Introduction to Micro-Services with Spring Boot

Prakash Badhe
prakash.badhe@vishwasoft.in

About Me: Prakash Badhe

- Technology Practices Trainer, Mentor and Consultant.
- Technologies Experience for 20+ years
- Passion for technologies, tools and frameworks
- Proficient in application frameworks, libraries and tools.
- Proficient in DevOps tools and technologies.
- Proficient in Docker, Kubernetes etc.
- Supporting agile technical practices and setup for agile development and production environments.
- Worked with teams from Symantec,Samsung, PTC, Cognizant, Oracle,Persistent etc.

About you..

- **Prerequisite** : Experience with server side java applications development and deployment.
- Linux system commands exposure.
- Let us know about your
- **Objectives for this Training**
- **Job Role and skill-Set : In web application development.**
- **Experience in different Technologies and tools.**
- **Prior experience with Virtual machines, Docker or any other tools on Linux platforms.**

Setup

- Windows 10/Linux 64 bit
- Adobe PDF Reader
- JDK1.8 64 bit
- Spring Tool Suite
- Apache Kafka
- MySQL
- Live Internet connection

SOA : Service Oriented Architecture

Why SOA..?

- Distributed Processing..
- Applications need data sharing and communication in the network across different machines.
- Heterogeneous applications with different programming platforms and different Operating systems.
- Lots of data compatibility issues
- Standard protocol and data format needed.

Applications as services

- At abstract level if the applications are defined as platform independent services, then they can share the data and understand the data exchange...
- The SOA is the answer..
- Service Oriented Architecture.

Service Oriented Architecture

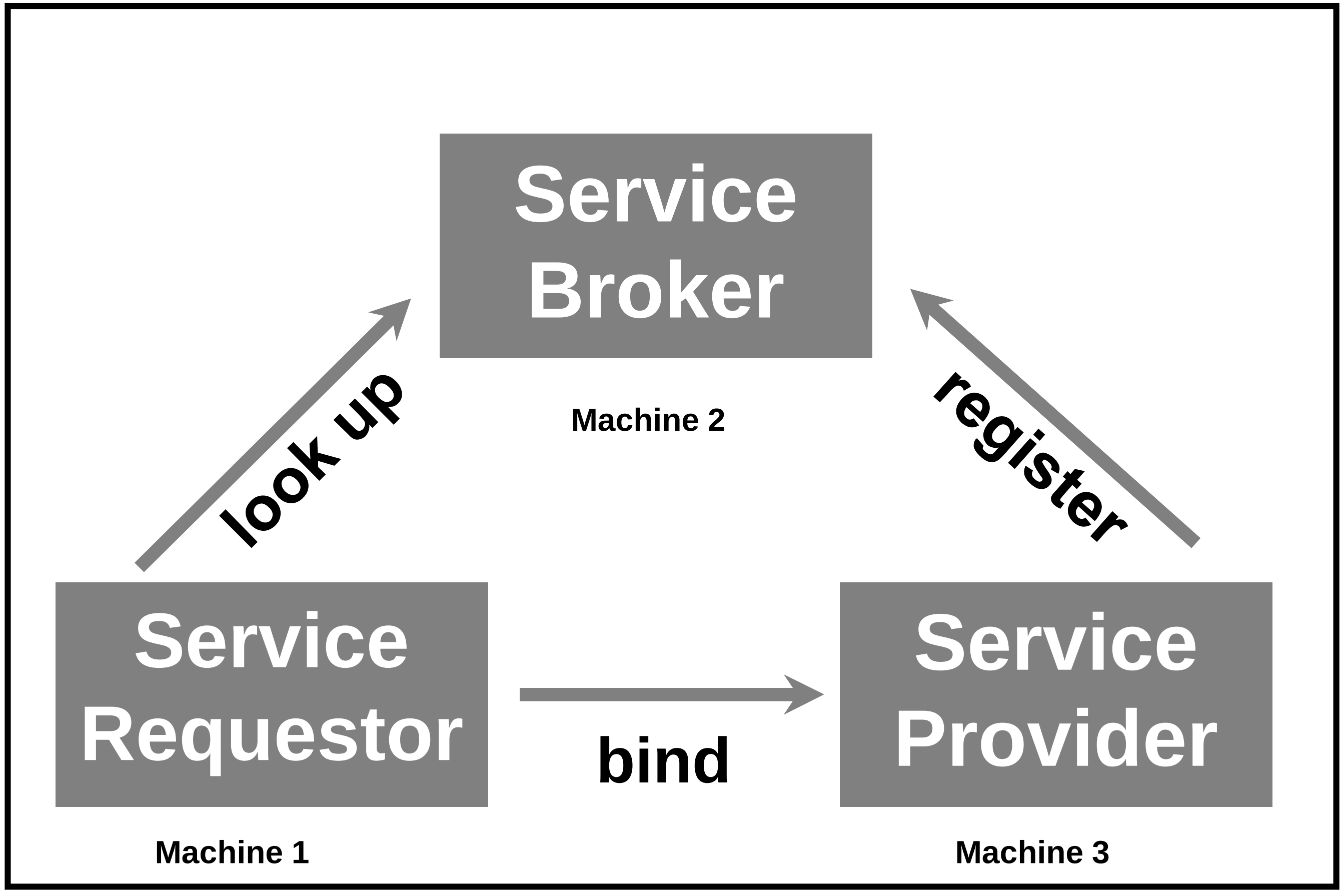
- In a service-oriented architecture, applications are made up from loosely coupled software services, which interact to provide all the functionality needed by the application.
- The applications exchange information across the platforms.
- Each service is generally designed to be very self-contained and stateless to simplify the communication that takes place between them.

Roles Involved In A SOA

Three main roles involved in a service-oriented architecture

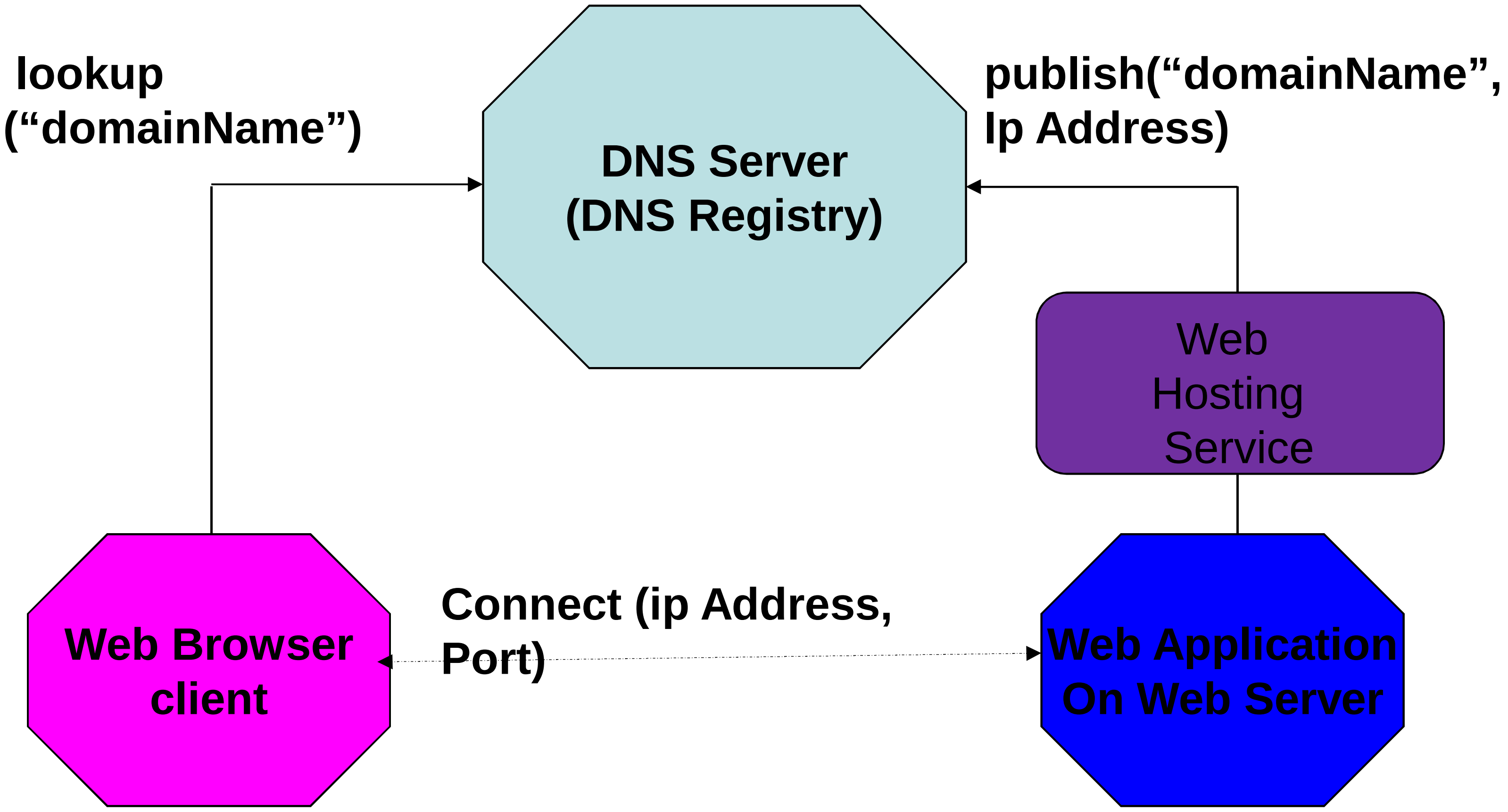
- **Service provider**
- **Service broker**
- **Service requestor**

SOA Communication Model

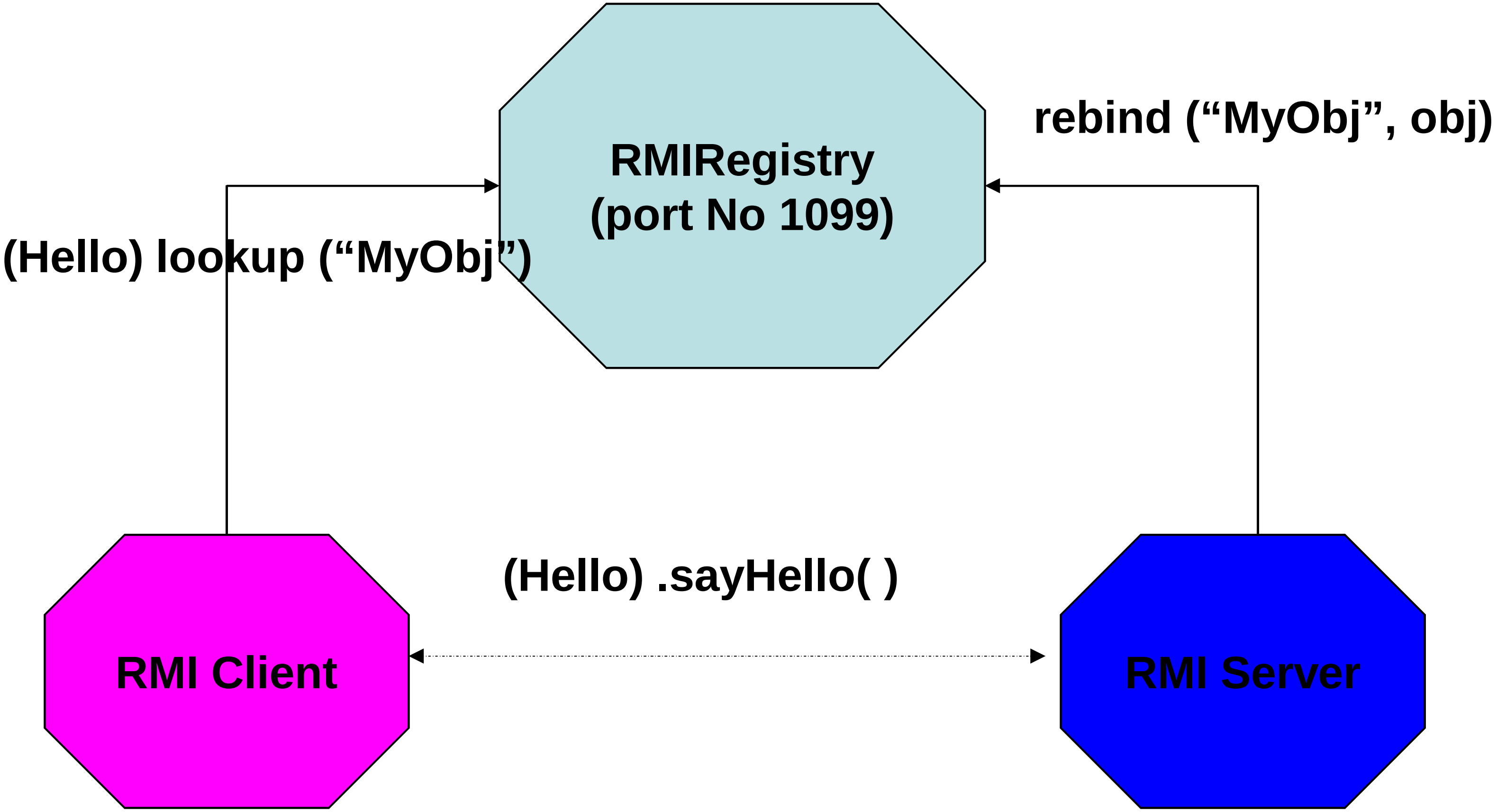


Service Oriented Architecture

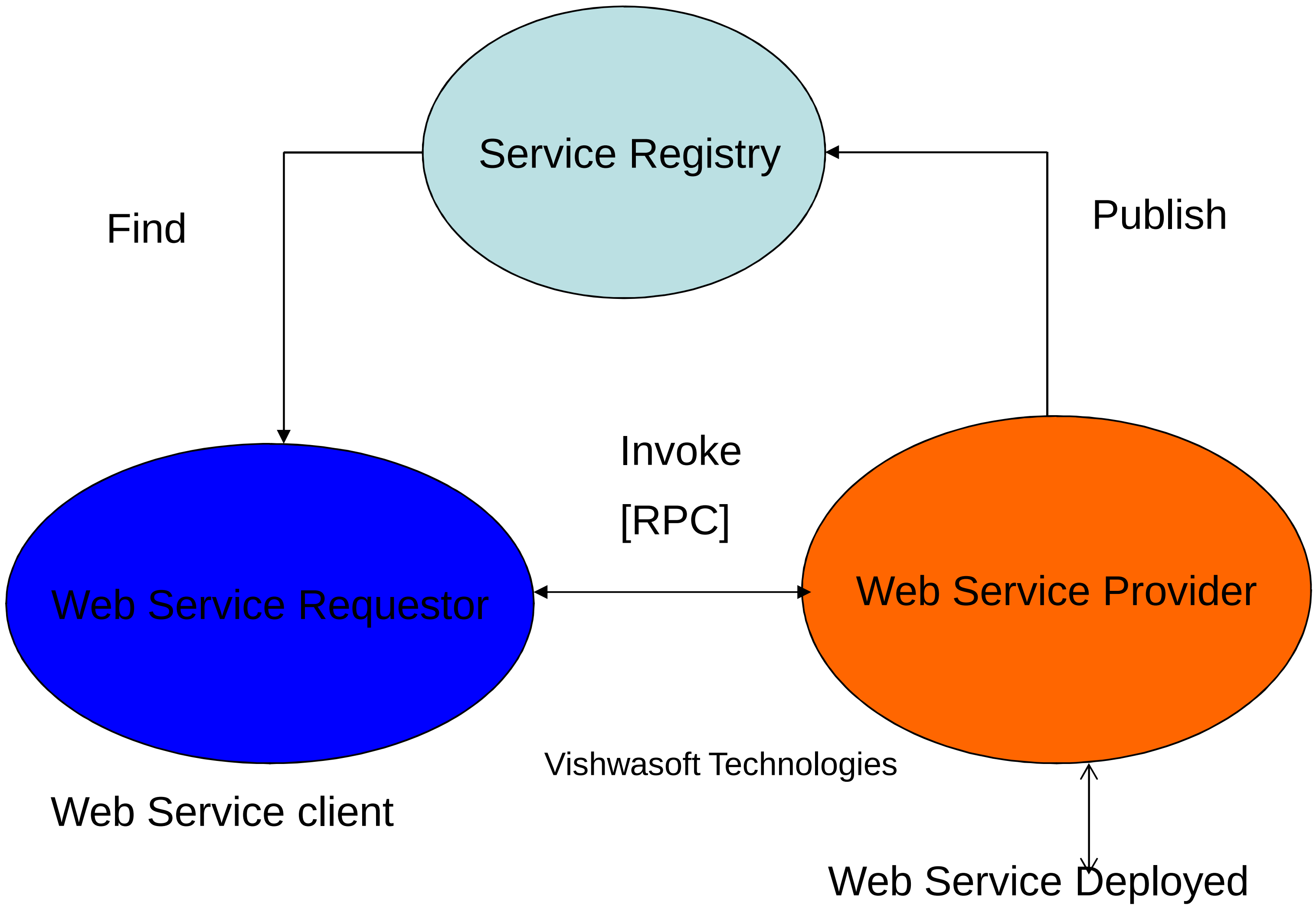
Web Application Process



RPC/RMI Architecture



Web Services Conceptually



Web Programming Model

- Involves client-server interaction on web.
- Uses http protocol and html for information exchange
- Exchange Messages that carry MIME-typed data
- Web applications are loosely coupled than the traditional distributed programming models like RPC, DCOM, and CORBA.

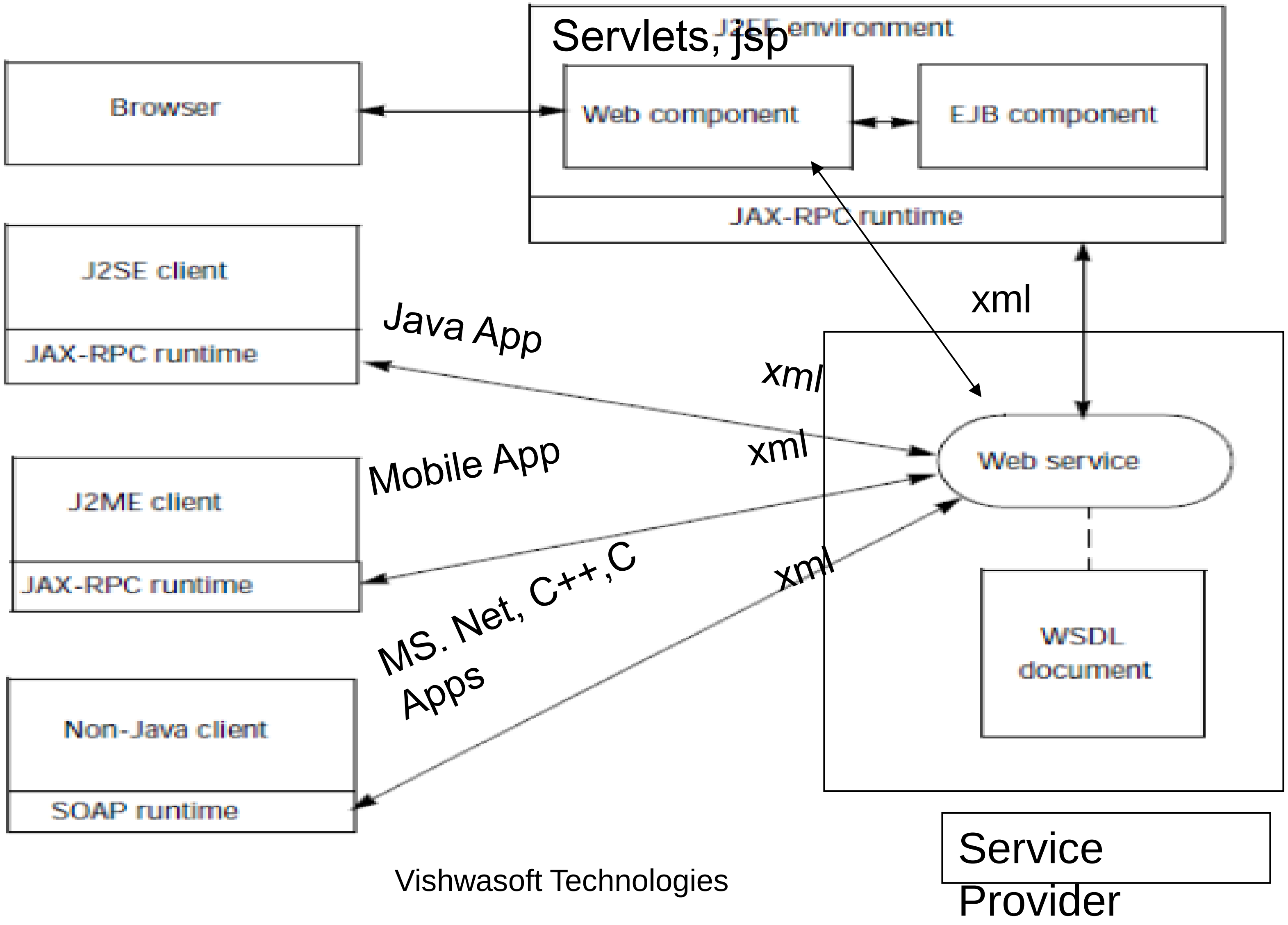
Web Service As SOA

- A Web service is a software application designed to support **interoperable** machine-to-machine interaction over a network
- *Generally web services are small task /service operations supported by applications on the server/s.*
- *These tasks are reusable across different OS platforms and programming language applications.*
- *Web services operate in a distributed environment.*

Web service features

- Loosely coupled Web programming model for use in applications that may not be browser-based.
- Provides a platform for building distributed applications using software components that are..
  - running on different operating systems and devices,
  - written using different programming languages and tools from multiple vendors,
  - all developed and deployed independently

Clients for Web Service



XML Usage

- Web Services fundamentally use XML to standardize the behaviors and data.
- Key XML standards to understand
  - XML : extensible mark up across programming languages.
  - XML Schema specifies xml structure.
  - XML Namespace avoids naming conflicts and categorizes elements as per the functionality

Web Service Standards

- SOAP
- – Simple Object Access Protocol for web service and clients to communicate.
- WSDL
- – Web Service Definition Language for web services to describe their services and other information.
- UDDI
- – Universal Description, Discovery, and Integration protocol for Web Services to discover web services.
- All these are specified in XML format.

Soap Web Services

- Manage language independent interactions across different platforms and applications
- Different versions evolved one by one to support additional features.
- Interoperability is still an issue across different implementations and versions.
- Heavy dependencies on the soap library and other xml implementations.
- Non-xml data format support is limited only in the form of attachments.
- Performance is an issue for mission critical applications.

RESTing the SOAP ?

Enter ‘REST’ Services

- REST is ‘Representational State Transfer’..
- REST is an architectural style rather than a protocol which removes the dependencies on soap and xml standards as wsdl,uddi etc.
- REST supports any understandable data formats across applications.
- REST works currently only with Http.
- REST specifies Resources/data on server rather than actions on them..
- REST specifies transfer of the state of Resource across applications.

Resources on the Web

Resources are not only represented as XML

- XML formats (HTML, XHTML, RSS, etc.)
- JPG, GIF, PNG
- MP3, WAV, OGG
- Anything else that can be on the web.

Resource Nouns

- Important ‘things’ (nouns) are Resources
  - Addressed through a URI
- Uniform interface (verbs)
  - In HTTP: GET, PUT, POST, DELETE
- Verb-noun séparation makes intégration easier
  - GET /customer/45 Instead of getCustomer(45)

REST Nouns and Verbs

- REST works with resources as nouns with their identities.
- The state of these nouns is shared with clients over http methods as verbs(operations)
- The http methods as verbs in REST
  - Get: get the resource state/values
  - Post : post new resource
  - Put : update the resource state
  - Delete :delete the resource on server
  - Options: read the options available with server.
  - Head : set the http headers on server.

REST Commandments

- Give every “thing” an ID
- Link things together
- Use standard methods
- Communicate statelessly

SOAP and REST

- REST is ready for the enterprise
- REST is strong at:
  - Internet scale computing
  - High levels of interoperability
  - Resource Oriented operations
- SOAP/WS is strong at:
  - Complex security (Trust and Federation)
  - Multi-transport services
  - Occasionally connected applications
- In the real world they are typically enabled by a combination of Soap and REST

Java support for REST

- Jax-RS is the java specification standard for implementing and consuming REST web services.
- Implementations a are Jboss RestEasy,Jersey platforms.
- Spring with Spring-Boot
- Oracle and IBM SOA Suite
- Eclipse Micro-Profile

SOA Applications

- Applications as services.
- Applications developed with different programming languages and running on different platforms able to communicate each other and exchange information/data.
- Applications as Services collaborate together as separate modules and share data.
- SOAP web services
- REST services
- Shared data as xml, json, text etc.

Multilayered Applications

- Presentation — responsible for handling HTTP requests and responding with either HTML or JSON/XML (for web services APIs).
- Business logic — the application’s business logic.
- Database access — data access objects responsible for access the database.
- Application integration — integration with other services (e.g. via messaging or REST API).
- Even with this modular layered architecture, the application is packaged and deployed as a single monolith. (All in One)

Monolithic applications

Monolithic application is a single-tiered software application where the user interface and data access code are combined into a single program running on a single platform as single process.

A monolithic application is self-contained and independent From other computing applications.

Monolithic application is designed without modularity.

Monolithic application is not scalable and difficult to maintain And develop with a team.

Monolith Benefits

- Simple to develop.
- Simple to test. For example you can implement end-to-end testing by simply launching the application and testing it.
- Simple to deploy. You just have to copy the packaged application to a server.
- Simple to scale horizontally by running multiple copies behind a load balancer.

Monolith Drawbacks

- Limitation in size and complexity.
- Application is too large and complex to fully understand and to be able to update with new changes fast and correctly.
- The size of the application can **slow down the start-up** time.
- The **response time and performance** becomes slow because of all execution happening in same process.
- The entire application has to be re-deployed on each update and during down time nothing of it can be accessible.

Monolith drawbacks..

- Difficult to manage team based development.
- Difficult to scale dynamically.
- Difficult to upgrade to new requirements
- Difficult for Continuous deployments
- Difficult to increase the scope
- Run time performance issues
- Crashes the entire application due to small bugs.
- Porting/migration issues

Monolith - Reliability

Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug impact the availability of the entire application

Scaling the Monolith

Challenging to scale when different modules have conflicting resource requirements.

Monolith- Sharing and Reuse

- The good parts of one application cannot be shared or re-used by other applications.
- Sharing of database for other processes and parallel updates is a challenge.

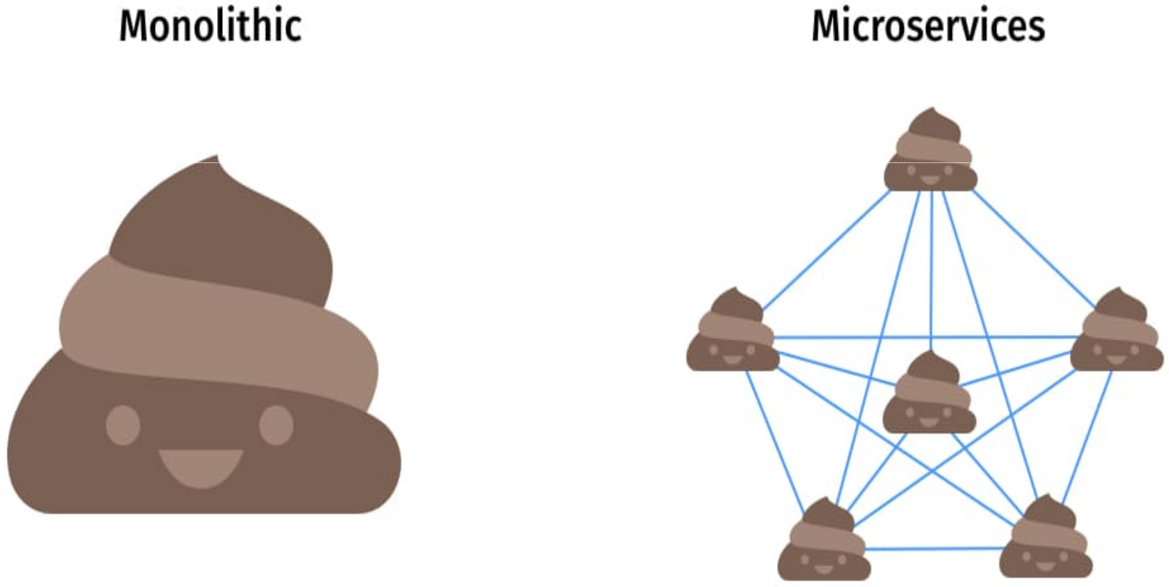
Break the Monolith…

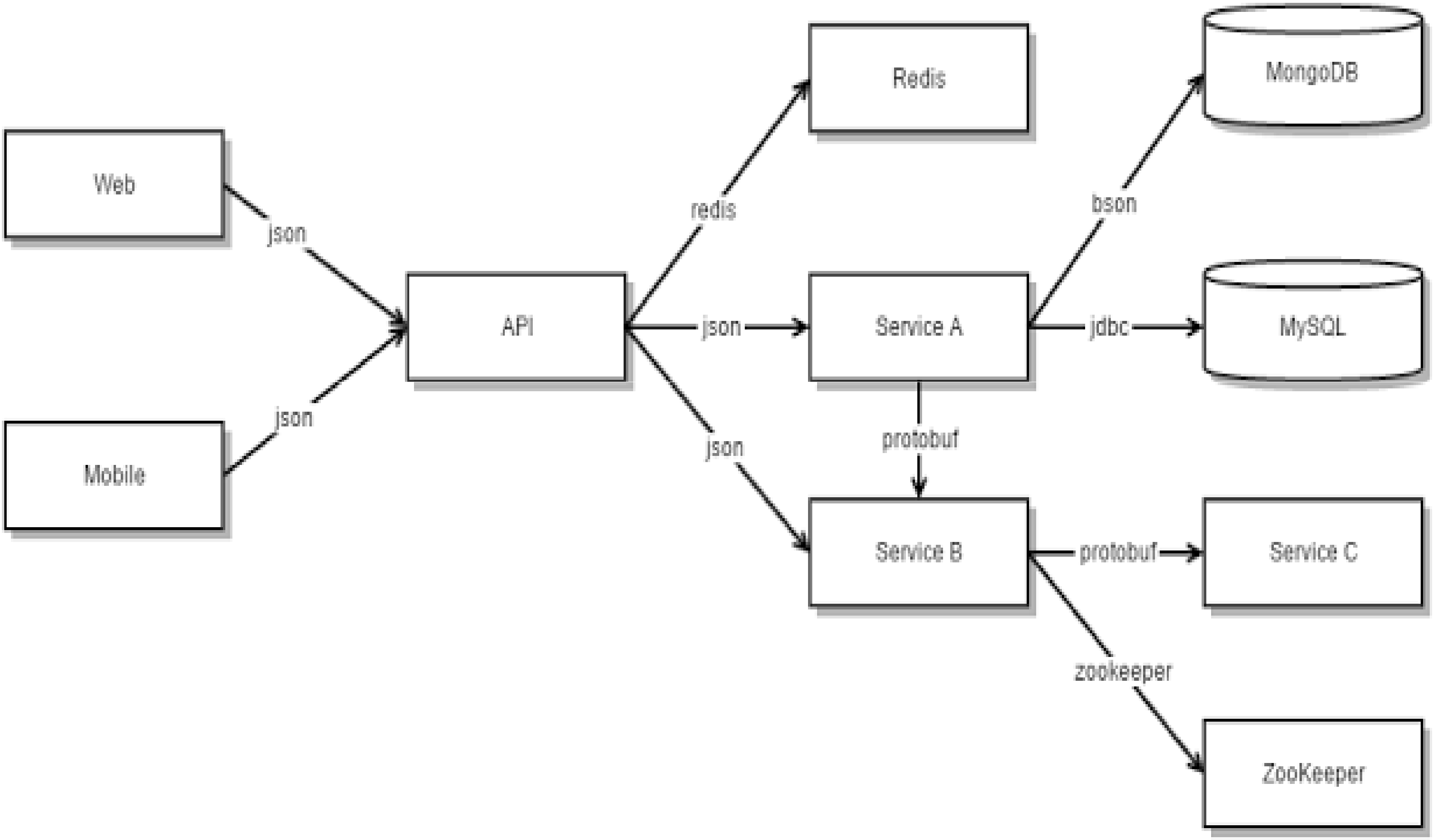
- The application is architecture is broken into a set of smaller, interconnected services (isolated processes) called as micro-services.
- Some micro-services expose a REST, RPC or message-based API and most services consume APIs provided by other services.
- Some micro-services might implement a web UI.
- The multiple modules work as service applications and collaborate together to form the application backbone.

..into MicroServices

- MicroService is an architecture pattern for service applications to collaborate together .
- Suitable to develop across team of developers.
- Easier to develop with smaller modules and smaller teams as well.
- Highly testable because of loosely coupling with other services/modules
- Easier to scale
- Easier to troubleshoot
- Flexible in code maintenance.
- Independently deployable units.

Monolith to Micro-Services




Web
json
Mobile
json
API
redis
json
json
Redis
Service A
protobuf
Service B
bson
jdbc
protobuf
zookeeper
MongoDB
MySQL
Service C
ZooKeeper

MicroService Benefits

- MicroServices reinforce modular structure, which is particularly important for larger teams.
- Services are easier to deploy, and since they are autonomous, are less likely to cause entire system failures when they go wrong.
- With MicroServices you can mix multiple technologies, languages, development frameworks and data-storage technologies.
- Reduced complexity.
- Each MicroService is to be deployed independently. As a result, it makes continuous deployment possible for complex applications

MicroService Benefits

- MicroService architecture enables each service to be scaled independently.
- Performance is improved.
- Better testability — services are smaller and faster to test.
- Better deploy-ability — services can be deployed independently.
- Improved fault isolation; for memory leak in one service then only that service is affected.

MicroService Challenges

- Manage the complexity of creating a distributed system.
- Manage the inter-service communication mechanism and deal with failures.
- Implementing requests spanning across multiple services is more difficult.
- Testing the services interactions is tedious.
- Troubleshooting at High level with integration issues .
- Increased resource consumption as memory , CPU since each service is a separate isolated process.

Cost of MicroServices

- Distributed systems are harder to implement, since remote calls are slow and are always at risk of failure. Uniform programming model can reduce the difficulties.
- Maintaining strong consistency is extremely difficult for a distributed system, which means everyone has to manage eventual consistency. With high availability nodes in clusters, this effect can be reduced.
- Need a mature operations team to manage lots of services, which are being redeployed regularly, with automation CI/CD this can be eased.
- The network, technology, bandwidth limits the performance and flexibility.

MicroService Design Principles

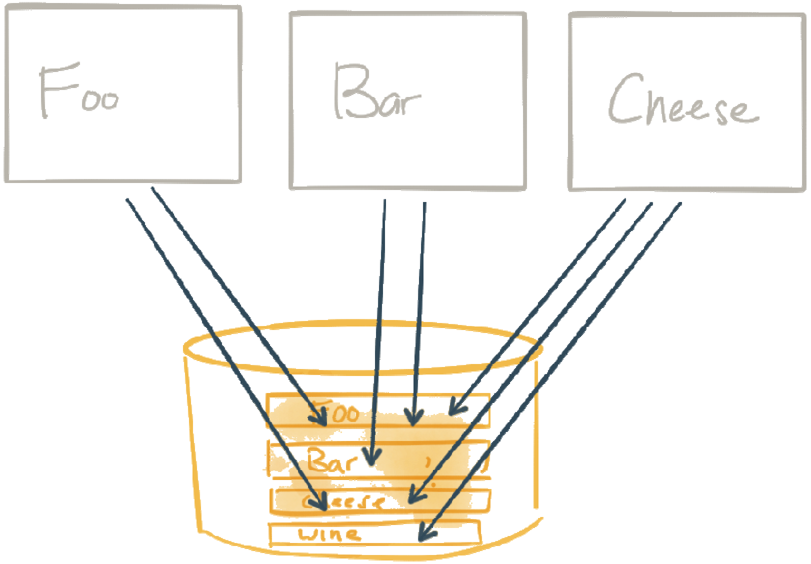
- ✓Scalability
- ✓Availability
- ✓Resiliency: ability to recover quickly from difficulties
- ✓Flexibility
- ✓Failure isolation
- ✓Loose coupling
- ✓Independent, autonomous
- ✓Decentralized
- ✓Failure isolation
- ✓Auto-Provisioning of Nodes
- ✓Continuous Delivery

Decompose by…

- Decompose by business capability and define independent services corresponding to business capabilities.
- Decompose by domain-driven design paradigm with isolated model design.
- Decompose by verb/action or use case and define separate services that are responsible for particular actions.
- Decompose by nouns/resources by defining separate a service that is responsible for all operations on particular entities/resources of a given type.e.g. Account, Users etc.
- Every service should have only a small/single set of responsibilities

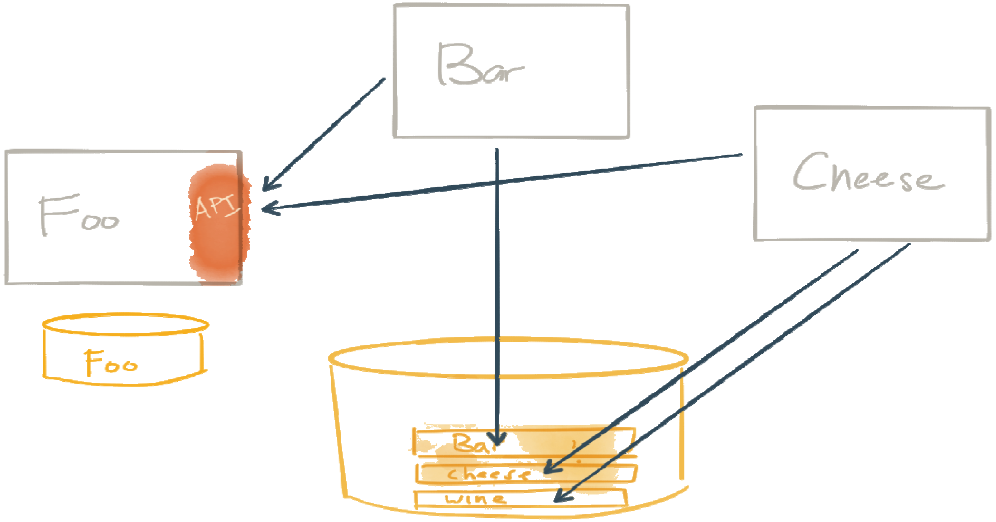
How to de-compose

Identify Modules by domain, categories, business use, related dependencies.

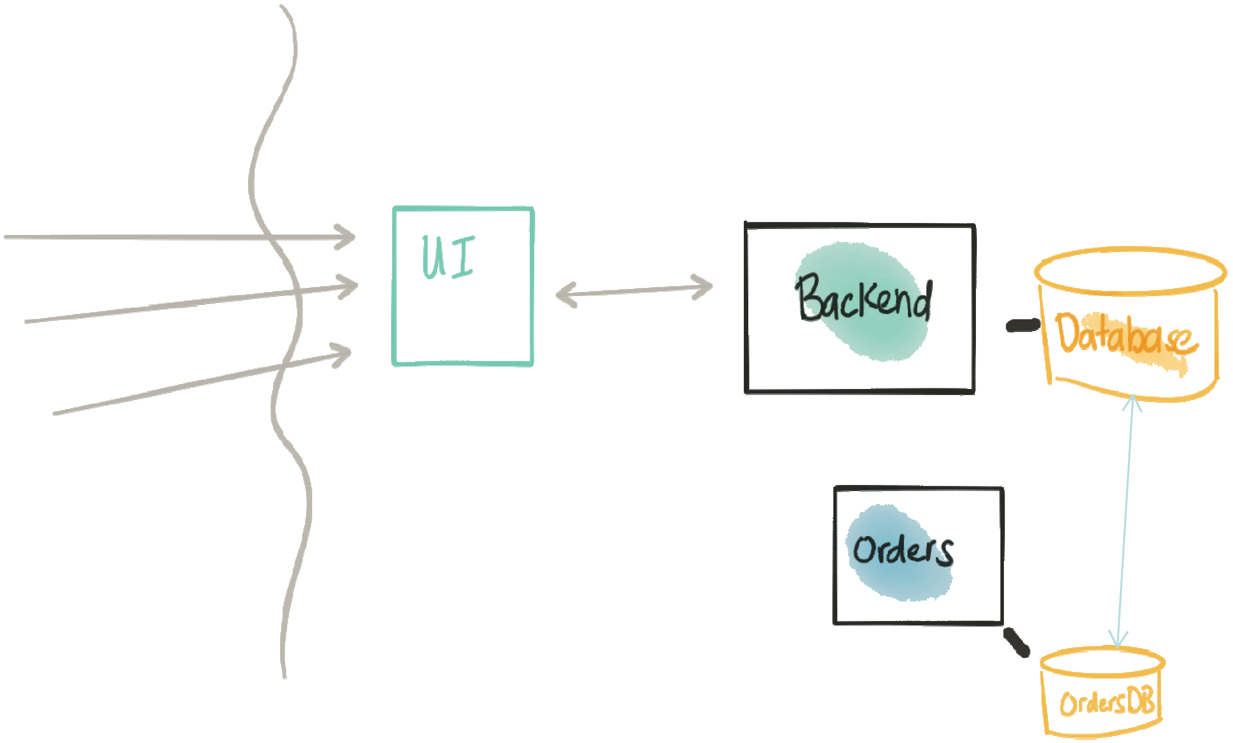


Database break-up

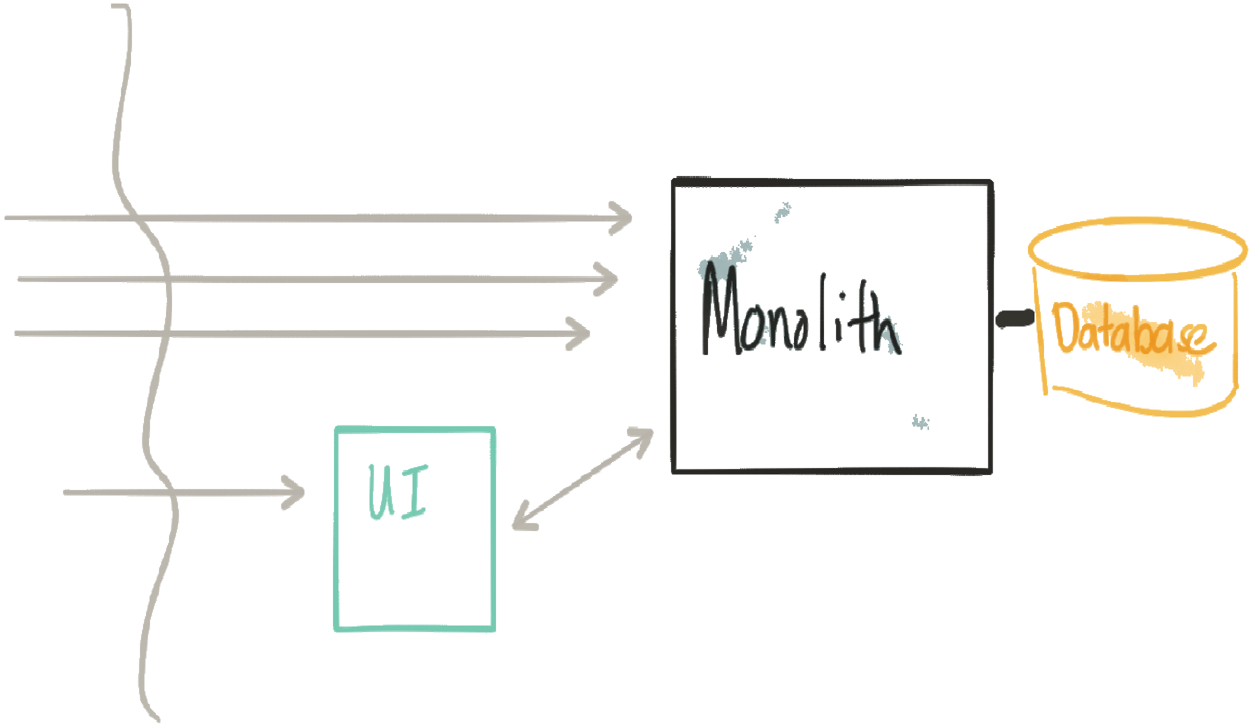
Break out database tables, wrap with service and, update dependencies



Service Modules



UI Extraction



Data consistency

- To ensure loose coupling, each service has its own database.
- Maintaining data consistency between services is a challenge since the 2 phase-commit/distributed transactions is not an option for many services.
- The services can use shared database.
- In certain Enterprise applications, the replication across the database instances is done in background to maintain consistency

Sharing of Database

- Instead of sharing a single database schema with other services, **each service has its own database schema.**
- **T**this can result in duplication of some data. However, having a **database schema per service is essential** to get the benefit from micro-services, because it **ensures loose coupling**.

Infrastructure for MicroServices

- The environment for MicroServices deployment and execution.
- API Gateway for Request-Response routing and security authentication.
- Load balancing : dispatch the requests to available nodes by calculating the service load or in round robin manner.
- Dynamic service discovery of services independent of the location/ip address by implementing service registry like domain name service (DNS) for web applications.
- Should support service health monitoring API points.

Infrastructure support...

- The fault tolerance features like Circuit Breaker, Bulkhead, Retry, Timeout, Fallback etc.
- Middleware for integration, Cache etc.
- Transporters for node to node communication in a cluster.
- Should support documentation about the services and their external APIs and data formats etc.
- Data Serializers and de-Serializers with custom schemas.

MicroService Platforms

- Spring with Spring Boot (Development and deployment)
- Spring Boot with Spring Cloud and Netflix(Deployment)
- Cloud Foundry supports multi-cloud application platform as a service (PaaS) governed by the Cloud Foundry Foundation.
- Spring Cloud with Cloud-foundry
- Eclipse MicroProfile.(development and deployment)
- Microsoft .NET platform for API(Develop and deploy)
- AWS Cloud , Google CE, Azure, OpenShift (RedHat) with Kubernetes as deployment platforms.
- Lagom MicroService platform

Spring Boot Features

- Quick and flexible development of services
- Generate API documentation with Swagger.
- Supports heath monitoring points with Actuator.
- The logging and tracing

Deployment Environments

- NetFlix
- Spring Cloud
- Google Cloud, AWS, Azure
- Moleculer Framework based on Node JS

Integration Patterns

- Aggregator Pattern
- Splitter
- UI separation
- Monitoring patterns
    - Log Aggregation
    - Performance Metrics for analysis
    - Distributed Tracing
    - Health Check
- Command Query Responsibility Segregation(CQRS)
- Asynchronous processing with Message Brokers.
- Event sourcing

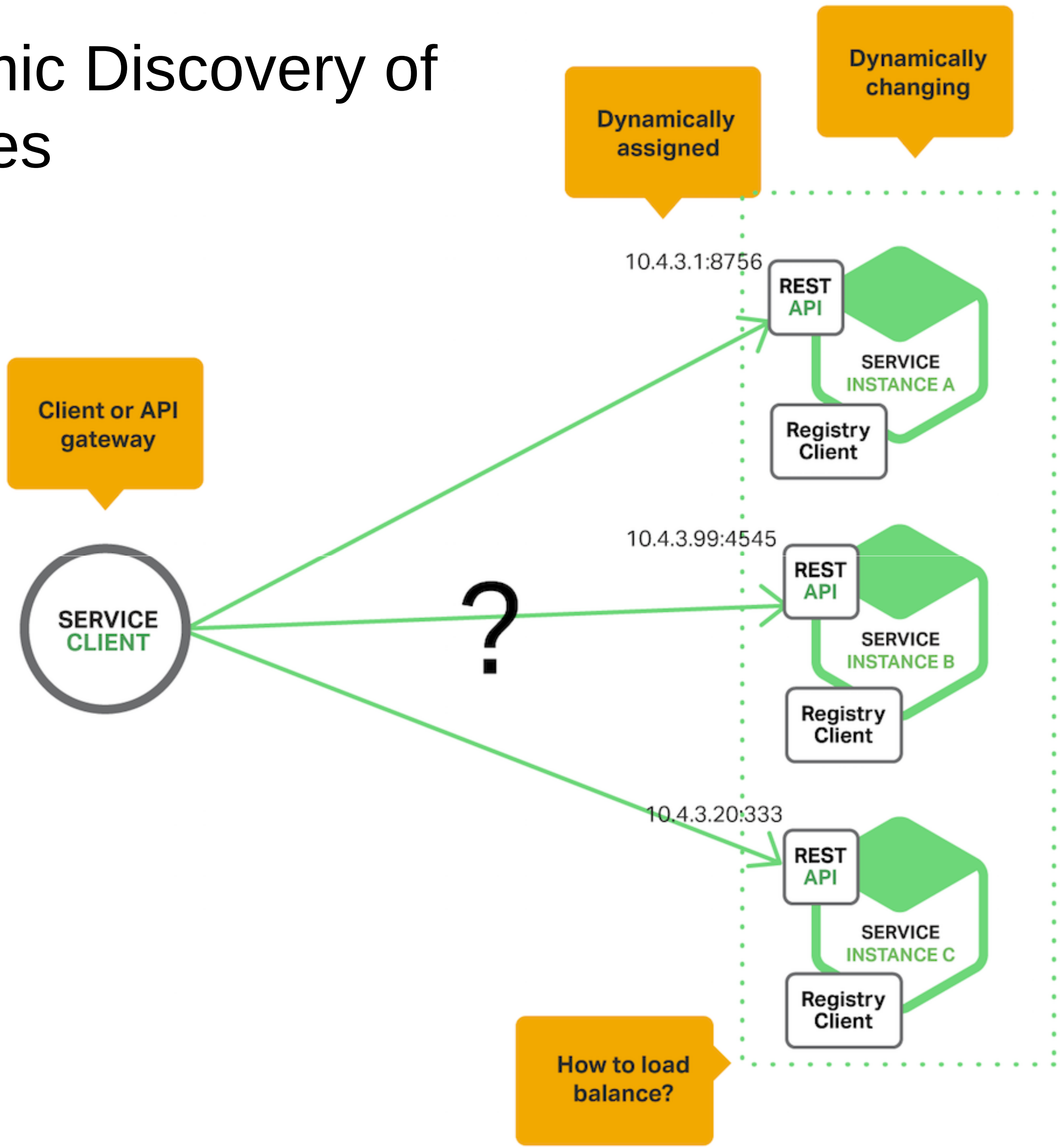
Deployment Patterns

- **Externalized Configuration**
- **Dynamic Service Discovery**
- **API Gateway**
- **Circuit Breaker**
- **Blue-Green Deployment**
- **Saga for consistency**
- **Load balancing**
- **High availability**

Service Registry and services

- Services should be isolated and independent.
- Services should NOT have hard coded references to other services.
- **Services should be able to register and discover from external Service Registry environment**.

Dynamic Discovery of services



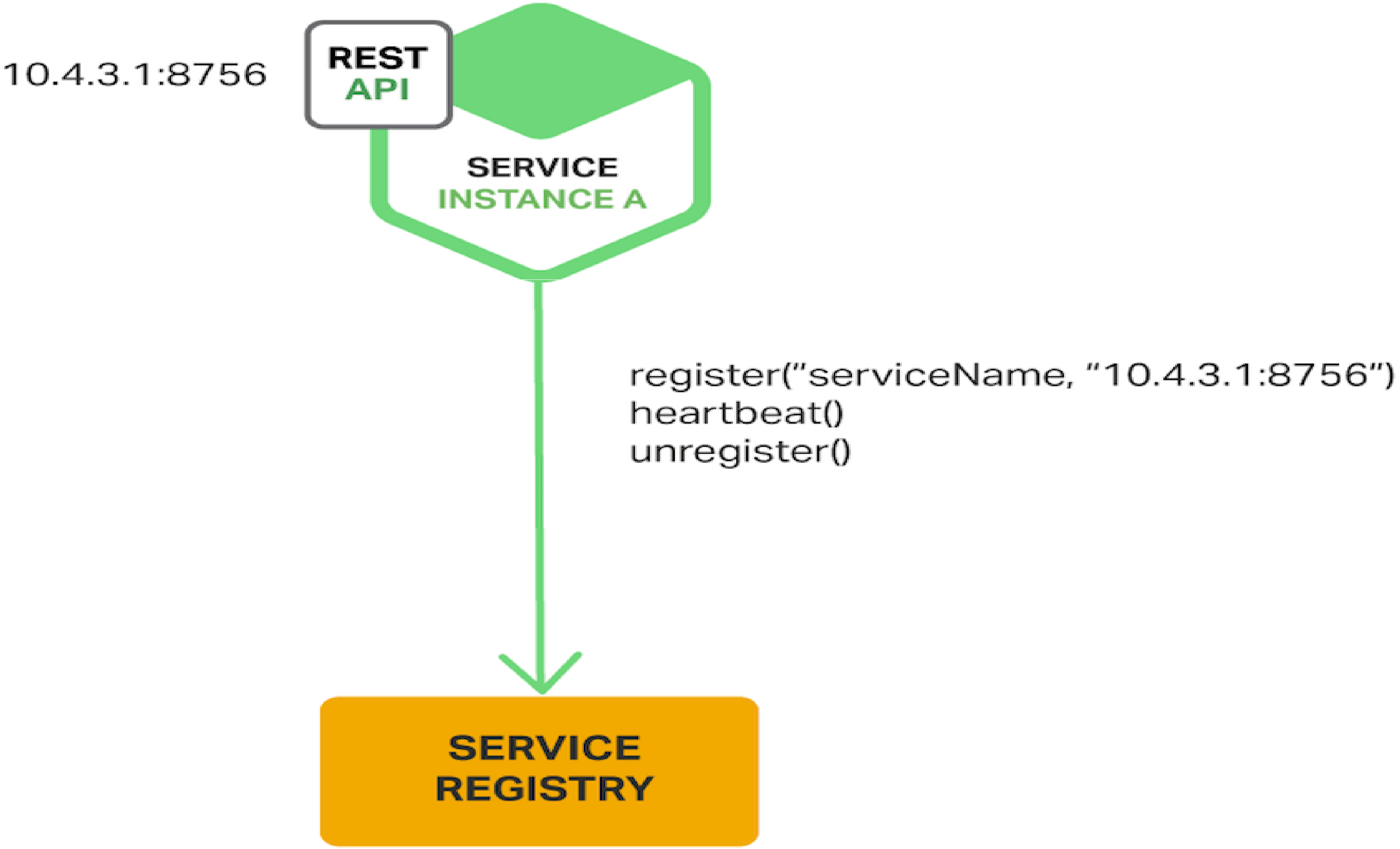
Dynamically changing

Dynamically assigned

Client or API gateway

How to load balance?

Self Registration



10.4.3.1:8756

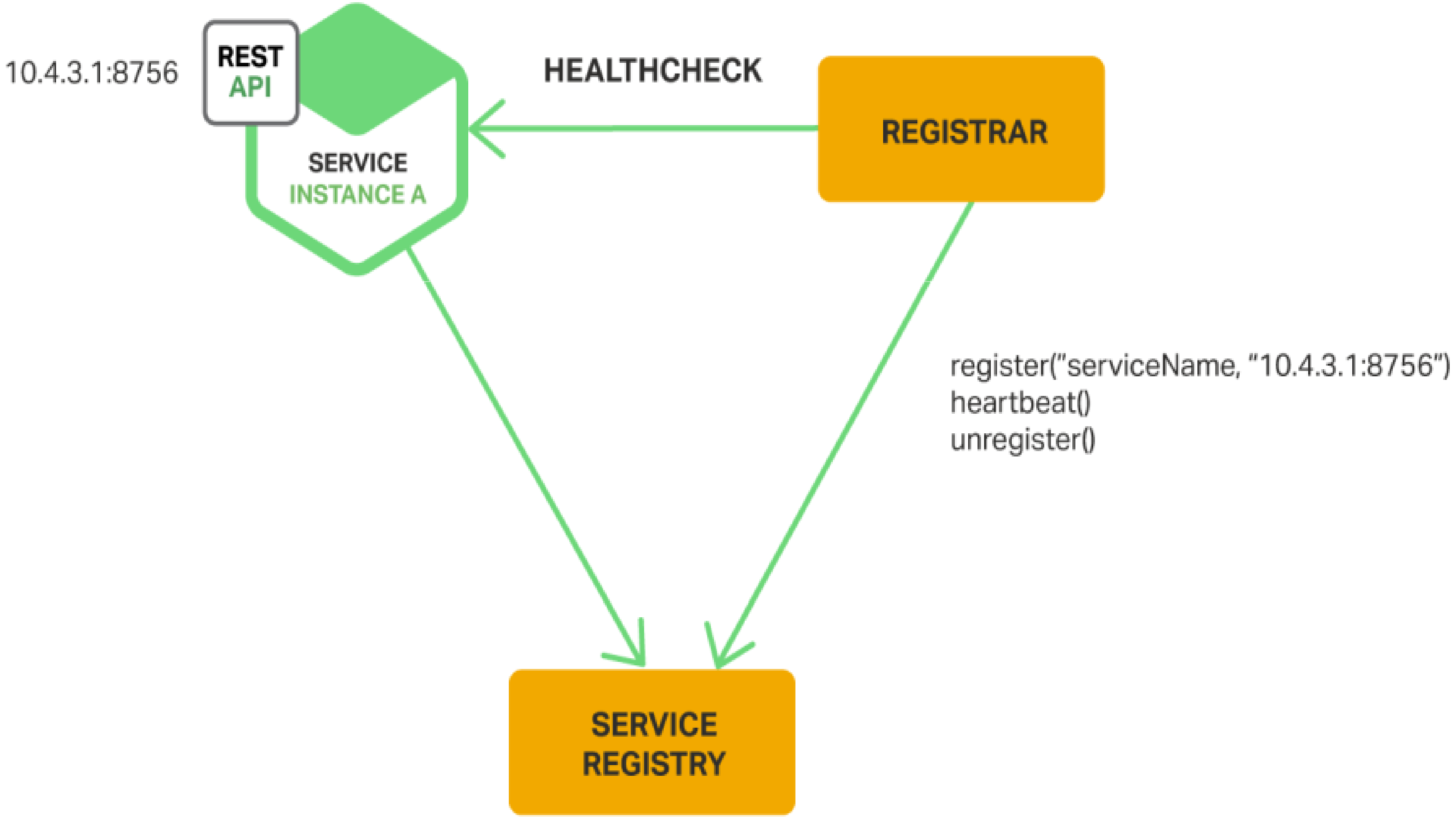
REST API

SERVICE INSTANCE A

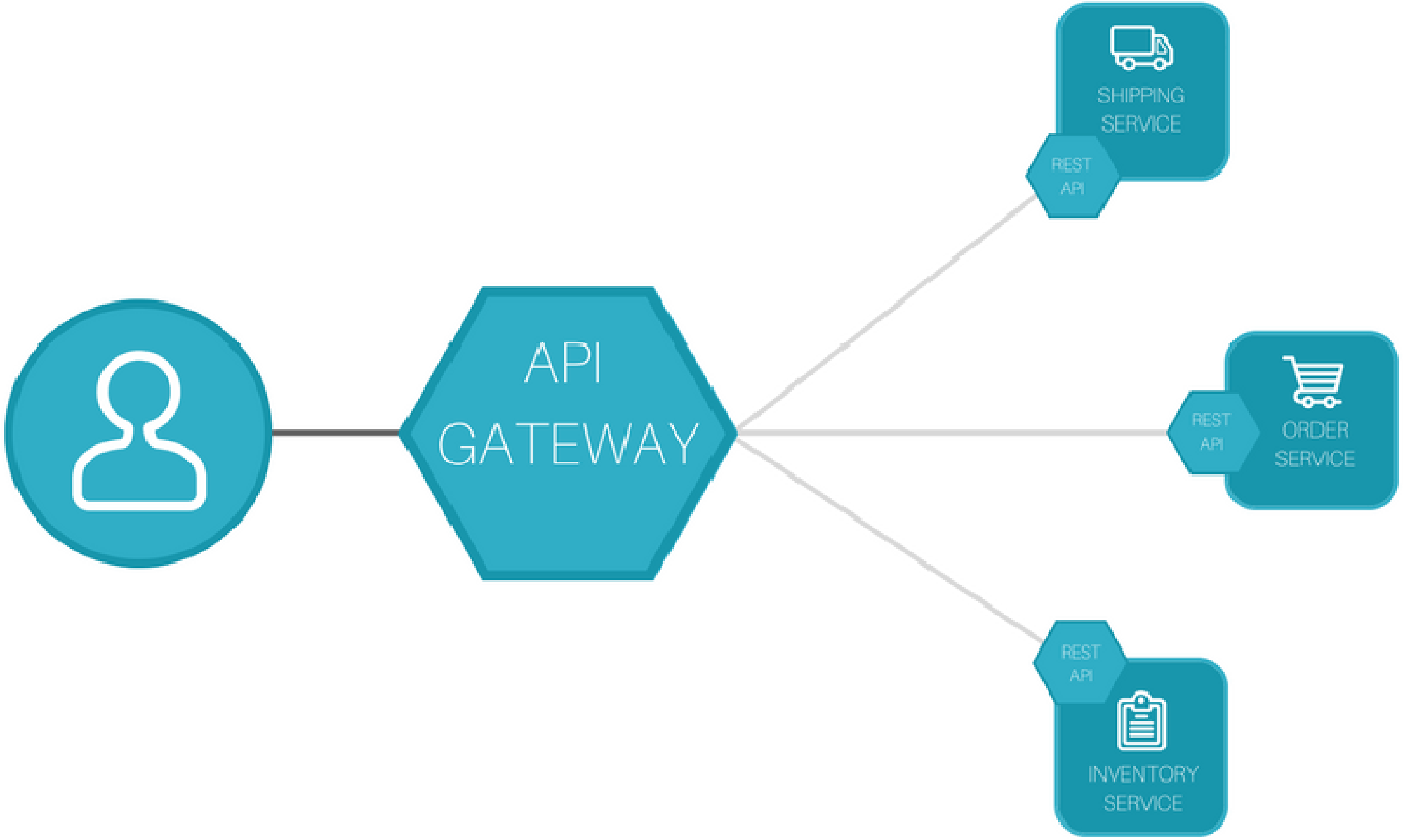
register("serviceName, "10.4.3.1:8756")
heartbeat()
unregister()

SERVICE REGISTRY

Third Party Registration



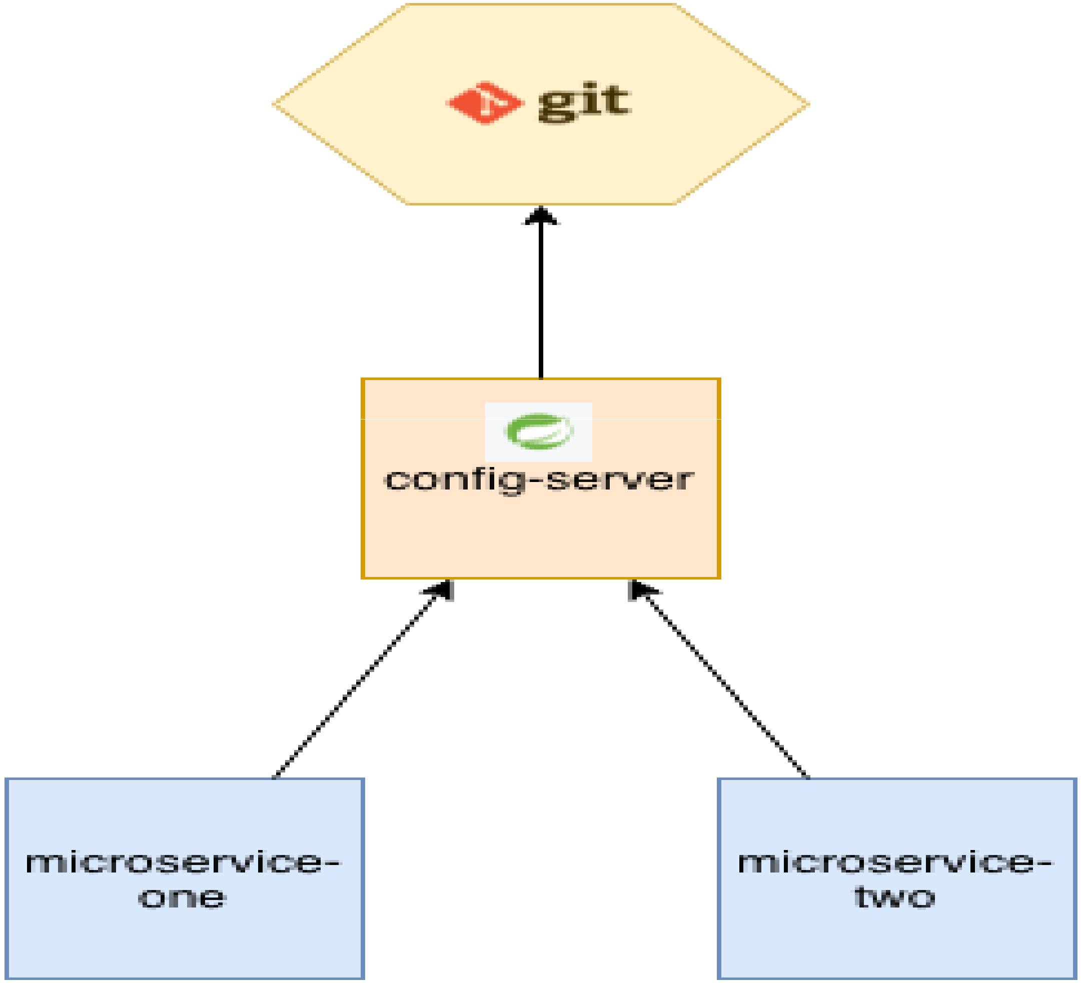
API Gateway


API
GATEWAY
SHIPPING
SERVICE
REST
API
ORDER
SERVICE
REST
API
REST
API
INVENTORY
SERVICE

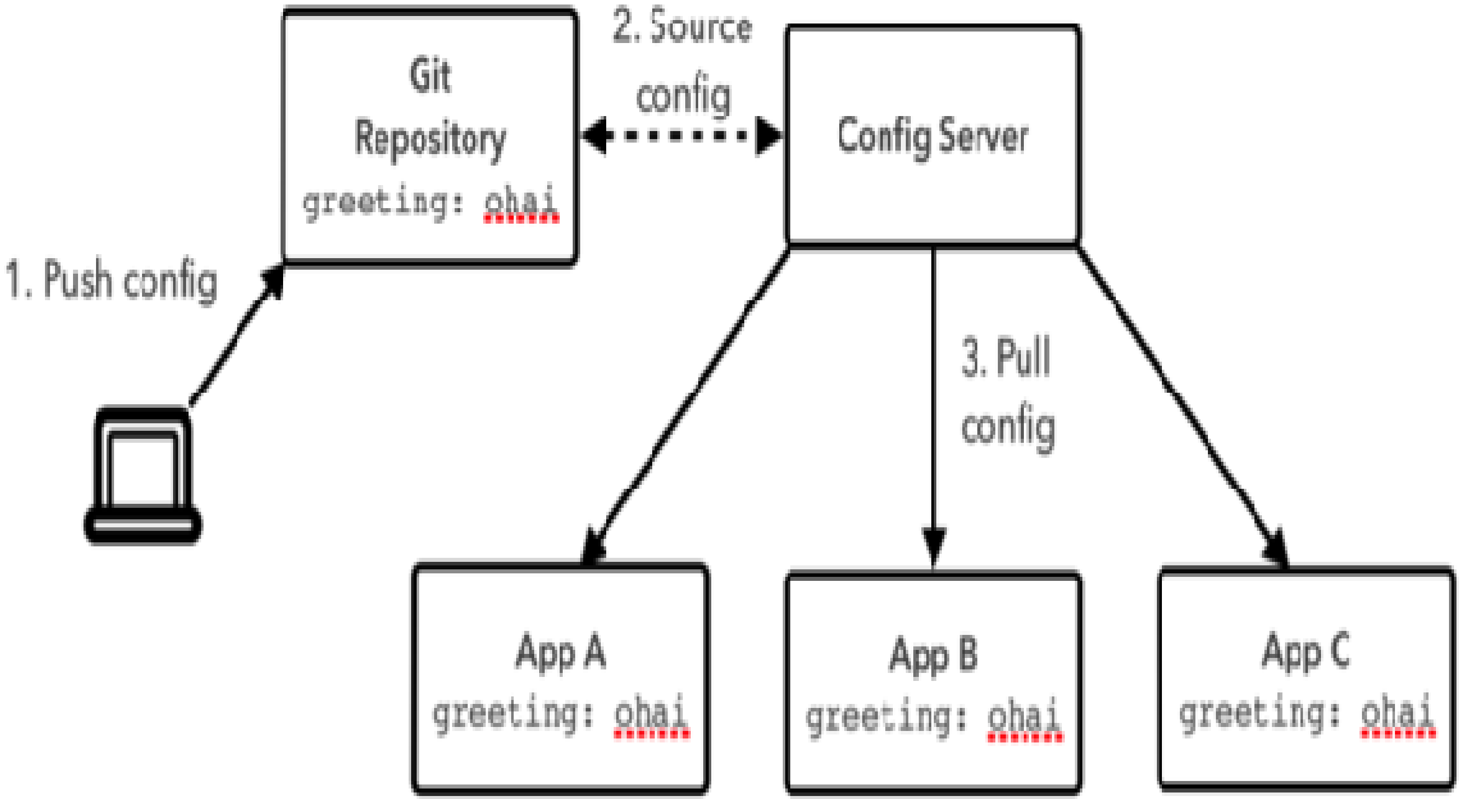
API Gateway Responsibilities

- Routing of requests to specific machine/node
- Security authentication and authorize for access control
- Load balance the requests across nodes/machines

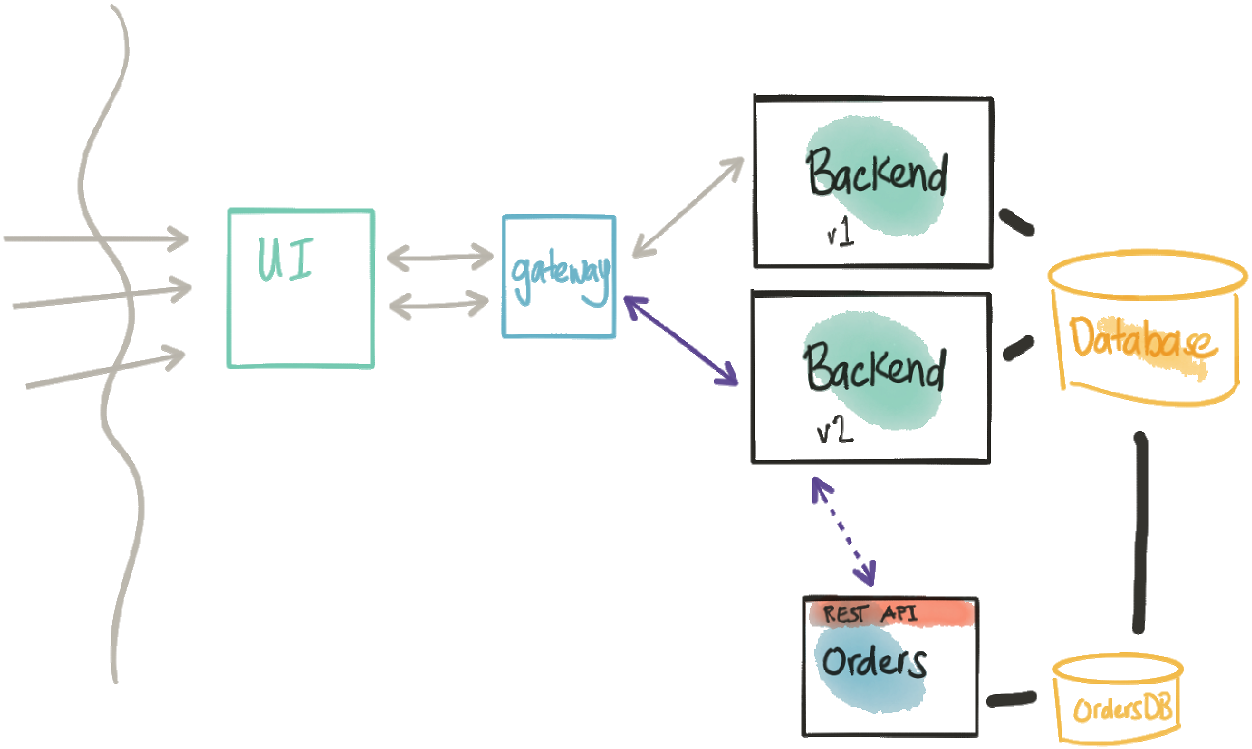
Externalized Configuration



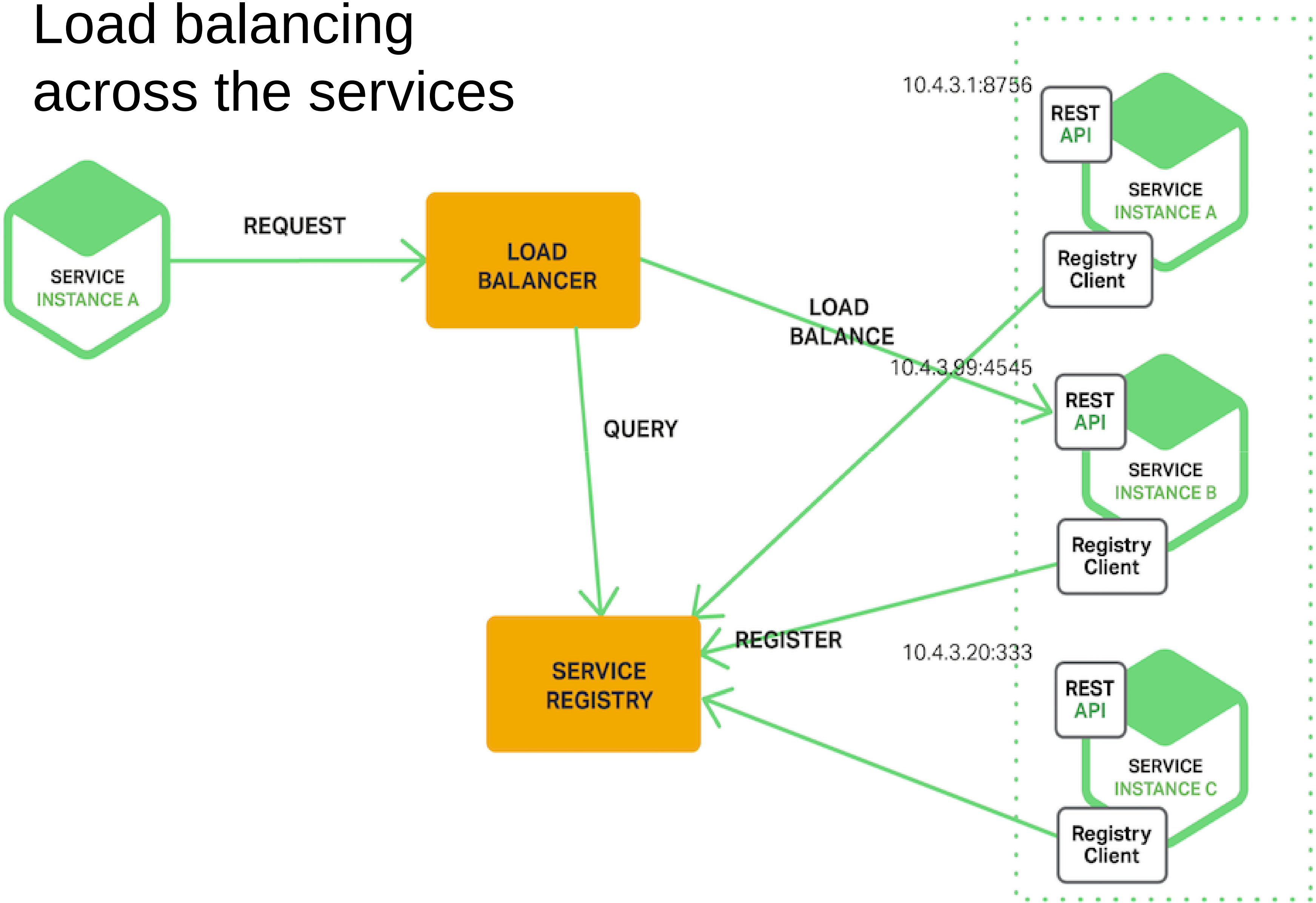
Getting the configuration



Load Balancing



Load balancing across the services



Algorithms for Load balancing

- Dispatch across nodes in Round Robin manner.
- Weighted dispatch depending on weight of the node calculated based on capacity, request load etc.
- Sticky node etc.

Need of communication

- The services collaborate to handle client requests.
- The services should use an inter-process communication mechanism and agreed data structure of information exchange.
- The synchronous communication between the services results in tight runtime coupling, both the client and service must be available for the duration of the request and creates performance bottlenecks.

Message Broker

- The services use **asynchronous** messaging for inter-service communication.
- Services communicate by exchanging messages over messaging channels.
  - Request/response mode - a service sends a request message to a recipient and expects to receive a reply message promptly
  - Notifications - a sender sends a message a recipient but does not expect a reply. Nor is one sent.
  - Request/asynchronous response - a service sends a request message to a recipient and expects to receive a reply message eventually.

Messaging over channel

- Publish/subscribe - a service publishes a message to zero or more recipients through common message destinations
- Publish/asynchronous response - a service publishes a request to one or recipients, some of whom send back a reply.
- Asynchronous messaging brokers
  - Apache Kafka
  - RabbitMQ
  - Apache ActiveMQ
  - JMS Messaging

Messaging benefits and Issues

- Loose runtime coupling between services since it decouples the message sender from the consumer.
- Improved availability since the message broker buffers messages until the consumer is able to process them
- Supports a variety of communication patterns including request/reply, notifications, request/async response, publish/subscribe, publish/async response etc.
- **This adds additional complexity to message broker, which must be highly available.**

Need of Atomic Updates

- Service to update the database **and** send messages/events to message broker/system.
- The database update and sending of the message must be atomic in order to avoid data inconsistencies and bugs. E.g if the database update fails, message update should roll back or so.
- One of the option is a distributed transaction (like JTA) that spans the database and the message broker to atomically update together, but may be time consuming and cause performance bottlenecks.

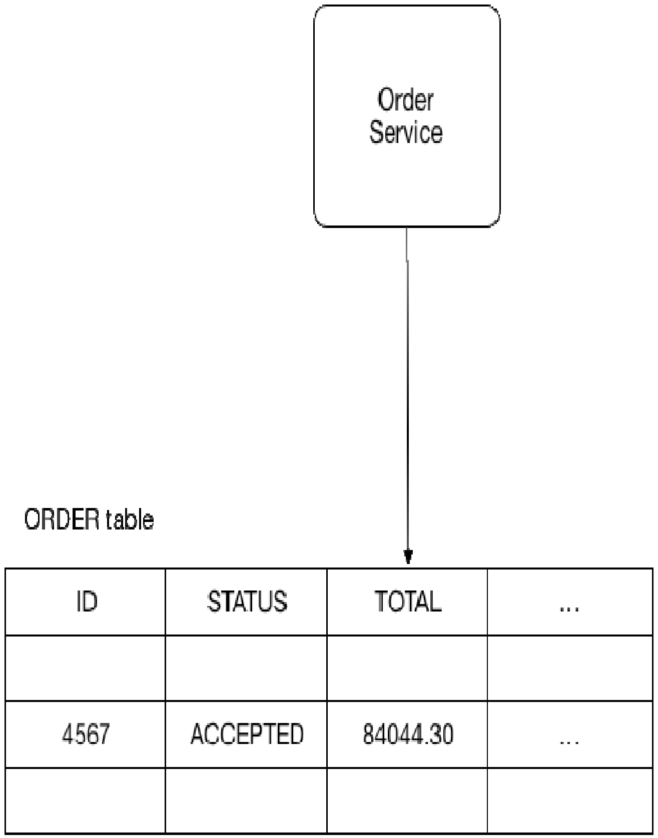
Event Sourcing

- To reliably/atomically update the database and publish messages/events.
- Event sourcing saves the state of a business entity such as Order or a Customer as a sequence of state-changing events.
- Saving an event is a single and inherently atomic operation.
- The application can reconstructs an entity’s current state by replaying the events.

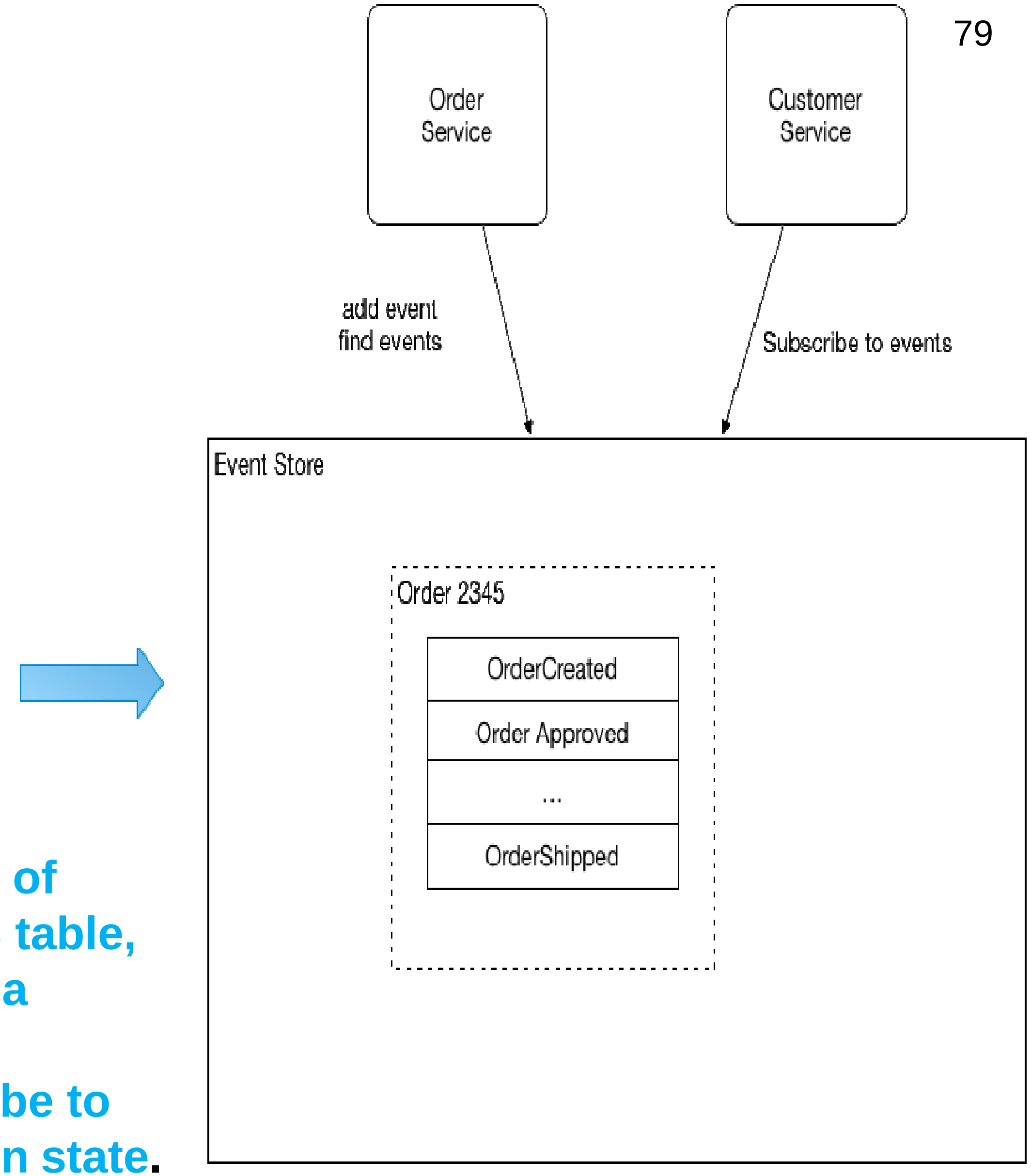
Event Sourcing in Action

- Persist events in an event store, which is a database of events.
- The store has an API for adding and retrieving an entity’s events and the event store also behaves like a message broker.
- It provides an API that enables services to subscribe to events.
- When a service saves an event in the event store, it is delivered to all interested subscribers.

Event Sourcing with Message Broker



ID	STATUS	TOTAL	...
4567	ACCEPTED	84044.30	...



Instead of storing the current state of each order as a row in an ORDERS table, the service persists each Order as a sequence of events.
The CustomerService then subscribe to the order events and update its own state.

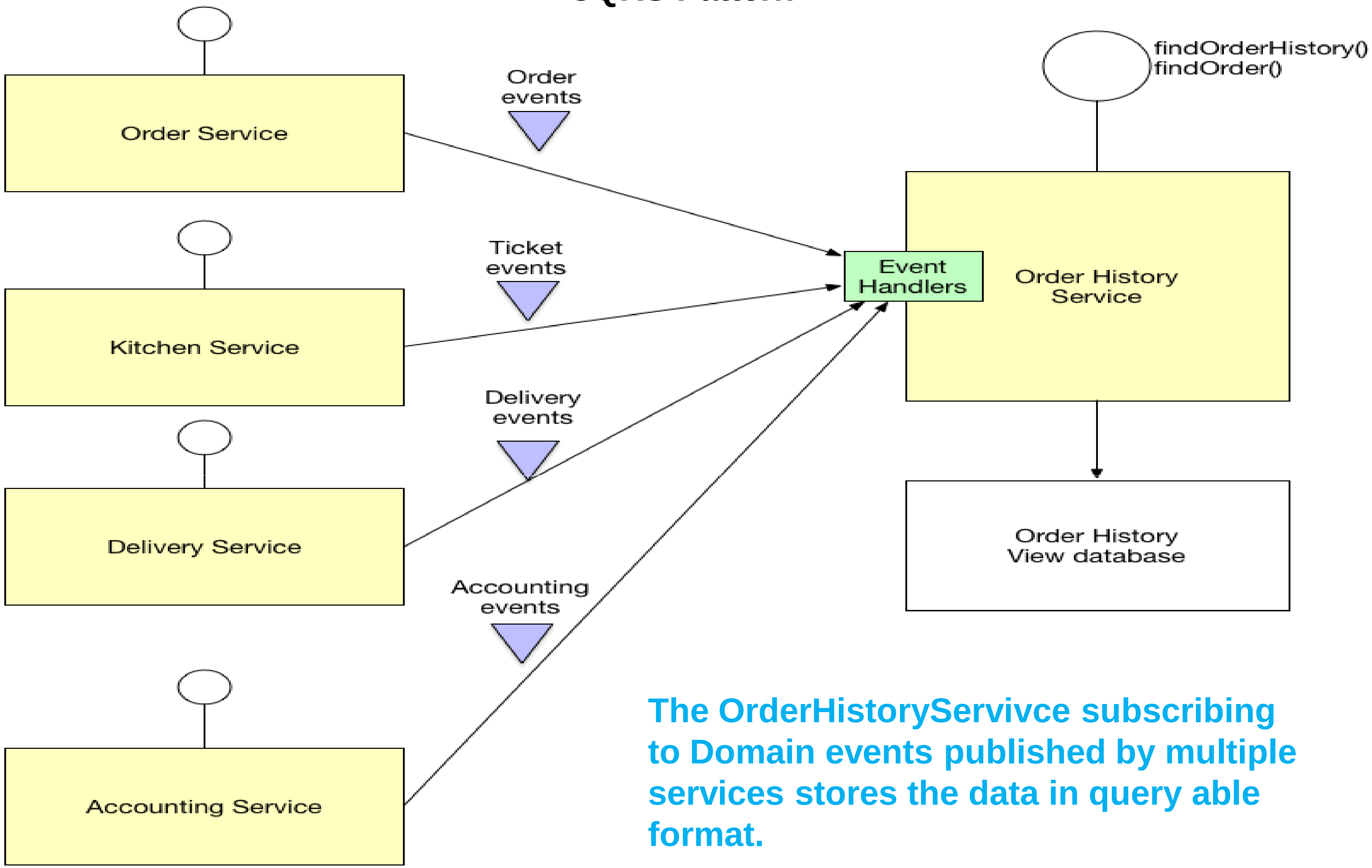
Query with the data

- To implement queries that joins data from multiple services.
- In case of Event sourcing pattern then the data is no longer easily query-able.
- A query system that enables to retrieve data from multiple services in a MicroService architecture.

Command Query Responsibility Segregation (CQRS)

- Multiple services publish data as events to event source using event handlers.
- The event handlers are subscribed by single Event Source service.
- The view database - a read-only (for faster queries) supports the aggregate join queries.
- The Event Source service application keeps the data upto date by subscribing to Domain events published by the services that owns the data.

CQRS Pattern



The OrderHistoryServivce subscribing to Domain events published by multiple services stores the data in query able format.

Vishwasoft Technologies

Handle Error Gracefully

- When services synchronously invokes another there is always the possibility that the other service is unavailable or is responding with high delay.
- The resources such as threads might be consumed in the caller while waiting for the other service to respond and this leads to resource exhaustion, which makes the calling service unable to handle other requests.
- The failure of one service can cascade to other services throughout the application.
- How to prevent a network or service failure from cascading to other services?

Circuit Breaker

- Parent service invokes the remote service via a proxy that functions similar to an electrical circuit breaker.
  - When the number of consecutive failures crosses a threshold, the circuit breaker trips.
  - 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.
- Parent service handles the failure of the services that they invoke.
- **Netflix Hystrix** is an implements this pattern.

Distributed Tracing

- Helps gather timing data needed to troubleshoot latency problems in service architectures.
- Data collection and analysis.
- Data can be structured.
- To quickly locate the data query based systems needed.
- Data can be summarized based on attributes.

Tracing with ZipKin

- Zipkin is a distributed tracing system.
- It helps gather timing data needed to troubleshoot latency problems in service architectures.
- Features include both the collection and lookup of this data.
- Data can be collected with trace ID in a log file, or jump directly to it.
- Run a query can query based on attributes such as service, operation name, tags and duration.
- Data can be summarized based on attributes.

Zipkin : Inputs for data

- The data can be reported to Zipkin via http or Kafka and many other options such as Apache ActiveMQ, gRPC and RabbitMQ.
- The data served to the UI is stored in-memory, or persistently with a supported backend such as Apache Cassandra or Elasticsearch.

ZipKin UI

- The Zipkin UI presents a Dependency diagram showing how many traced requests went through each application.
- This is helpful for identifying aggregate behavior including error paths or calls to deprecated services.

Spring Cloud Sleuth

- Spring Cloud Sleuth implements a distributed tracing solution for Spring Cloud,.
- Data can be captured in logs, or by sending it to a remote collector service.

Logging with ELK

- LogStash to collect the logs from different sources.
- The log collection can be load balanced.
- Passes the log data to Elasticsearch for storage.
- Elasticsearch supports analysis queries over RERST API.

All in One →Consul

- Consul is a service networking solution to connect and secure services across **any runtime platform and public or private clouds.**
- Supports service discovery, authorization, gateways as proxies and lot more as runtime platform.
- Consul supports service applications in platform independent ways.