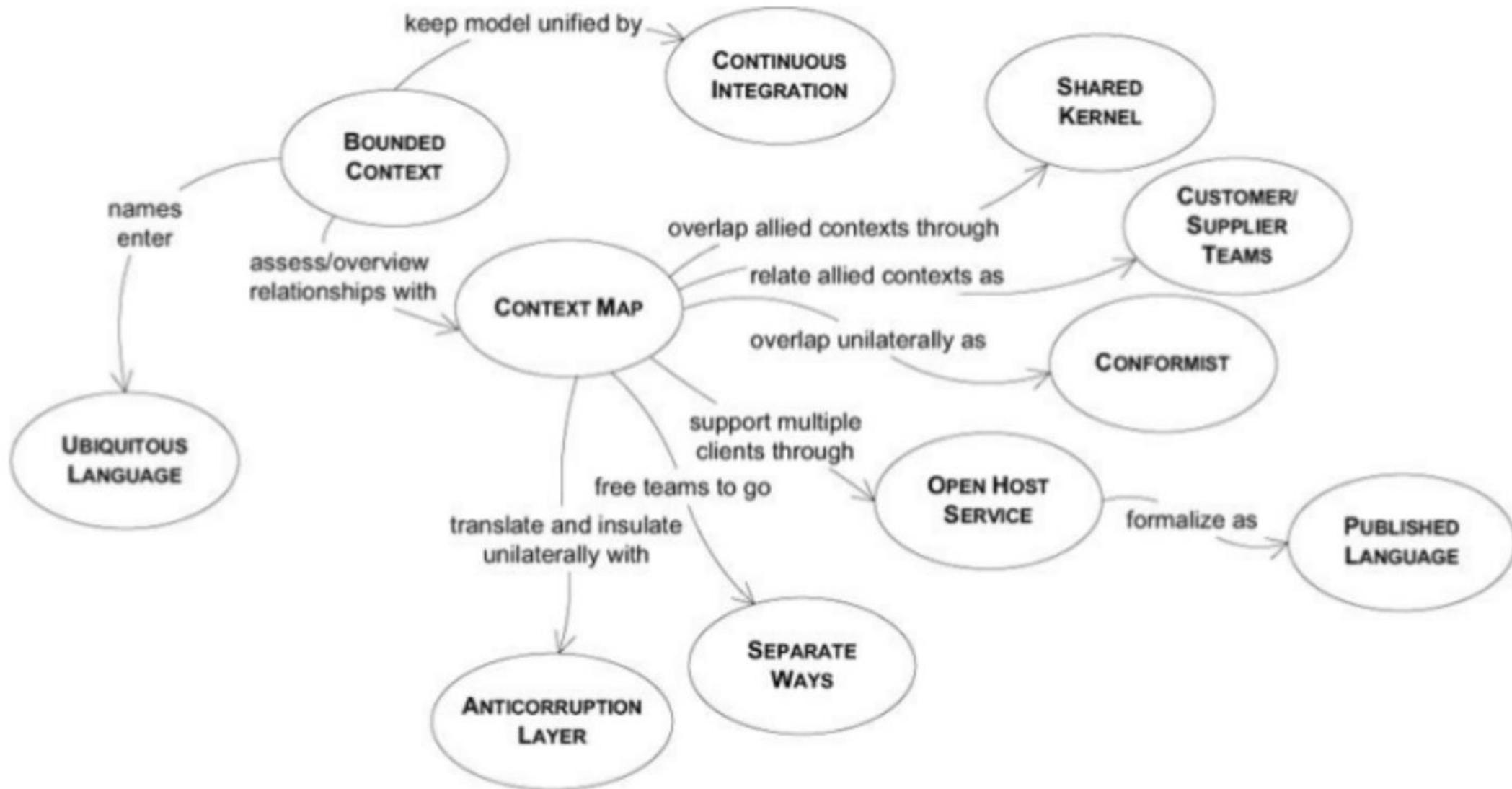
Generic Subdomains

Generic subdomains are those things that all companies do the same way. Those **are considered “solved” problems.**

Ex : identity and access management, ERP module , Invoicing & reporting

Strategic Design

- It would be preferable to **have a single, unified model.**
- It is a **set of principles for maintaining model integrity.**

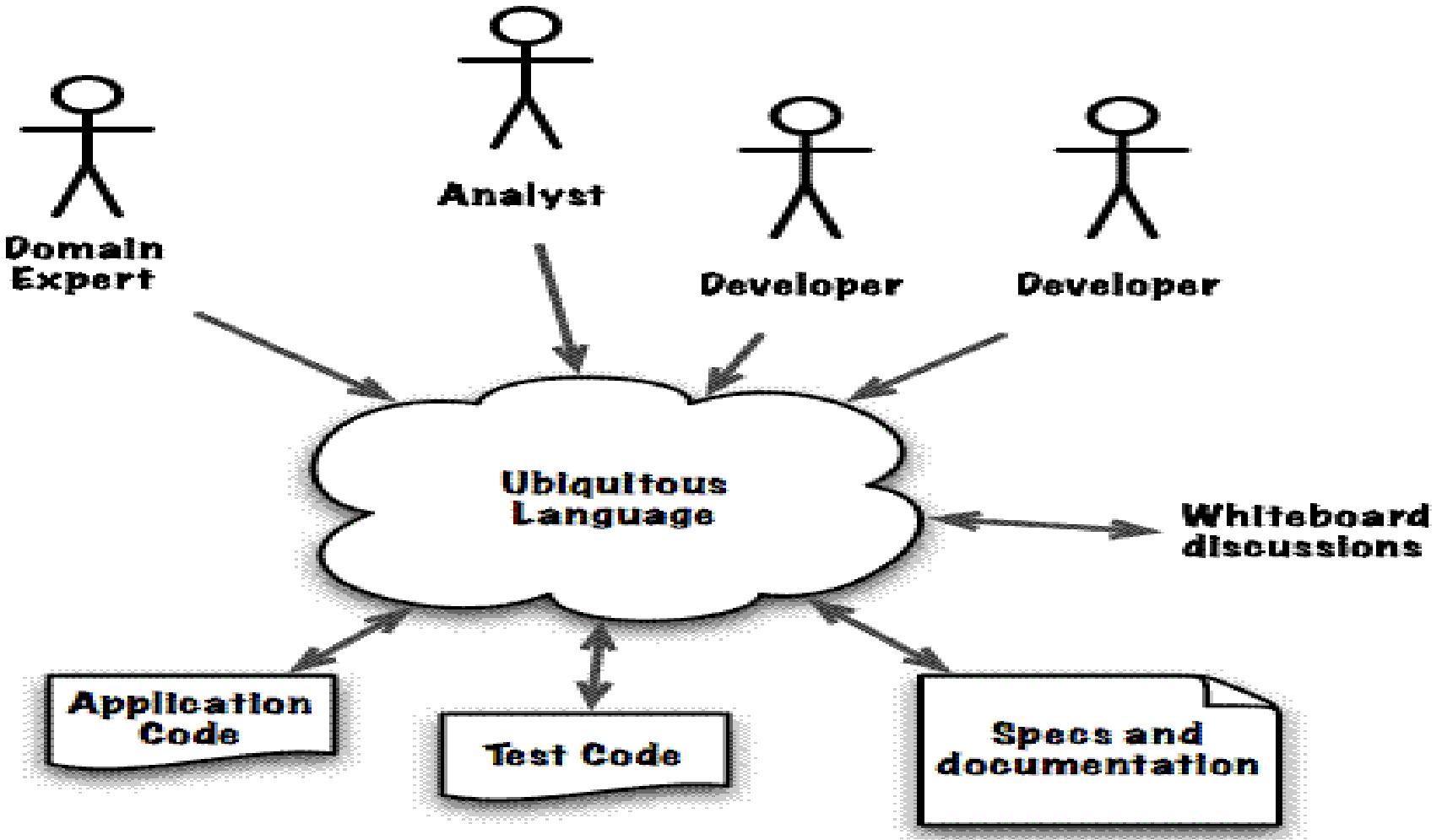


Strategic Design Patterns

The Ubiquitous Language

The Need for a Common Language

- ❑ **Developer** think in terms of **inheritance, composition** etc.
And they talk like that all the time.
- ❑ But the **Domain Experts** usually know nothing about any of that. They have no idea about software libraries, frameworks, persistence, in many case not even databases. They **know about their specific area of expertise.**
- ❑ It is absolutely necessary to **develop a model of the domain** by having the **software specialists work with the domain experts;**



The Ubiquitous Language should be the **only language** used to express a model. Everybody in the team should be able to agree on every specific term without ambiguities and no translation should be needed

Creating the Ubiquitous Language

How can we start building a language?

Here is a **hypothetical dialog between a software developer and a domain expert in the inventory management & Reorder Point Management Project**

Ubiquitous Language is a concept from Domain-Driven Design (DDD) that promotes a shared and consistent language between business stakeholders, developers, and all team members involved in a project. This helps ensure everyone has a common understanding of the domain and its intricacies. Here's a simplified conversation in the context of a sales domain, adhering to Ubiquitous Language principles:

Salesperson: Hey, I've been talking to the client, and they're interested in our product.

Product Manager: That's great news! What features are they specifically looking for?

Salesperson: They're really focused on the inventory management and reporting capabilities. They want real-time updates on stock levels.

Developer: So, you mean they're interested in the Inventory Control subdomain, right?

Salesperson: Yes, that's correct. They also mentioned wanting to set up automated reorder points.

Business Analyst: Sounds like they need a Reorder Point Management module. How frequently do they want the system to check stock levels?

Salesperson: They're looking for daily checks with email alerts when inventory drops below the reorder point.

Designer: Got it, so we'll need to design an email notification system. What kind of reports are they expecting?

Salesperson: They want monthly sales reports and real-time stock status dashboards.

Data Analyst: We'll need to create a Sales Reporting subdomain for those monthly sales reports. For real-time dashboards, we can integrate with the Inventory Control subdomain.

Product Manager: It seems like the client is mainly interested in the Inventory Control and Sales Reporting subdomains. Let's prioritize those for development.

In this conversation, the team members use a common language to discuss the sales domain, referring to specific subdomains, modules, and features.

This ensures that everyone understands the client's requirements and can work together effectively to deliver a solution that meets the client's needs.

This shared language is a fundamental aspect of Domain-Driven Design and helps prevent misunderstandings and miscommunication within the team.

The outcome of the conversation in the given context would typically be as follows:

Understanding Client Requirements: The Salesperson communicates the client's interest in the project, focusing on inventory management and reporting.

Identifying Specific Needs: The team, including the Product Manager, Developer, Business Analyst, Designer, and Data Analyst, engages to gather more specific information about the client's requirements.

Defining Subdomains: Through the conversation, it becomes clear that there are specific subdomains within the project, such as "Inventory Control" and "Sales Reporting."

Prioritization: The team discusses the client's priorities and decides to focus on developing features related to the "Inventory Control" and "Sales Reporting" subdomains.

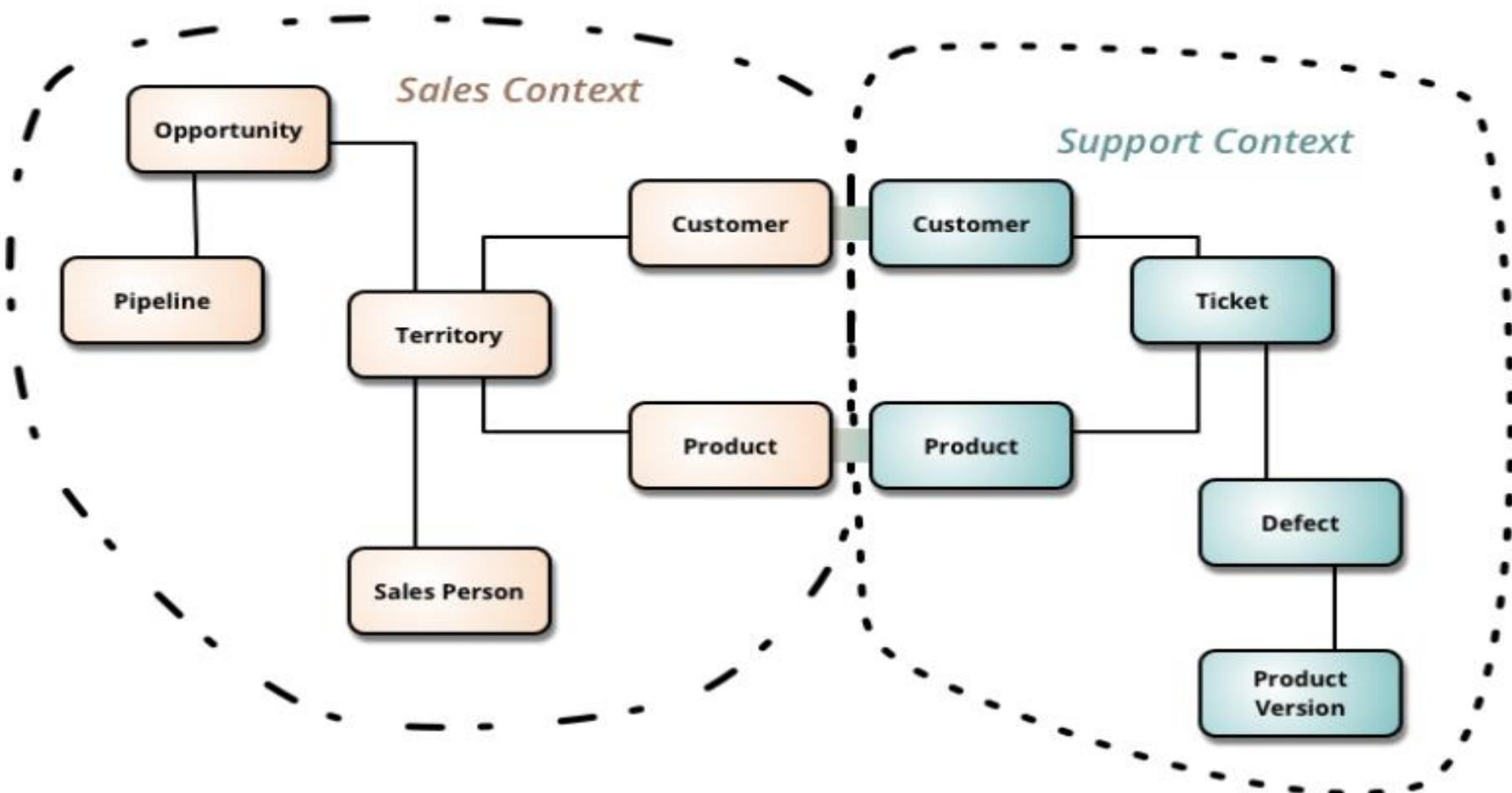
The outcome of this conversation is a shared understanding among team members regarding the client's needs, the specific focus areas of the project, and the initial steps to be taken, which may include further detailed requirements gathering, project planning, and development efforts. This shared understanding is crucial for effective collaboration and successful project delivery in the context of Domain-Driven Design and Ubiquitous Language.

Bounded Context

In DDD, a **subdomain in the problem space** is mapped to a **bounded context in the solution space**.

A **bounded context** is an area of the application that requires its **own ubiquitous language** and its own architecture. Or, put another way, a bounded context is a boundary within which the ubiquitous language is consistent. A **bounded context** can have **relationships to other bounded contexts**.

Bounded Context is a central pattern in Domain-Driven Design. It is the focus of DDD's strategic design section which is all about dealing with large models and teams. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.



Customer and Product Relationships:

•Sales Context:

- ✓ In the Sales context, the Customer entity may have relationships with products based on purchases or orders. For example:
 - ✓ One-to-Many Relationship: A customer can place multiple orders, each containing one or more products.
 - ✓ Many-to-Many Relationship: A product can be purchased by multiple customers, and a customer can purchase multiple products.
- ✓ These relationships are typically transactional and focused on sales activities, such as tracking orders, managing inventory, and calculating sales metrics.

Support Context:

In the Support context, the Customer entity may have relationships with products based on support interactions or issues. For example:

- ✓ Support Tickets: Each support ticket raised by a customer may be related to a specific product or category of products.
 - ✓ Service History: Tracking the support history of products for each customer, including resolution status, response times, etc.
- These relationships are more focused on support interactions, troubleshooting, and maintaining customer satisfaction.

Inter-Context Relationships:

- ✓ While entities within a bounded context are primarily related to each other based on the context's specific domain, there can be interactions or data flows between different contexts.
- ✓ For example, a support ticket raised by a customer in the Support context may be related to a specific order or product purchase made by the same customer in the Sales context. This cross-context relationship may require integration or data sharing mechanisms between the Sales and Support systems.

Contextual Integrity and Consistency:

- ✓ When dealing with relationships across contexts, it's essential to ensure data integrity and consistency. Changes in one context (e.g., order status update in Sales) may need to be reflected or propagated to related entities in other contexts (e.g., support tickets in Support).
- ✓ Design patterns such as Domain Events, Saga Patterns, or Distributed Transactions can be employed to manage data consistency and maintain contextual integrity across bounded contexts.

Microservices architecture focuses on breaking down a large application into smaller, independent services that can be developed, deployed, and scaled separately. Here are some microservices that might exist in the Support Context:

Customer Support Service:

- ✓ This microservice handles customer support interactions, such as creating and managing support tickets, tracking service requests, and resolving customer issues.
- ✓ It may expose APIs for creating tickets, updating ticket status, assigning support agents, and retrieving support history for customers.
- ✓ Example endpoints:
 - POST /support/tickets: Create a new support ticket.
 - PUT /support/tickets/{ticketId}: Update the status or details of a support ticket.
 - GET /support/customers/{customerId}/tickets: Get all support tickets for a specific customer.

Product Support Service:

- ✓ This microservice manages product-related support information, including troubleshooting guides, FAQs, known issues, and product documentation.
- ✓ It may provide APIs for retrieving product support information, updating knowledge base entries, and handling product-specific support requests.
- ✓ Example endpoints:
 - GET /support/products/{productId}/info: Get product support information and documentation.
 - POST /support/products/{productId}/issues: Report a new issue or bug related to a product.
 - PUT /support/products/{productId}/knowledge-base: Update knowledge base entries for a product.

Feedback and Survey Service:

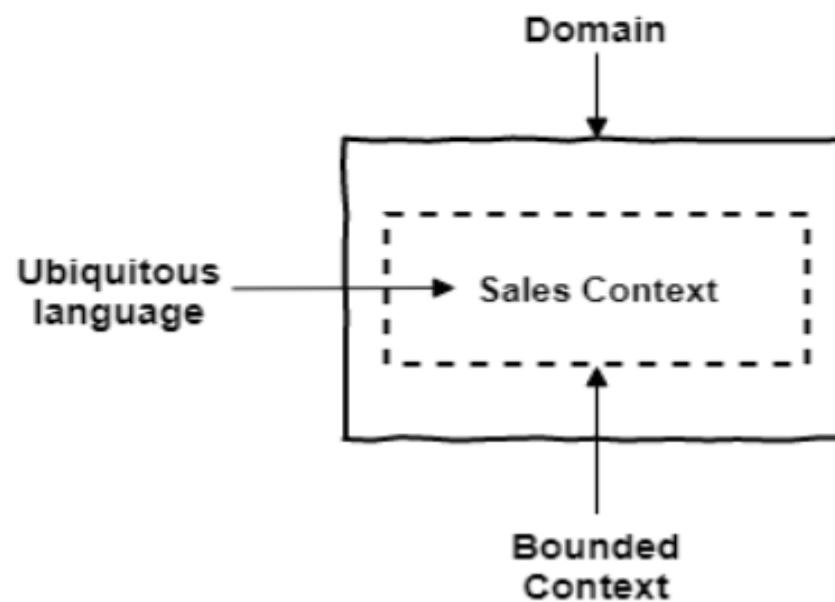
- ✓ This microservice handles customer feedback, surveys, and satisfaction metrics to gather insights and improve support services.
- ✓ It may integrate with external systems for conducting surveys, analyzing feedback sentiment, and generating reports or analytics.
- ✓ Example endpoints:
 - POST /support/feedback: Submit feedback or ratings from customers.
 - GET /support/surveys/{surveyId}/responses: Get survey responses and analytics data.
 - PUT /support/feedback/{feedbackId}: Update or respond to specific feedback entries.

Notification Service:

- ✓ This microservice manages notifications and alerts related to support activities, such as ticket updates, service status changes, or scheduled maintenance notifications.
- ✓ It may use messaging or notification APIs to send notifications via email, SMS, or in-app notifications.
- ✓ Example endpoints:
 - POST /support/notifications: Send a notification to customers or support agents.
 - GET /support/notifications/{notificationId}: Retrieve details of a specific notification.

- ✓ These microservices encapsulate specific functionalities and responsibilities related to customer support, product support, feedback management, and notifications within the Support Context.
- ✓ Each microservice can be developed, deployed, and scaled independently, promoting flexibility, maintainability, and agility in the overall system architecture.
- ✓ Integrating these microservices with other components in the system, such as the Sales Context or external systems, can be done through well-defined APIs and integration patterns.

Ubiquitous Language is modeled within a Limited context, where the terms and concepts of the business domain are identified, and there should be no ambiguity.

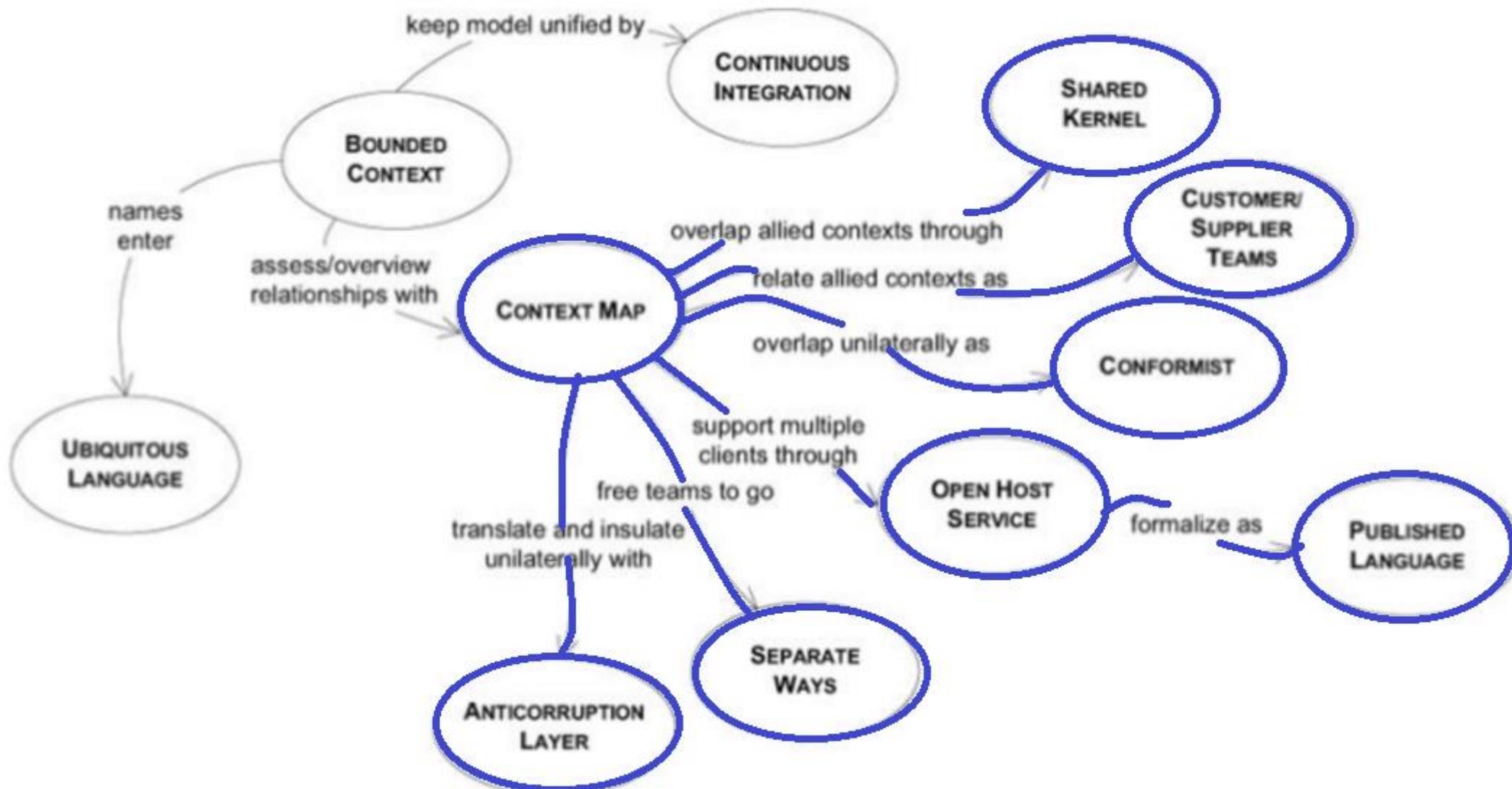


Bounded Context Communication: Any design has two common parts, abstraction of the data model and communicates with other parts of the system.

By Bounded context we separate the data model in a simple term abstracting the commonalities in the business but **How one Bounded context communicate with others?**

Here the concept of **Context Map** stepped in, Using Context map we can discover How one Context depends on other Bounded Context:

- ❑ Like are two Context has strong dependencies.
- ❑ or one domain sends a confirmation message to another domain(Conformist)
- ❑ or may use a shared kernel/Shared model,



Domain : Online Student management system

Online Student management system is where:

- Student can register to the site & choose course
- Pay the Course fee
- Student will be tagged to a batch
- Teacher & Student is notified about the batch details

We have to identify the bounded context of the different domain related to this business logic.

There are 4 bounded contexts Registration, Payments, Scheduler, and Notification.

- 1. Registration process:** Which takes care of Registration of Student.
- 2. Payment System:** Which will process the Course fee and publish online payment status.
- 3. Batch Scheduling:** Upon confirmation of payment, this function checks the Teacher availability, batch availability and based on that create a batch and assign the candidates or update an existing batch with the candidate.
- 4. Notification System:** It will notify Teacher and Student about the timings and slot information.

Shared Kernel :

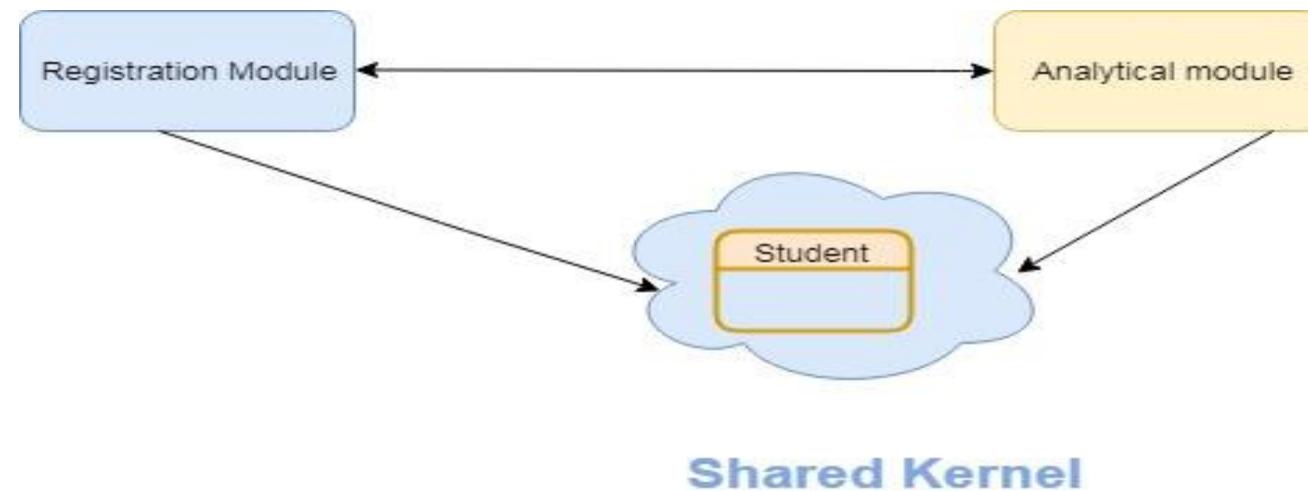
Shared kernel talks about a partnership relation where two or many teams shared common data model/ value object;

It reduces the code duplication as different context use that common model, but that common model/value object is very sensitive;

Any changes major/minor should be agreed upon all the parties unless it would break other parties code.

Example : Analytical module is used to analysis which courses are most chosen by the students, which students are chosen more than five courses etc, so that module works with Student model, course model.

Analytics module can share Registration modules student model, and they also agreed upon any changes on student model.



Customer/ Supplier:

Generally this is the common relationship between two contexts, where a context consumers or depend on data from another context, the context which produces data marked as upstream and the context which consumes data called downstream.

Upstream as the leader:

In this type of relationship, the upstream team is in a **commendable position**, that team does not care about the downstream team, as they producing the data and downstream team need to change their model based on the data structure produces by upstream.

Example : Payment application decide what information in which structure they provide and Notification module consumes that data structure.



Upstream as the Leader

Downstream as the Leader:

In some cases, the relationship is revert although upstream is produces data, it must have to follow the rule, the data structure for downstream, in this scenario downstream is in a **commendable position**.

Example : Student registration system we need to submit Form 16 to government as a tax payee so our payment module has to submit form 16 data to Government exposed API but government API has certain rules and data structure for submitting form 16 data, so although government API is downstream it has total control, our Payment module should communicate with down steam in such a way so it can fulfil downstream rules.



The customer-supplier relation works best when both the parties upstream and downstream are aligned with the work both party agreed upon the interfaces and change in the structure.

In case of any changes in the contract both parties will do a discussion synchronize their priority backlogs and agreed upon the changes, If one party does nor care upon another party then every time contract will be broken and it is tough to maintain a customer-supplier relationship.

Conformist:

Sometimes, there is a relation between two parties in such a way that downstream team always dependent on an upstream team and they **can't do a mutual agreement** with upstream about requirements.

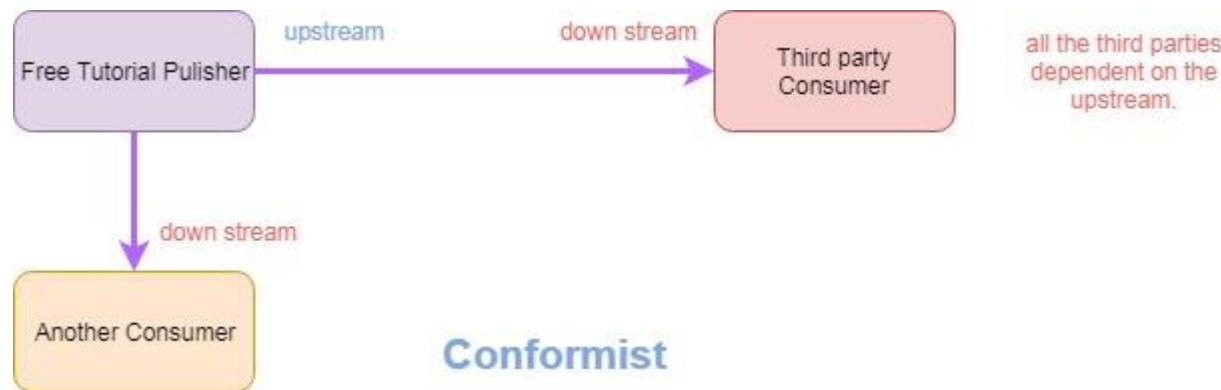
The **upstream** is not aligned with downstream and does not care they are **free to change their published endpoint** or contract any time and not taking any request from downstream.

It happens when **Upstream team is an external system** or under a different management hierarchy , and many downstream systems are registered with it so it can't give a priority to any downstream, rather then all downstream system must be aligned with upstream contact and data structures.

Example : Free tutorial module is used by all students or other applications can embed them in their application.

Here Free tutorial module acts as upstream and independent of any other third party app who consumes our free tutorials.

We can't give any priority to them and we don't have any contract with them if we change the contracts or data structures it is other third parties duty to change their application accordingly to consume our free tutorials. Other parties are acting as a conformist.



Anti Corruption Layer:

When two system interacts if we consume the **data directly from upstream** we **pollute** our **downstream system** as upstream data structure leak through the downstream so if the upstream become polluted our downstream too as it imitates the upstream data while consuming.

So it is a good idea while consume data from third party or from a legacy application always use a translation layer where the **upstream data translate to downstream** data structure before fed in to downstream.

This will help to resist the data leakage from upstream, if upstream contract changes it does not pollute downstream internal system only Translation layer has to be changed in order to adopt new data structure from upstream and convert it into downstream data structure, **this technique is called Anti-corruption layer.**

Anti-corruption layers save the downstream system from upstream changes.

Example : Notification module can implement an ACL while consuming data from payment module so if payment module data structure changes only ACL layers affected.



Open Host : In some cases, your Domain API needs to be accessed by many other services like our Free Tutorial Publisher module, Many external or internal domains want to consume this service, so as **Upstream** it should be **hosted** as a service and maintains a protocol and **service contract** like REST and JSON structure so another system can consume the data.

Published Language: Often two or more system receive and send messages among themselves, in that case, a common language will be needed for the transformation of the data from one system to another like XML, JSON we call that structure as Published language.

Tactical design tools

- Tactical design tools are **concerned with implementation details.**
- Generally takes care of components inside a bounded context.
- A de facto standard in development world
- **Tactical patterns are services, entities, repositories, factories etc.,**
- Tactical patterns is **expected to change** during product development.

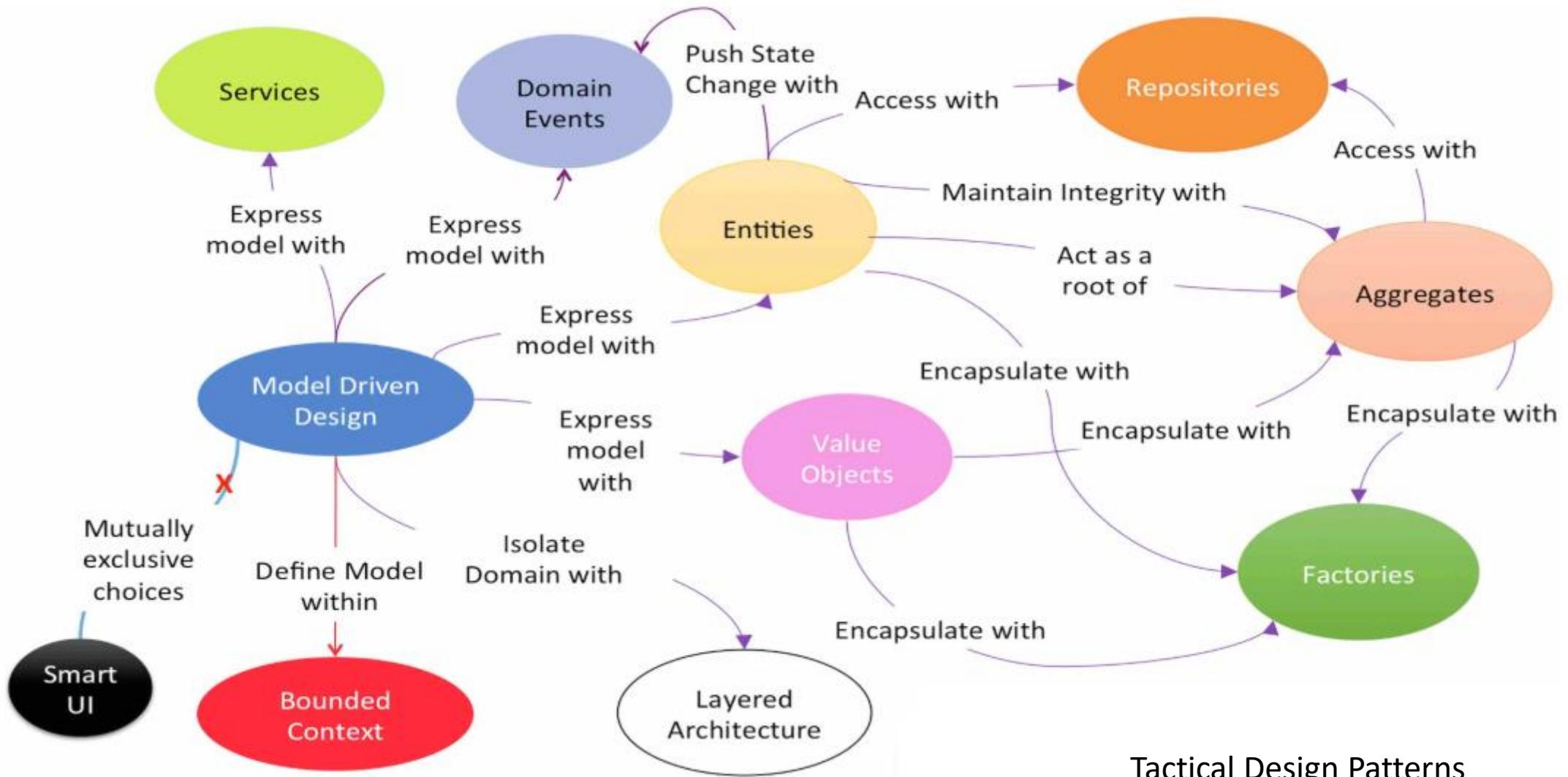
Model Driven Design

Domain

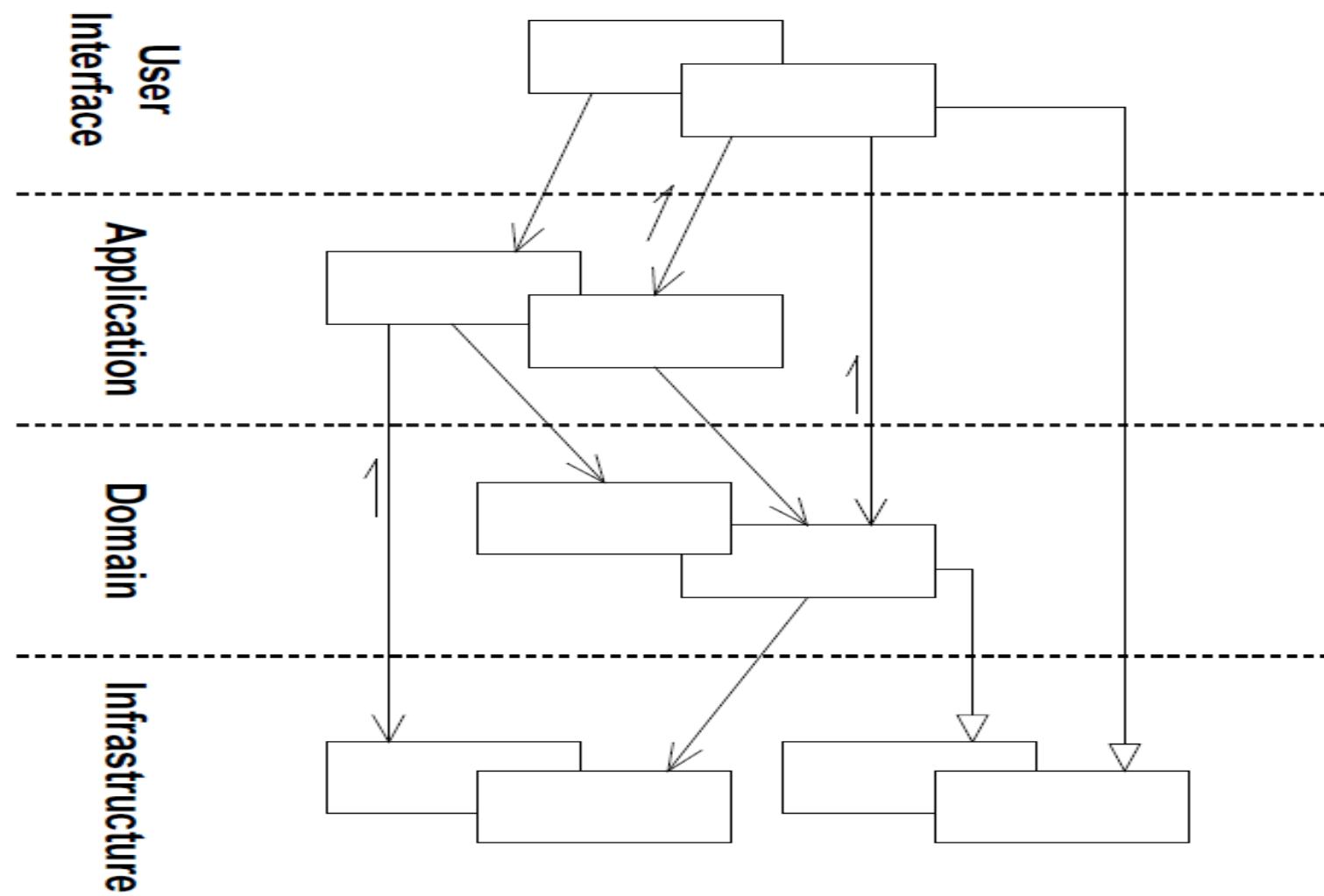
**Subdomain
Bounded Context**
Domain Model
Ubiquitous Language

**Subdomain
Bounded Context**
Domain Model
Ubiquitous Language

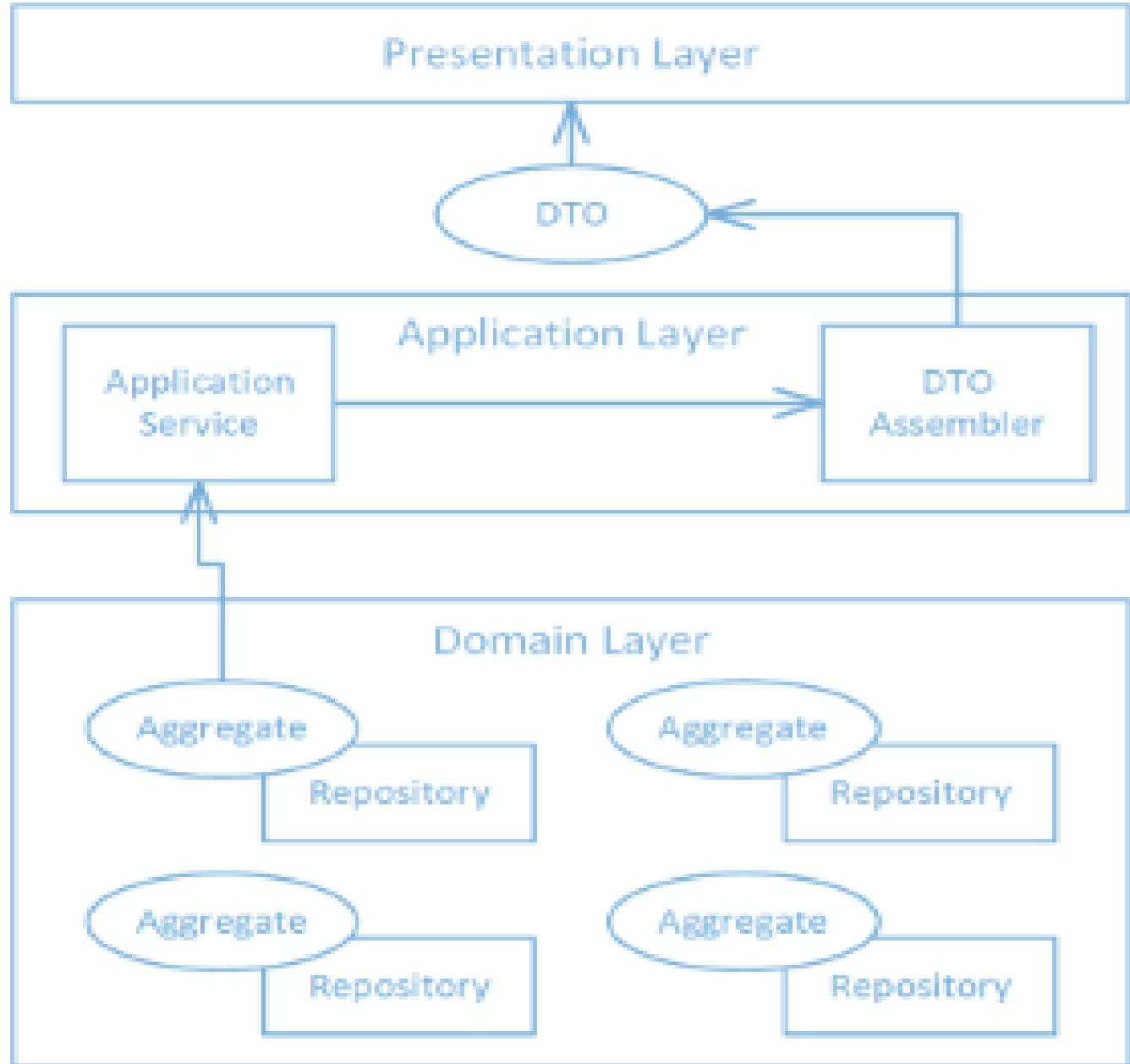
**Subdomain
Bounded Context**
Domain Model
Ubiquitous Language

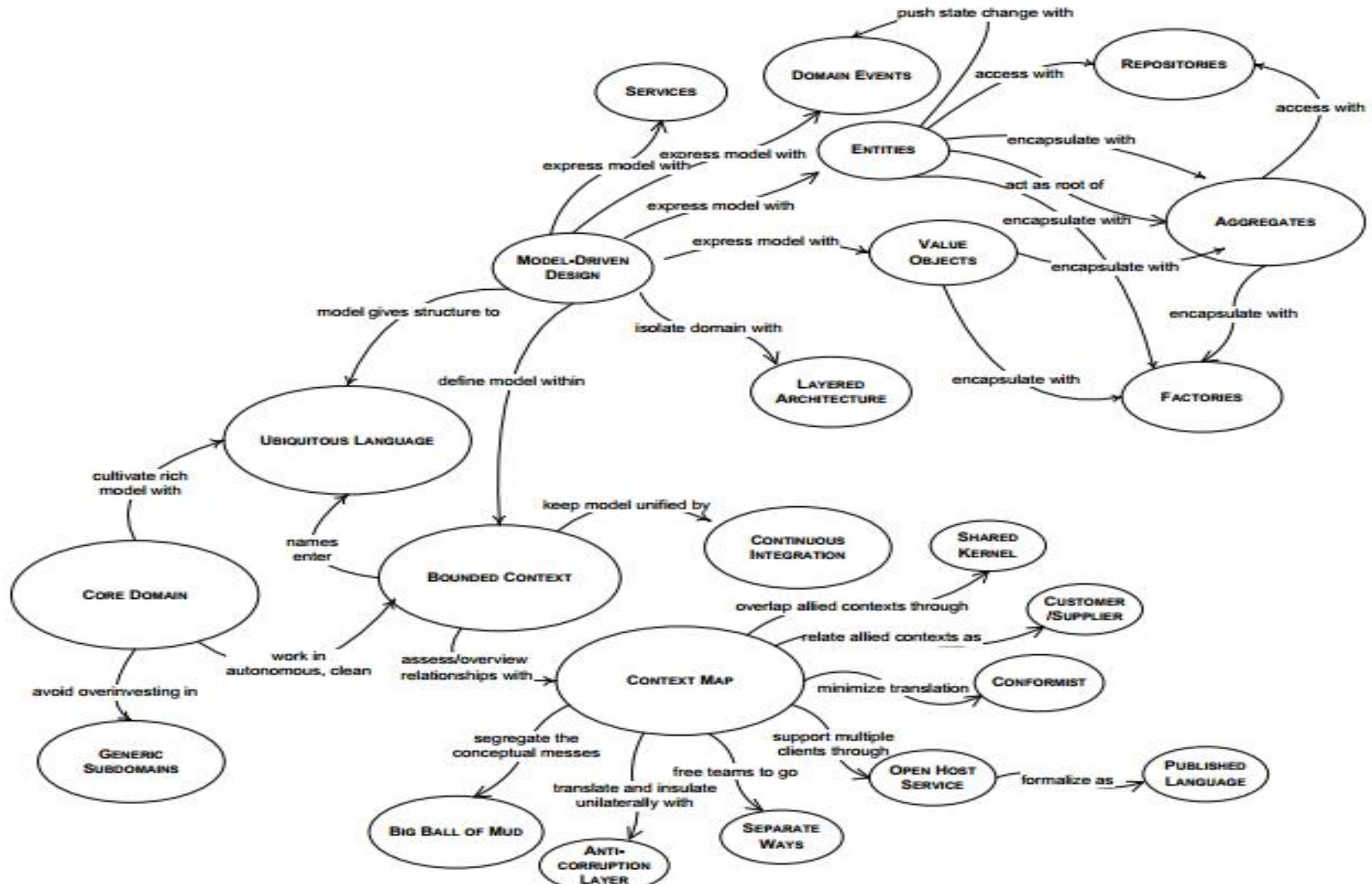


LAYERED ARCHITECTURE



User Interface (aka Presentation Layer)	Responsible for showing information to the user and interpreting the user's commands. The external actor might sometimes be another computer system rather than a human user.
Application Layer	Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.
Domain Layer (aka Model Layer)	Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. <i>This layer is the heart of business software.</i>
Infrastructure Layer	Provide generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, etc. The infrastructure layer may also support the pattern of interactions between the four layers through an architectural framework.





During the strategic phase of DDD, you are mapping out the business domain and defining bounded contexts for your domain models.

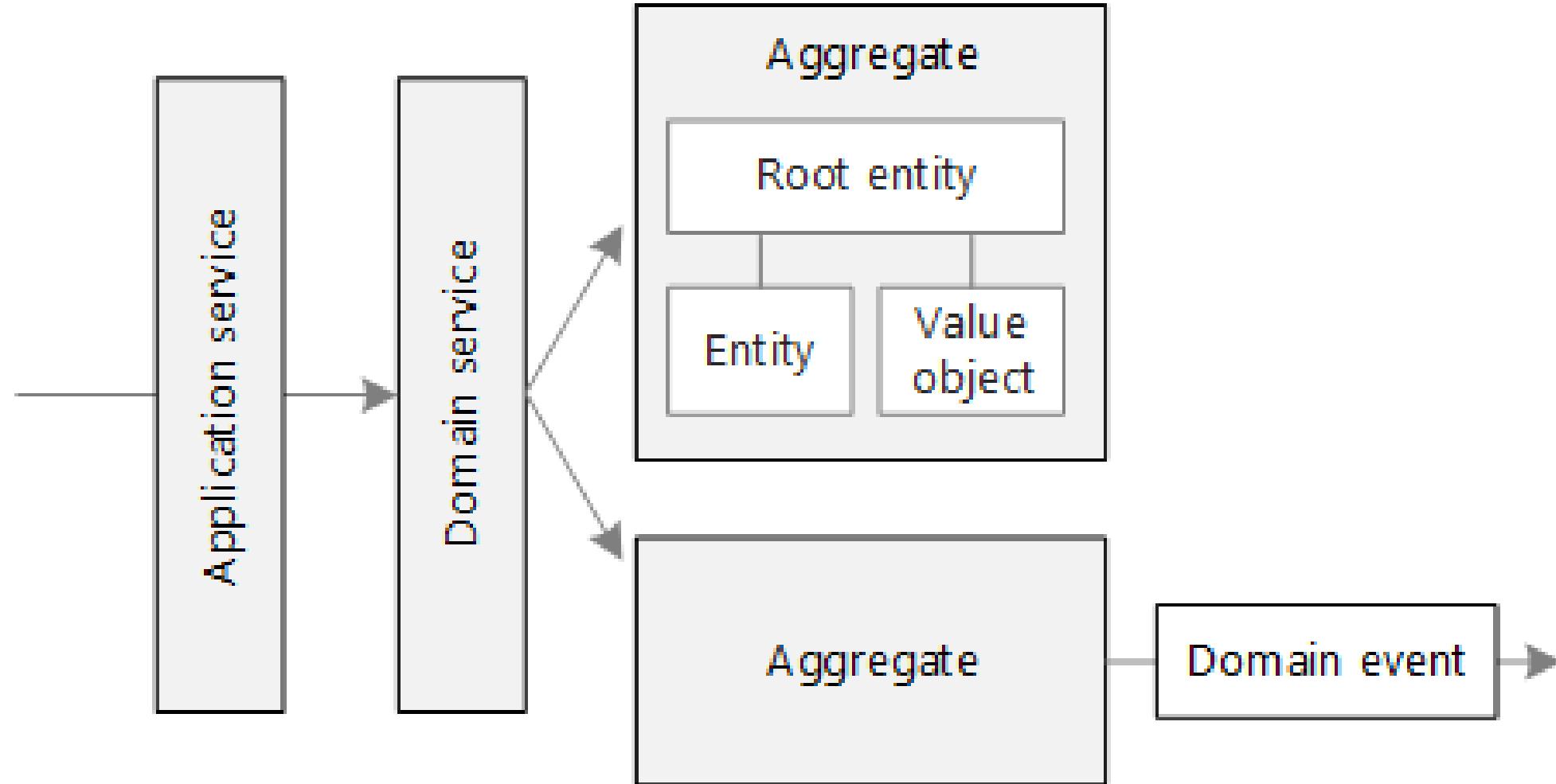
Tactical DDD is when you define your domain models with more precision.

The tactical patterns are applied within a single bounded context.

In a microservices architecture, the entity and aggregate patterns are very important.

Applying these patterns will help us to identify natural boundaries for the services in our application.

As a general principle, a microservice should be no smaller than an aggregate, and no larger than a bounded context.



Domain-driven design leads to entities

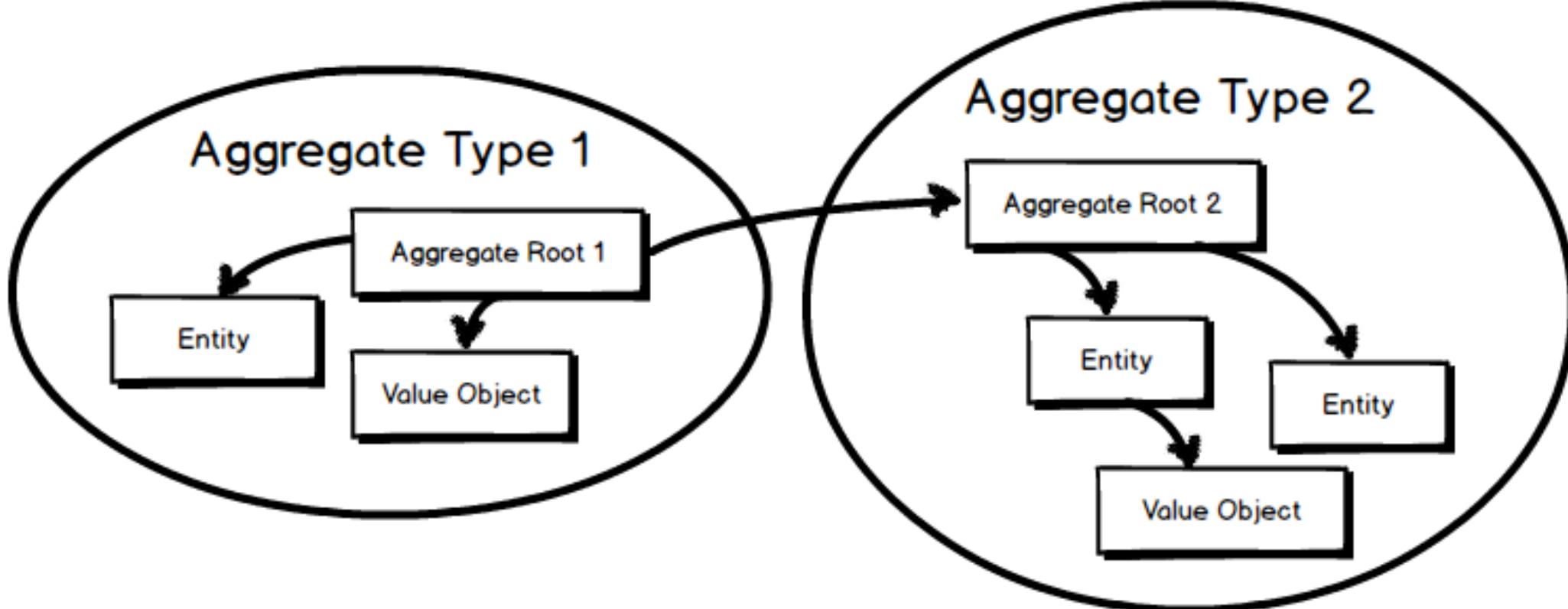
From the approach in domain-driven design, you get the following objects, among others:

Entity : “An object that is not defined by its attributes, but rather by a thread of continuity and its identity.”

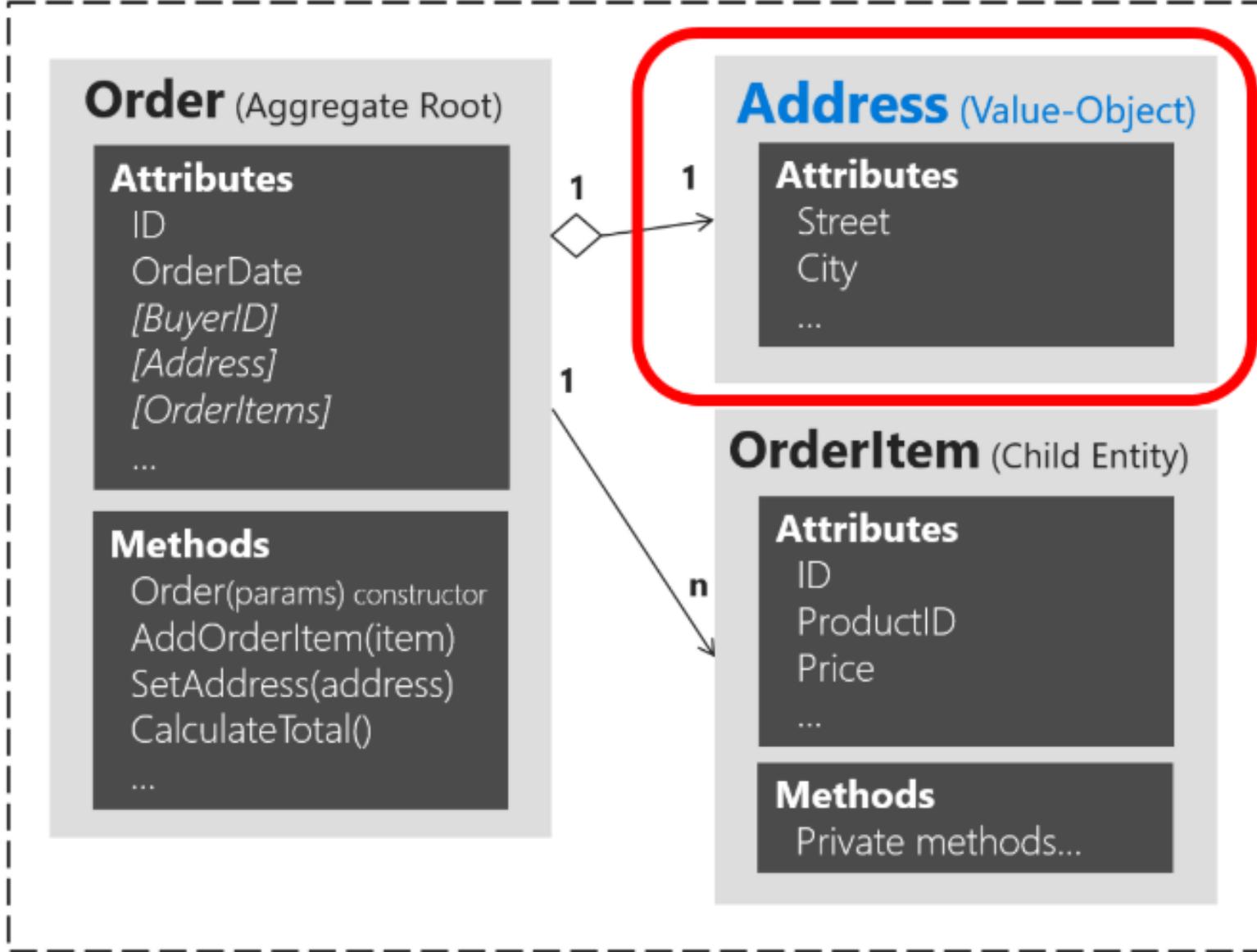
Value Objects : “An object that contains attributes but has no conceptual identity. They should be treated as immutable.”

Aggregate : “A collection of objects that are bound together by a root entity, otherwise known as an aggregate root. The aggregate root guarantees the consistency of changes being made within the aggregate by forbidding external objects from holding references to its members.”

Repository : “Methods for retrieving domain objects should delegate to a specialized Repository object such that alternative storage implementations may be easily interchanged.”



Order Aggregate (Multiple entities and Value-Object)



A **value object** is a small object that represents a simple entity whose equality is not based on identity: i.e. two value objects are equal when they have the same value, not necessarily being the same object.

Examples of value objects are objects representing an address, amount of money or a date range.

Value objects should be immutable; this is required for the implicit contract that two value objects created equal, should remain equal. It is also useful for value objects to be immutable, as client code cannot put the value object in an invalid state or introduce buggy behaviour after instantiation.

Value objects are among the building blocks of DDD.

```
// entity:  
class Person {  
    PersonId id; // global identity  
    FirstName firstName;  
    LastName lastName;  
    Address address;  
}  
// value objects:  
class PersonId {  
    Long value;  
}  
class FirstName {  
    String value;  
}  
class LastName {  
    String value;  
}  
class Address {  
    String street;  
    String streetNo;  
    String city;  
    String postalCode;  
}
```

As the value objects have no identity, we compare them together by simply comparing all the values they contain:

```
// Address
@Override
public boolean equals(Object o) {
    // basic checks and casting cut out for brevity
    return Objects.equals(street, address.street) &&
        Objects.equals(streetNo,
address.streetNo) &&
        Objects.equals(city, address.city) &&
        Objects.equals(postalCode,
address.postalCode);
}
@Override
public int hashCode() {
    return Objects.hash(street, streetNo, city,
postalCode);
}
```

Usually, we also make/treat the value objects as immutable, i.e. instead of changing the value objects, we create new instances that wrap the new values:

```
// wrong:
```

```
this.address.setStreet(event.street);
```

```
// good:
```

```
this.address = new Address(event.street, ...);
```

Immutable types are handy, as they can be easily shared between different objects and returned by the entities without the risk of compromising consistency.

From the conceptual perspective, it makes sense to create a new instance of a value object when the value changes, as we're literally assigning a new value.

The code gets more expressive:

```
// without:  
Map<Long, String>  
// with:  
Map<PersonId, PhoneNumber>
```

Repository is a pattern used to separate the domain model from the data access code. It acts as a bridge between the domain model and the data source (usually a database) and provides a set of methods for performing data access operations without exposing the underlying storage details to the domain logic.

Here are the key aspects and purposes of a Repository in DDD:

Abstraction of Data Access: A Repository abstracts the data access code, allowing the domain model to work with domain objects rather than directly dealing with database queries or storage mechanisms. This separation enhances the maintainability and testability of the domain model.

Aggregates and Entities: In DDD, the Repository primarily deals with Aggregates and Entities, which are fundamental building blocks of the domain model. Aggregates are consistency boundaries that group together Entities and Value Objects. Repositories are typically associated with specific Aggregates or Entity types.

CRUD Operations: Repositories provide methods for creating, reading, updating, and deleting domain objects. These methods are usually tailored to the needs of the specific domain and the associated Aggregate.

Persistence Ignorance: A well-designed Repository is often persistence-agnostic, meaning it doesn't have specific knowledge of how data is stored or retrieved. This allows for flexibility in choosing different data storage technologies, such as relational databases, NoSQL databases, or in-memory storage.

Query Methods: Repositories may also offer query methods for retrieving domain objects based on specific criteria. These query methods help in finding objects based on the domain's needs without exposing low-level query details.

Transaction Management: Repositories can help manage transactions by ensuring that data operations are performed within a single unit of work. This is important for maintaining data consistency.

Caching and Performance Optimization: Repositories can also incorporate caching mechanisms to improve data access performance, but this depends on the specific requirements of the application.

Domain and application services:

In DDD terminology, a service is an object that implements some logic without holding any state.

domain services, which encapsulate domain logic, and application services, which provide technical functionality, such as user authentication or sending an SMS message.

Domain services are often used to model behaviour that spans multiple entities.

A high-level structure of the monolithic
“Travel portal application”

```
monolithic-travel-portal/
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── example
│   │   │           └── travelportal
│   │   │               ├── TravelPortalApplication.java
│   │   │               ├── config
│   │   │               │   ├── SecurityConfig.java
│   │   │               │   └── WebConfig.java
│   │   │               ├── controller
│   │   │               │   ├── FlightController.java
│   │   │               │   ├── HotelController.java
│   │   │               │   ├── UserController.java
│   │   │               │   └── PaymentController.java
│   │   │               ├── model
│   │   │               │   ├── Flight.java
│   │   │               │   ├── Hotel.java
│   │   │               │   ├── User.java
│   │   │               │   └── Payment.java
│   │   │               ├── repository
│   │   │               │   ├── FlightRepository.java
│   │   │               │   ├── HotelRepository.java
│   │   │               │   ├── UserRepository.java
│   │   │               │   └── PaymentRepository.java
│   │   │               └── service
│   │   │                   ├── FlightService.java
│   │   │                   ├── HotelService.java
│   │   │                   ├── UserService.java
│   │   │                   └── PaymentService.java
│   │   └── resources
│   │       └── application.properties
└── test
└── pom.xml
```

Advantages of Monolithic Travel Portal:

- 1. Simplicity:** The monolithic architecture is easier to develop, deploy, and manage since all functionalities are contained within a single application.
- 2. Easier Development:** It's easier to develop and debug in a monolithic environment as all components are bundled together.
- 3. Performance:** In some cases, monolithic applications can have better performance due to direct method calls and in-process communication.
- 4. Less Overhead:** There's no overhead associated with managing multiple services, deployment pipelines, and communication protocols.

Disadvantages of Monolithic Travel Portal:

- 1. Scalability:** Monolithic applications can be challenging to scale horizontally as all components must scale together.
- 2. Technology Diversity:** Since all components are built together, it's difficult to use different technologies or languages for different parts of the application.
- 3. Deployment Complexity:** Updating or deploying changes to a monolithic application often requires redeploying the entire application, leading to downtime.
- 4. Team Autonomy:** Monolithic applications can hinder team autonomy as changes to any part of the application require coordination among multiple teams.
- 5. Maintenance Challenges:** As the application grows, it becomes more challenging to maintain and understand due to its size and complexity.

Domain Driven Patterns

Domain:

- A domain represents a specific sphere of knowledge, activity, or interest within a business or problem space. It encompasses the subject area in which the software system operates and includes all the concepts, rules, and processes related to that subject area.
- In DDD, the domain is the central focus of software development, and the goal is to model the domain effectively to solve business problems and deliver value to stakeholders.
- For example, in the context of a travel portal application, the domain includes concepts such as users, flights, hotels, bookings, payments, etc.

Subdomain:

- A subdomain is a logical partition or subdivision of the domain that represents a specific aspect or area of the business or problem space.
- Subdomains help to organize and categorize related concepts, rules, and processes within the larger domain.
- Subdomains do not necessarily imply any specific implementation or technical boundaries. They are primarily concerned with organizing and understanding the domain from a business perspective.

Bounded Context:

- A bounded context is a technical and implementation-oriented concept within DDD that defines the scope and boundaries within which a particular model or set of concepts applies.
- Bounded contexts help to manage complexity and ensure consistency by establishing clear boundaries for models, concepts, and interactions within a specific context.
- Bounded contexts often align closely with subdomains, but they are more concerned with delineating technical and implementation boundaries rather than purely conceptual divisions.
- Bounded contexts typically include a well-defined model, language, and set of rules that apply within the context, and they may interact with other bounded contexts through explicit interfaces or integration points.

Both subdomains and bounded contexts help to organize and understand complex domains, they serve different purposes and focus on different aspects of domain-driven design.

Subdomains primarily represent logical divisions of the domain from a business perspective, while bounded contexts define technical boundaries and implementation scopes within which specific models or concepts apply.

Subdomains:

- Within the travel portal domain, there are several subdomains, each focusing on a specific aspect or area of the travel booking and management process.
- Subdomains in the travel portal might include:
 - User Management: Responsible for user authentication, registration, profile management, etc.
 - Flight Booking: Deals with searching for flights, booking flights, managing flight reservations, etc.
 - Hotel Booking: Handles searching for hotels, booking hotels, managing hotel reservations, etc.
 - Payment Processing: Manages payment transactions, processing payments, handling refunds, etc.
- Each subdomain represents a logical partition or subdivision of the travel portal domain, focusing on a specific business capability or concern.

Bounded Contexts:

- Bounded contexts define the scope and boundaries within which a particular model or set of concepts applies within the travel portal.
- In the context of the travel portal, each subdomain can be considered a bounded context.
- For example:
 - The User Management subdomain may have its own bounded context, defining the scope of user-related concepts, rules, and processes.
 - The Flight Booking subdomain may have another bounded context, specifying the scope of flight-related concepts and interactions.
 - Similarly, the Hotel Booking and Payment Processing subdomains each have their own bounded contexts.
- Bounded contexts help to manage complexity and ensure consistency by establishing clear boundaries for models, concepts, and interactions within each specific area of the travel portal.

Ubiquitous Language:

- ✓ Ubiquitous Language refers to a shared vocabulary and glossary of terms that is understood and used consistently by all stakeholders, including domain experts, developers, and users.
- ✓ The language used in the codebase should closely match the language used by domain experts when discussing the problem domain.
- ✓ By using a ubiquitous language, we ensure that communication is clear and effective between all parties involved in the development process.

- ✓ For example, terms such as User, Flight, Hotel, Booking, and Payment should be consistently used and understood across the codebase and documentation.
- ✓ When naming classes, methods, variables, and other artifacts within the codebase, we should use domain-specific terminology that aligns with the ubiquitous language.
- ✓ Additionally, domain concepts and rules should be expressed in code using the same terminology used in discussions with domain experts.
- ✓ For example, when defining a service responsible for flight booking, we might name it FlightBookingService to clearly indicate its purpose and align with the ubiquitous language used by stakeholders.

Context Mapping:

- We'll define explicit boundaries and relationships between the subdomains (bounded contexts) within the travel portal.
- We'll establish communication channels and integration patterns between microservices to ensure consistency and data synchronization.
- For example, we may use synchronous RESTful APIs for communication between bounded contexts that have a strong coupling, and asynchronous messaging (e.g., Kafka) for loosely coupled interactions.

Shared Kernel:

- We'll identify common domain elements, concepts, and interfaces that are shared across multiple bounded contexts.
- These shared elements will be encapsulated in a shared kernel, which serves as a central repository of domain knowledge and functionality.
- For example, common data structures or utility functions related to user authentication and authorization may be included in the shared kernel

Entity Pattern:

- ✓ Entities represent objects within the domain that have a distinct identity and lifecycle.
- ✓ In the context of the travel portal, entities may include User, Flight, Hotel, Booking, Payment, etc.
- ✓ Each entity encapsulates its state and behavior and is uniquely identifiable within the system.
- ✓ Entities are responsible for enforcing business rules and ensuring consistency of their state.

- ✓ For example, the User entity within the User Management bounded context represents individual users of the travel portal. Each user has a unique identity (e.g., user ID) and associated attributes such as username, email, password, etc.
- ✓ Similarly, the Flight entity within the Flight Booking bounded context represents individual flights available for booking. Each flight has attributes such as flight number, departure and arrival locations, departure time, etc.
- ✓ Entities may also have relationships with other entities, such as a Booking entity being associated with a User and a Flight entity, representing a user's reservation of a particular flight.

Aggregates:

- ✓ Within each bounded context, we'll identify aggregate roots and define aggregates to enforce consistency boundaries and transactional integrity.
- ✓ Aggregates will encapsulate related domain entities and specify rules for their lifecycle and interactions.
- ✓ For example, within the Flight Booking bounded context, the Booking entity may serve as an aggregate root, encapsulating the Flight and Reservation entities.

Value Objects:

- ✓ We'll use value objects to represent immutable, interchangeable elements within the domain that do not have a distinct identity.
- ✓ Value objects help to reduce complexity and improve expressiveness by encapsulating related attributes and behaviors into cohesive units.
- ✓ For example, the Address entity within the User Management bounded context may be implemented as a value object.

Domain Services:

- ✓ We'll define domain services to encapsulate domain logic that does not naturally fit within the scope of an entity or value object.
- ✓ Domain services provide a way to express behavior that is not tied to any specific entity and promotes encapsulation and reusability of domain logic.
- ✓ For example, a BookingService within the Flight Booking bounded context may contain business logic for searching for available flights, validating bookings, etc.

Repositories:

- ✓ We'll use repositories to abstract data access and provide a way to retrieve and manipulate domain objects from the underlying data store.
- ✓ Repositories act as a bridge between the domain model and the data persistence layer, facilitating separation of concerns and promoting testability and maintainability.
- ✓ For example, UserRepository within the User Management bounded context may define methods for retrieving and updating user data from the database.

```
user-service/
└── src
    ├── main
    │   ├── java
    │   │   └── com.example.userservice
    │   │       ├── controller
    │   │       │   └── UserController.java
    │   │       ├── model
    │   │       │   └── User.java
    │   │       ├── repository
    │   │       │   └── UserRepository.java
    │   │       └── service
    │   │           └── UserService.java
    │   └── resources
    │       └── application.properties
    └── test
└── pom.xml
```

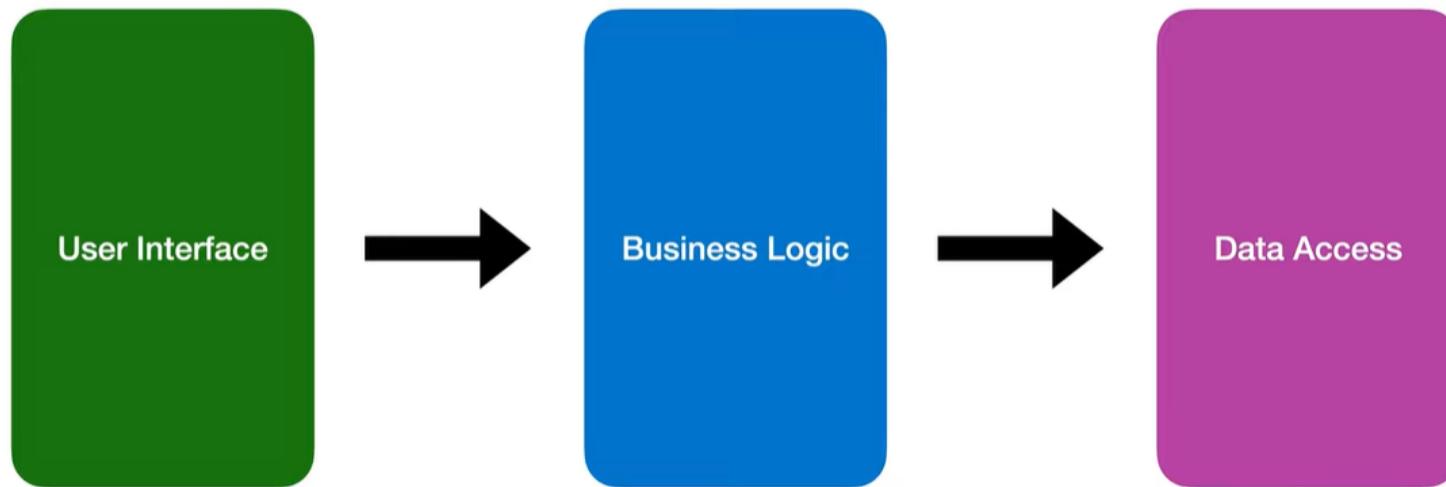
```
flight-service/
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com.example.flightservice
│   │   │       ├── controller
│   │   │       │   └── FlightController.java
│   │   │       ├── model
│   │   │       │   └── Flight.java
│   │   │       ├── repository
│   │   │       │   └── FlightRepository.java
│   │   │       └── service
│   │   │           └── FlightService.java
│   │   └── resources
│   │       └── application.properties
│   └── test
└── pom.xml
```

```
hotel-service/
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com.example.hotelservice
│   │   │       ├── controller
│   │   │       │   └── HotelController.java
│   │   │       ├── model
│   │   │       │   └── Hotel.java
│   │   │       ├── repository
│   │   │       │   └── HotelRepository.java
│   │   │       └── service
│   │   │           └── HotelService.java
│   │   └── resources
│   │       └── application.properties
└── test
└── pom.xml
```

```
payment-service/
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com.example.paymentservice
│   │   │       ├── controller
│   │   │       │   └── PaymentController.java
│   │   │       ├── model
│   │   │       │   └── Payment.java
│   │   │       ├── repository
│   │   │       │   └── PaymentRepository.java
│   │   │       └── service
│   │   │           └── PaymentService.java
│   │   └── resources
│   │       └── application.properties
└── test
└── pom.xml
```

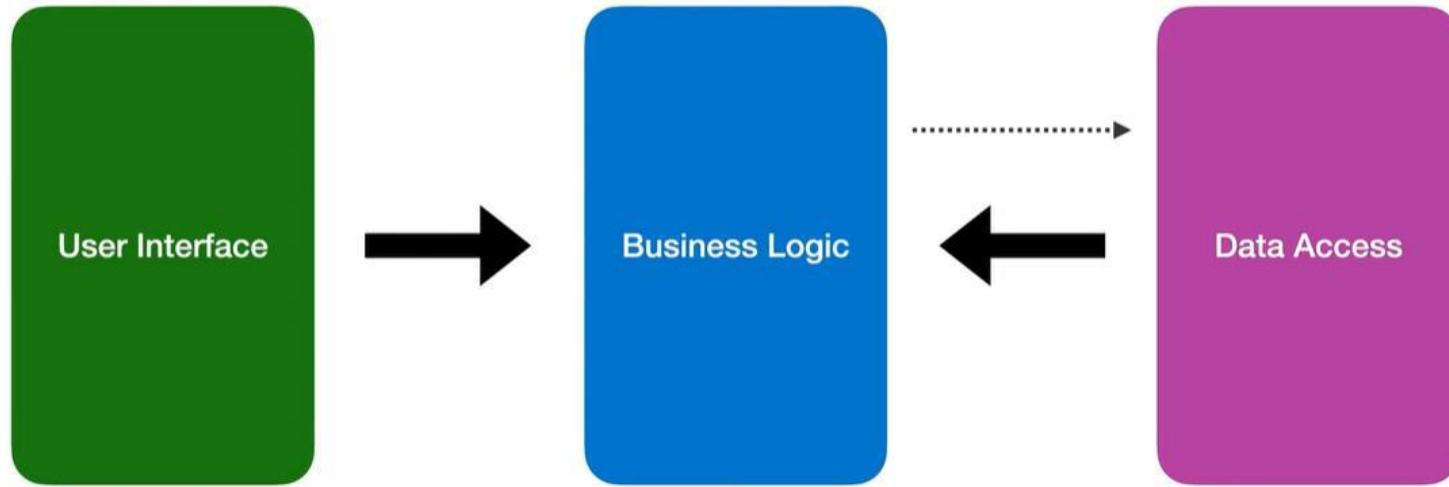
Hexagonal, Onion & Clean Architecture

N-Tier / Layered Architecture



When we invert the dependency between the business logic and the data access. The business logic still makes calls into data access layer. But it does not know what it's calling.

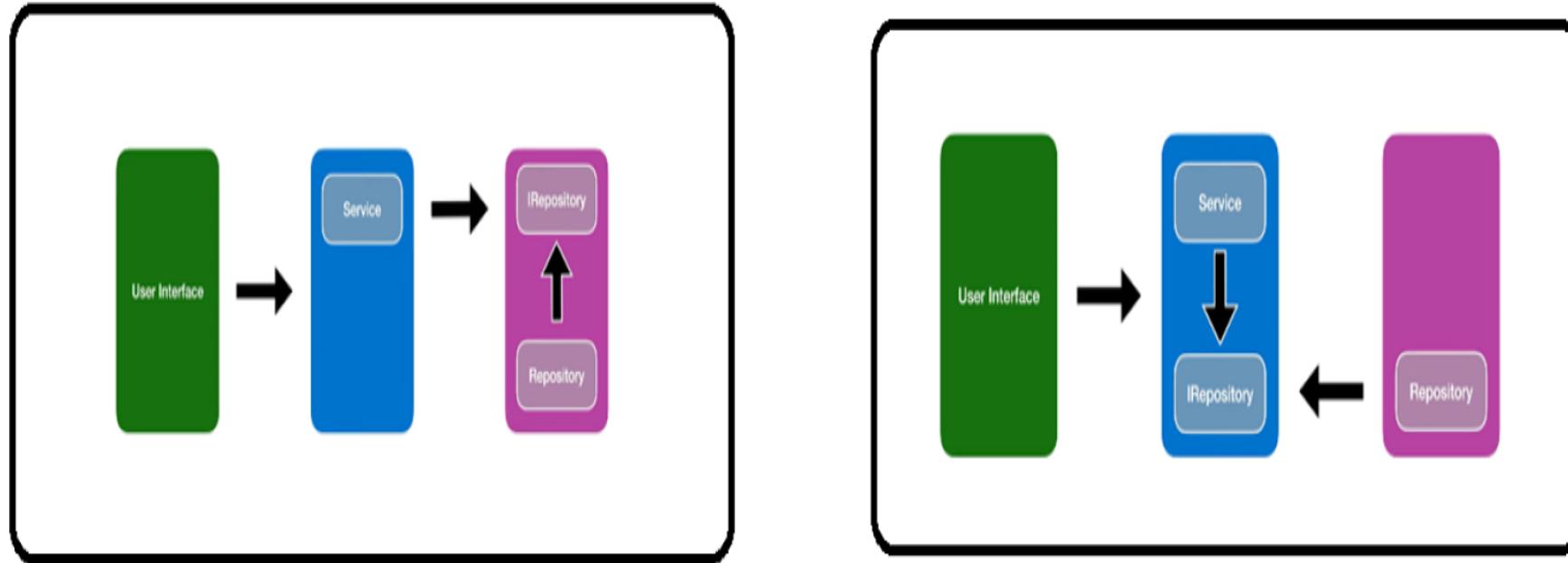
The business logic layer defines an interface that it expects to be implemented in the data access layer.



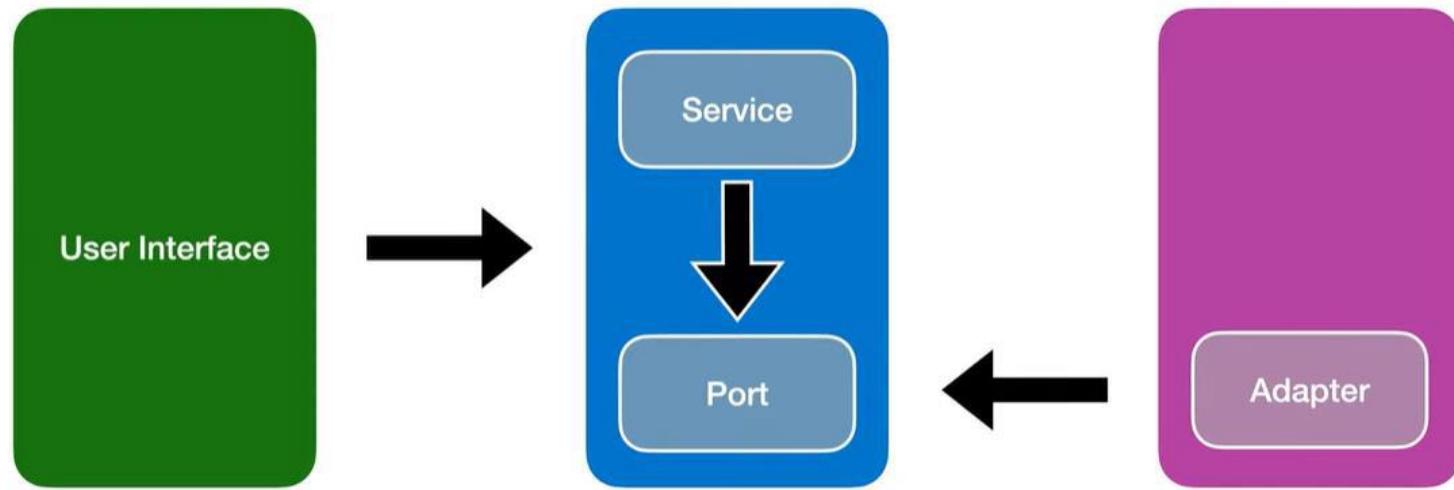
When we invert the dependency between the business logic and the data access. The business logic still makes calls into data access layer.

But it does not know what it's calling.

The business logic layer defines an interface that it expects to be implemented in the data access layer.

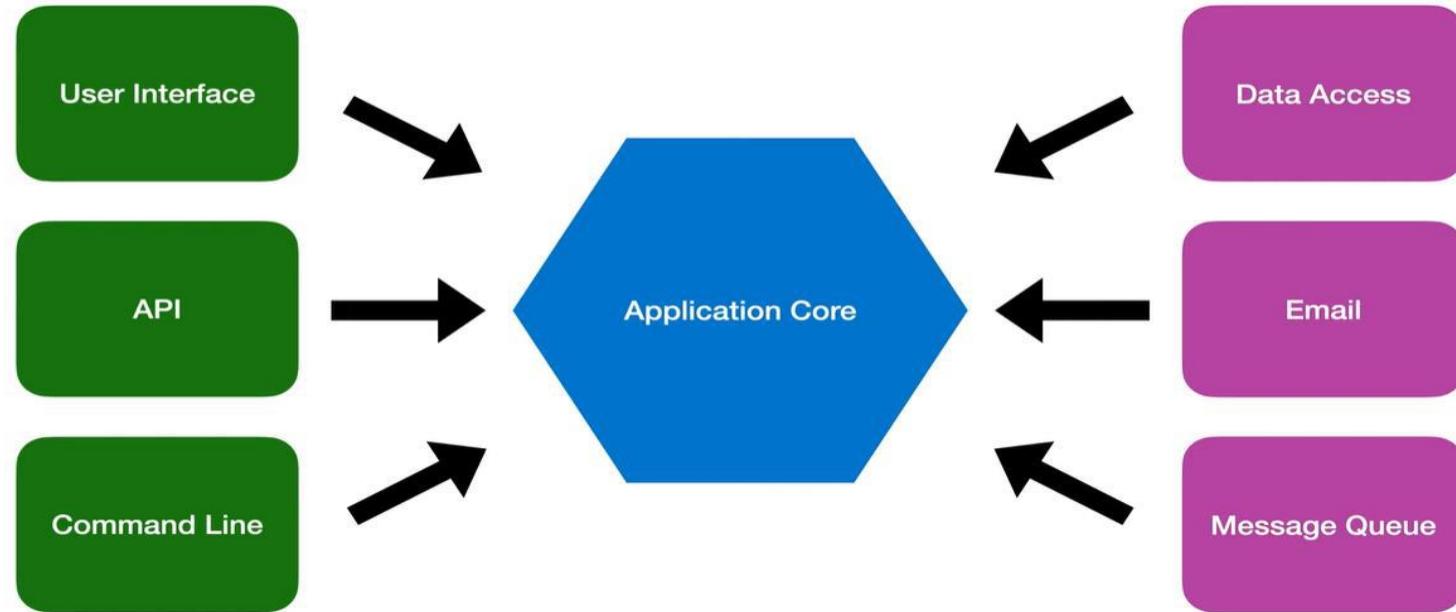


Normally **IRepository** interface living alongside the repository implementation, but simply moving this interface means we invert the direction of dependency between these layers.



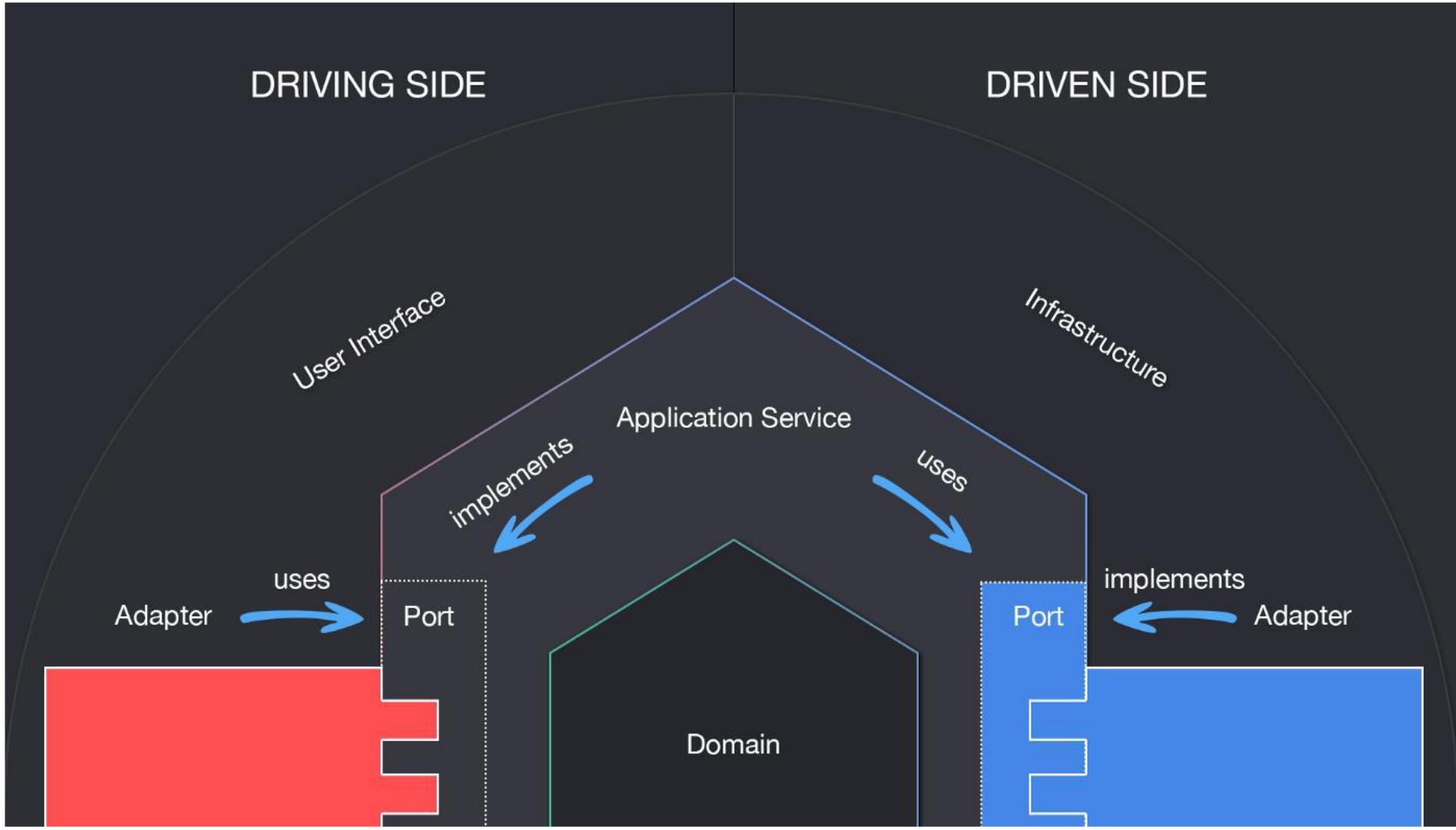
The interface is called a "port". The data access layer knows how to implement that port by using an "adapter".

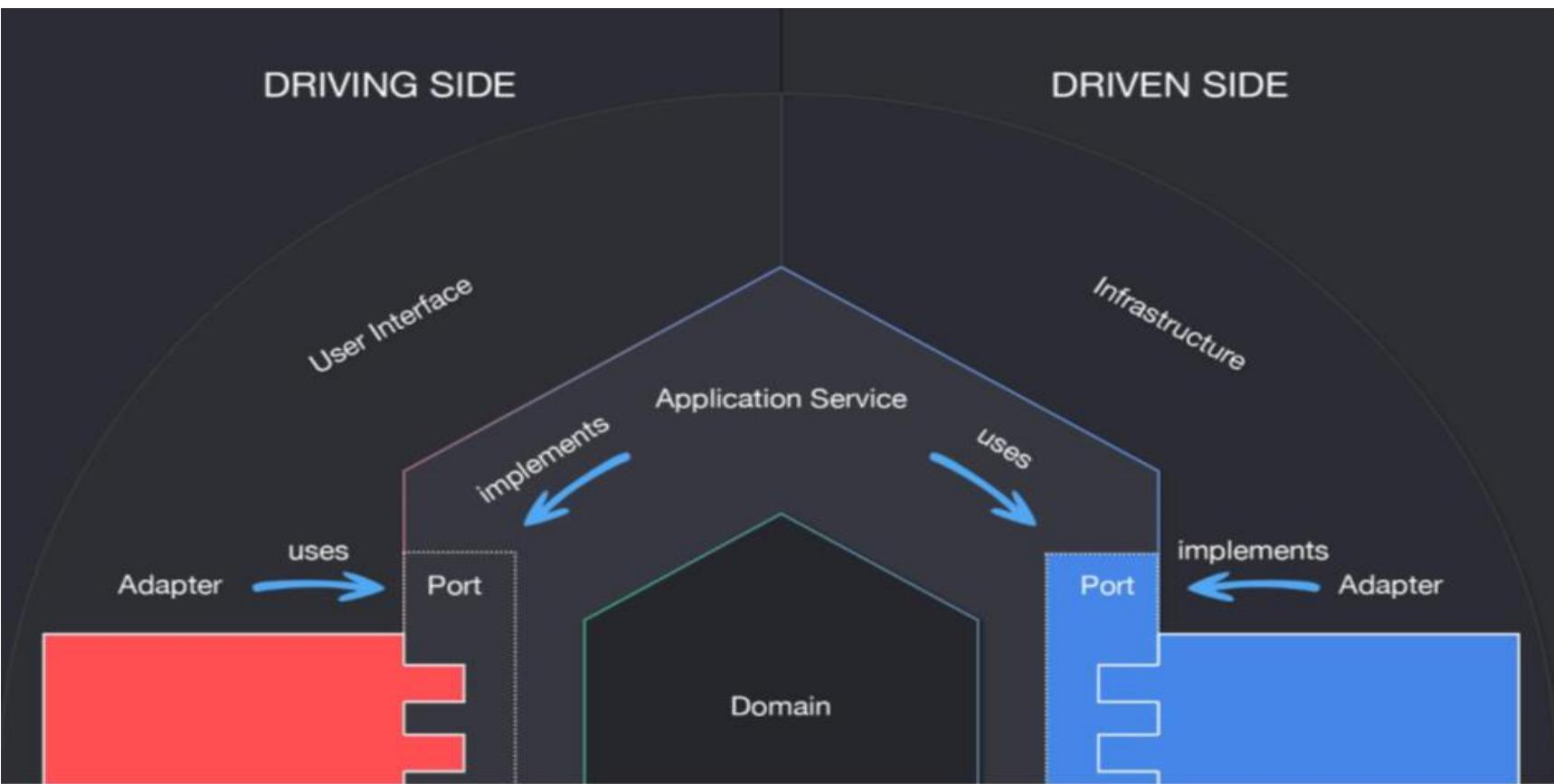
This architecture became known as "Ports and Adapters", but it was originally called the "Hexagonal Architecture".



The core logic has no dependencies; it is pure code with no networks and no reference to the outside world, which makes it very easy to write tests for.

On the left we can add adapters for other presentations such as an API or a command line interface. These are called the "Primary" or the "Driving" adapters because they drive, or they make calls into, the core. On the right hand side we have the adapters for the infrastructure, such as first sending emails or using a message queue. These are called the "Secondary" or the "driven" adapters because of the inverted control; it is the core that drives them.





Application Service -> UserService

Driving Side

|_ Port : SubmitNewUser, DisplayUser, ModifyUser, DeleteUser

|_ Adapter: UserRestApiAdapter

Driven Side

|_ Port: UserRepository

|_ Adapter: UserInMemoryDatabaseAdapter,UserJpaAdapter

Domain Model: The domain model encapsulates the core business logic and rules of the application.

Application Services act as mediators between external clients and the domain model, invoking domain-specific operations and coordinating the execution of use cases based on the requests received from external clients.

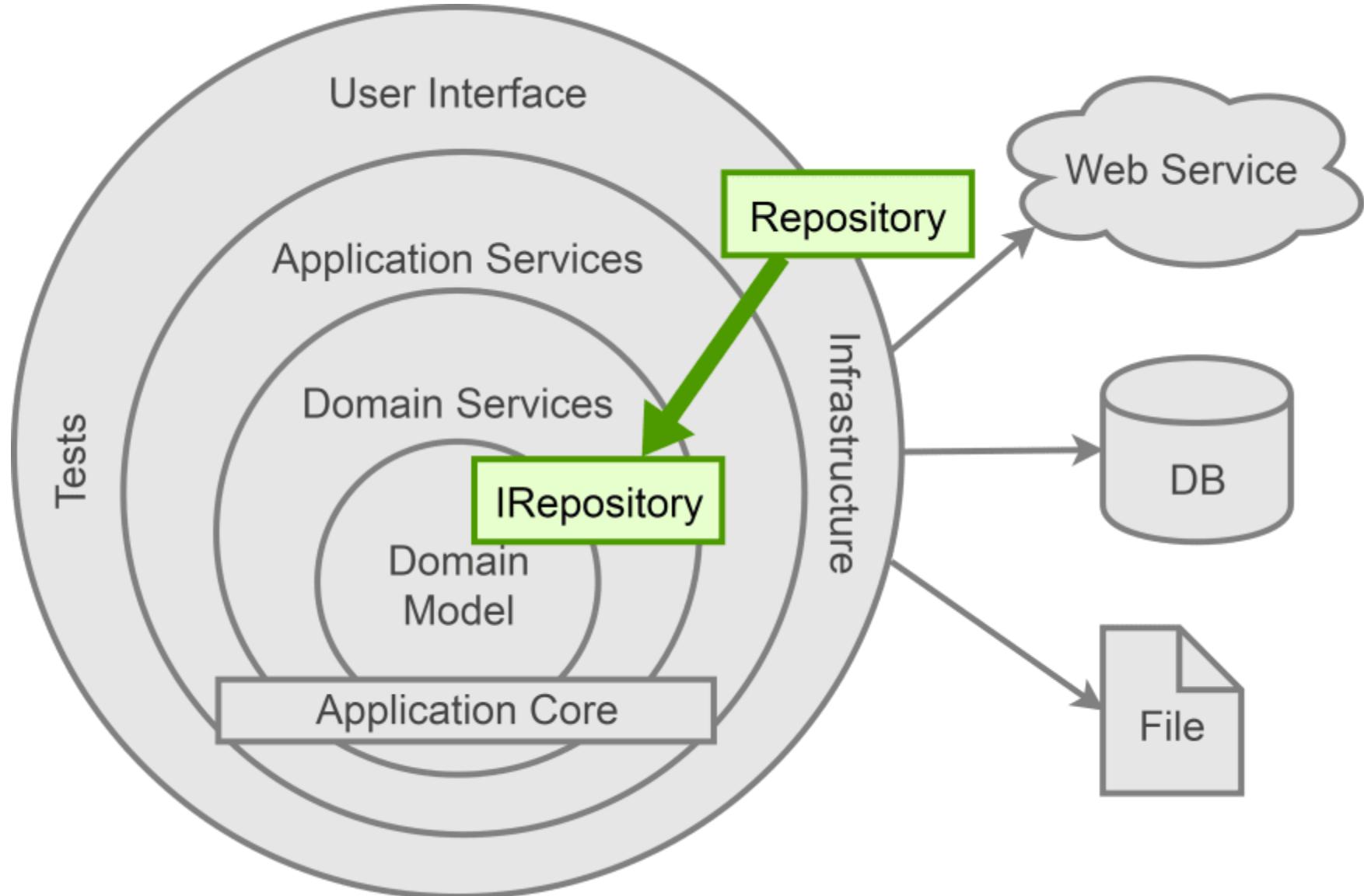
Driving Side vs Driven Side

- ✓ Driving (or primary) actors are the ones that initiate the interaction, and are always depicted on the left side.
- ✓ For example, a driving adapter could be a controller which is the one that takes the (user) input and passes it to the Application via a Port.
- ✓ Driven (or secondary) actors are the ones that are “kicked into behaviour” by the Application. For example, a database Adapter is called by the Application so that it fetches a certain data set from persistence.

- ✓ Ports will be (most of the time, depending on the language you choose) represented as interfaces in code.
- ✓ Driving Adapters will use a Port and an Application Service will implement the Interface defined by the Port, in this case both the Port's interface and implementation are inside the Hexagon.
- ✓ Driven adapters will implement the Port and an Application Service will use it, in this case the Port is inside the Hexagon, but the implementation is in the Adapter, therefore outside of the Hexagon.

- ✓ Dependency Inversion in the Hexagonal Architecture Context
Agile Software Development Principles:
 - ✓ High-level modules should not depend on low-level modules. Both should depend on abstractions.
 - ✓ Abstractions should not depend on details. Details should depend on abstractions.
 - ✓ The left and right sides of the Hexagon contain 2 different types of actors, Driving and Driven where both Ports and Adapters exist.
 - ✓ On the Driving side, the Adapter depends on the Port, which is implemented by the Application Service, therefore the Adapter doesn't know who will react to its invocations, it just knows what methods are guaranteed to be available, therefore it depends on an abstraction.
 - ✓ On the Driven side, the Application Service is the one that depends on the Port, and the Adapter is the one that implements the Port's Interface, effectively inverting the dependency since the 'low-level' adapter (i.e. database repository) is forced to implement the abstraction defined in the application's core, which is 'higher-level'.

The Onion Architecture



Onion Architecture is a software architectural pattern that promotes a modular and loosely coupled design, focusing on separation of concerns and maintainability. It helps developers create applications that are more flexible, testable, and easier to evolve over time.

Key Concepts of Onion Architecture:

Dependency Rule:

Dependencies flow inward, with inner layers having no knowledge of outer layers. This ensures that high-level modules do not depend on low-level modules directly. Instead, both depend on abstractions, enabling interchangeable implementations and reducing coupling.

Layers of the Onion:

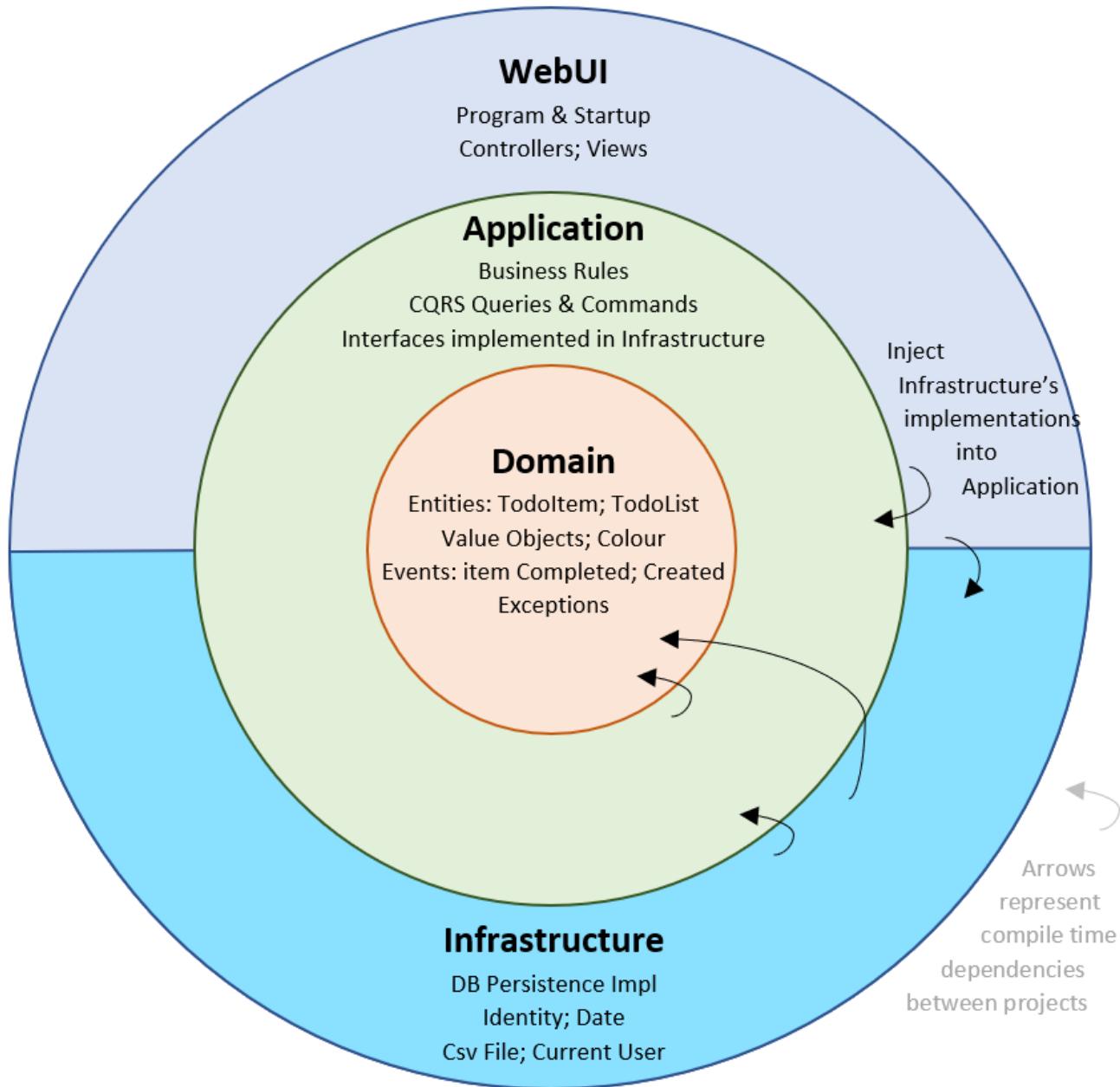
Domain Layer: Contains the core business logic, entities, and business rules of the application.

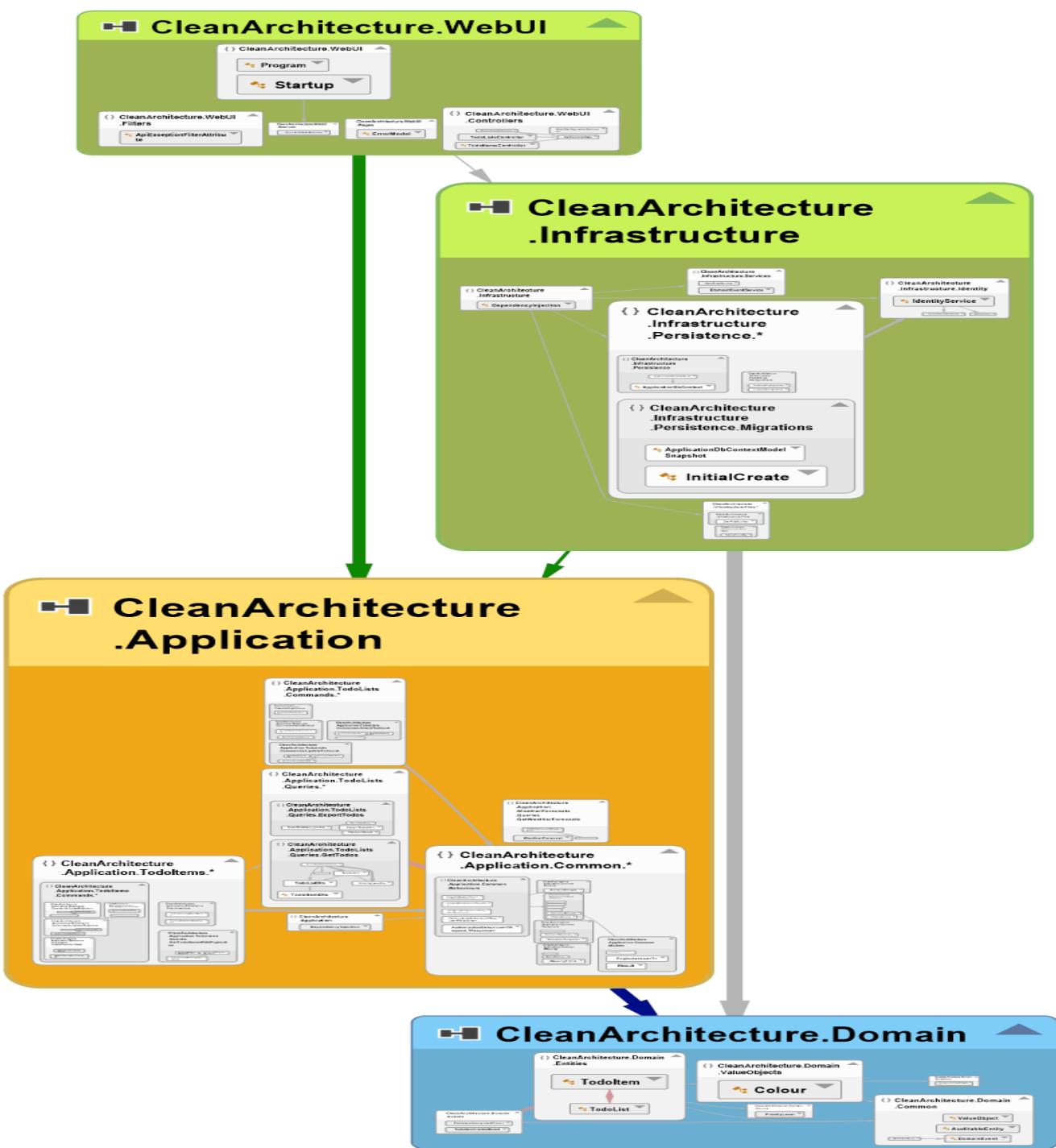
Application Layer: Implements use cases and coordinates the flow of data between the domain and infrastructure layers.

Infrastructure Layer: Handles external concerns such as databases, file systems, or external services.

Presentation Layer: Handles user interfaces and presentation-related logic.

The Clean Architecture





Key Principles of Clean Architecture

The clean architecture combines many software design principles and practices in a single architecture. Some of the key principles of clean architecture are as follows:

Separation of Concerns

Clean architecture organized the code into layers in such a way that each layer is responsible for a specific part of the application. All layers of the system are independent or decoupled, allowing us to introduce a change and test business logic or user interfaces without impacting other layers or areas of the application.

Dependency Inversion Principle (DIP)

In clean architecture, the high-level modules are defined close to the domain or business logic, and low-level modules are defined close to the input and output of the program.

The dependency inversion principle states that high-level modules should not depend on low-level modules.

The dependencies should be inverted towards the inner layers. In other words, the abstractions should not depend upon details, the details should depend upon abstractions.

The key layers in Clean Architecture are typically:

Entities (Core Business Logic) : Represent core business objects or domain entities.

Use Cases (Application Service Layer) : Contain application-specific business rules and use cases, independent of external frameworks or delivery mechanisms.

Interface Adapters (Web Layer) : Convert data from external sources (such as UI, databases, or external services) into formats usable by the use cases and vice versa.

Frameworks and Drivers (Infrastructure Layer) : Consist of external frameworks, tools, and delivery mechanisms such as web frameworks, databases, UIs, etc.

Single Responsibility

Each software module or a class should have one and only one reason to change

Liskov Substitution

You should be able to use any derived class instead of a base class without modification

Dependency Inversion

High level classes should not depend on low level classes instead both should depend upon abstraction

S O L I D

Open/Closed

A Software Class or module should be open for extension but closed for modification

Design Principles

Interface Segregation

Client should not be forced to use an interface which is not relevant to it

The SOLID principles and design patterns are both fundamental concepts in software engineering, but they serve different purposes and operate at different levels of abstraction. Here are the key differences between the two:

Purpose:

- ✓ **SOLID Principles:** These are a set of five principles aimed at guiding software design to make software more understandable, flexible, and maintainable. They focus on the design of individual classes and the relationships between them.
- ✓ **Design Patterns:** Design patterns are reusable solutions to commonly occurring problems in software design. They provide general, high-level templates for organizing code and solving specific design problems.

Granularity:

- ✓ **SOLID Principles:** These principles operate at a lower level of abstraction, focusing on the design of individual classes and their responsibilities, relationships, and interactions.
- ✓ **Design Patterns:** Design patterns operate at a higher level of abstraction, providing solutions that can be applied to entire components, subsystems, or even entire applications.

Scope:

- ✓ **SOLID Principles:** These principles are more universal and apply across various programming languages and paradigms. They are fundamental guidelines for designing object-oriented software.

- ✓ **Design Patterns:** Design patterns are more specific and often tied to particular programming paradigms or languages. While many design patterns are applicable across different languages, some are more closely associated with specific environments or technologies.

Number:

- ✓ **SOLID Principles:** There are five SOLID principles: Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP).
- ✓ **Design Patterns:** There are numerous design patterns, categorized into three main types: creational, structural, and behavioral patterns. Examples include Singleton, Factory Method, Observer, Strategy, etc.

Usage:

- SOLID Principles:** SOLID principles guide developers in creating well-structured, maintainable, and extensible code. They help in designing classes and their relationships in a way that makes the code easier to understand, maintain, and extend.

- Design Patterns:** Design patterns provide specific solutions to recurring design problems. They help developers create more modular, reusable, and maintainable code by following proven best practices.

Single Responsibility Principle (SRP):

A class should have only one reason to change. In other words, each class should have only one responsibility or should only do one thing.

Single Responsibility Principle (SRP):

Use case: E-commerce Platform

Microservice: Order Management Service

Responsibility: Handles order creation, updating, and processing

Justification: This microservice has a single responsibility: managing orders. It doesn't handle user authentication or inventory management. This separation ensures that changes to the order management process won't affect other parts of the system.

Open/Closed Principle (OCP):

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to extend the behavior of a module without modifying its source code

Open/Closed Principle (OCP):

Use case: E-commerce Platform

Microservice: Payment Gateway Service

Requirement: Support for new payment methods

Justification: The Payment Gateway Service is designed to be open for extension. When a new payment method needs to be added (e.g., cryptocurrency), a new module can be created to handle this without modifying the existing codebase. This ensures that existing payment methods continue to work without disruption.

Liskov Substitution Principle (LSP):

Subtypes must be substitutable for their base types without affecting the correctness of the program. In simpler terms, objects of a superclass should be replaceable with objects of its subclasses without altering the correctness of the program.

Liskov Substitution Principle (LSP):

Use case: E-commerce Platform

Microservice: Shipping Service

Requirement: Support for different shipping carriers (e.g., UPS, FedEx, DHL)

Justification: Each shipping carrier implementation should be substitutable without affecting the correctness of the system. For example, if the system expects to receive tracking information from any carrier, the implementation for UPS should behave similarly to the implementation for FedEx. This allows for seamless switching between carriers.

Interface Segregation Principle (ISP):

A client should not be forced to implement interfaces they don't use. This principle encourages the creation of small, cohesive interfaces rather than large, monolithic ones.

Interface Segregation Principle (ISP):

Use case: E-commerce Platform

Microservice: Product Catalog Service

Interfaces: ProductInformation, ProductSearch

Justification: Clients consuming the Product Catalog Service might only need to retrieve product information or search for products. By segregating the interfaces into smaller, focused ones, clients can implement only the interfaces they require, reducing unnecessary dependencies and potential coupling.

DIP: Microservices should depend on abstractions rather than concrete implementations. This allows for flexibility in choosing different implementations or technologies without impacting the overall architecture.

* Dependency Injection is a technique used to implement the Dependency Inversion Principle.

Dependency Inversion Principle (DIP):

Use case: E-commerce Platform

Microservice: Notification Service

Dependency: Email Service

Justification: Rather than directly depending on a specific email service implementation, the Notification Service depends on an abstraction (e.g., EmailSender interface). This allows for easy switching between different email service providers (e.g., SendGrid, AWS SES) without modifying the Notification Service itself. Additionally, it facilitates testing by enabling the use of mock implementations during testing.

The SOLID principles are a set of design principles intended to make software designs more understandable, flexible, and maintainable. When applied to microservices architecture, each SOLID principle can guide the design and implementation of individual microservices as well as the interactions between them. Here's how each SOLID principle relates to microservices:

1. Single Responsibility Principle (SRP):

- ✓ In microservices, each service should have a single responsibility or purpose. This means that each microservice should encapsulate a specific business capability or functionality.
- ✓ By adhering to SRP, microservices become more focused and easier to understand, maintain, and scale.

2. Open/Closed Principle (OCP):

- ✓ Microservices should be open for extension but closed for modification. This means that you should be able to extend the functionality of a microservice without modifying its existing codebase.
- ✓ OCP encourages designing microservices in a way that allows for easy addition of new features or changes in requirements without disrupting existing functionality.

Liskov Substitution Principle (LSP):

- ✓ In microservices, different implementations of the same service interface should be interchangeable without affecting the correctness of the system.
- ✓ LSP ensures that microservices can be replaced or upgraded with alternative implementations without causing unexpected behavior or breaking other parts of the system.

Interface Segregation Principle (ISP):

- ✓ Microservices should expose well-defined and focused interfaces that are tailored to the specific needs of clients.

- ✓ ISP encourages designing microservice interfaces that are cohesive and only expose functionality relevant to the clients that consume them. This helps reduce unnecessary dependencies and potential coupling between microservices.

5. Dependency Inversion Principle (DIP):

- ✓ Microservices should depend on abstractions rather than concrete implementations. This allows for flexibility in choosing and changing dependencies.
- ✓ DIP promotes designing microservices with loosely coupled dependencies, enabling easier management of dependencies and facilitating the replacement of dependencies with alternative implementations.

By applying the SOLID principles to microservices architecture, you can create a system that is more modular, maintainable, and scalable. Each microservice becomes a cohesive unit with clear boundaries and responsibilities, making it easier to develop, test, deploy, and evolve independently.

What is a Microservices Architecture in a Nutshell?

“single responsibility principle” which states “gather together those things that change for the same reason, and separate those things that change for different reasons.”

A microservices architecture takes this same approach and extends it to the loosely coupled services which can be developed, deployed, and maintained independently. Each of these services is responsible for discrete task and can communicate with other services through simple APIs to solve a larger complex business problem.