Spring Boot Microservices

macalak@itexperts.sk

Course Agenda

- Introduction to Microservices
- Modeling Microservices
- Microservices Integration
- Introduction to Spring Boot
- Introduction to Spring Cloud
- Microservices Deployment
- Monitoring Microservices
- Microservices Security

Module 1
Introduction to Microservices

What is the Architecture?

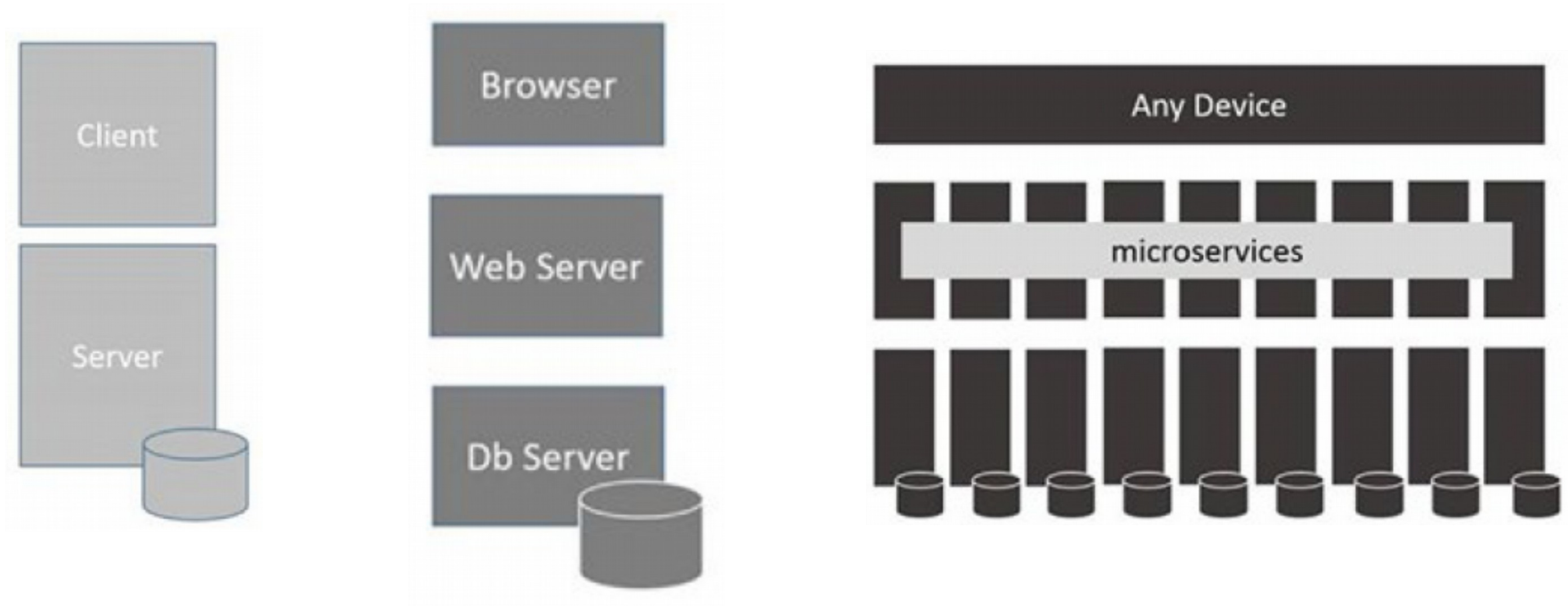
- The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

- The software architecture of a system supports the most critical requirements for the system.

Why Microservices

- Microservices style architectures are a response to adjust software architecture to an ever-evolving spectrum. It addresses **Business Agility** through technology
  - Usage of cloud-based infrastructure and services
  - DevOps
  - The need to scale up the number of people/teams
  - Client-side revolution both in technologies and devices

Architecture Evolution



Microservices Architecture

- Set of loosely coupled, collaborating services
- Services can be developed and deployed independently of one another
- Supports polyglot model with multiple programming languages, data transmission protocols, and persistence mechanisms
- Each service has its own database in order to be decoupled from other services
- Data consistency between services is maintained using an event-driven architecture

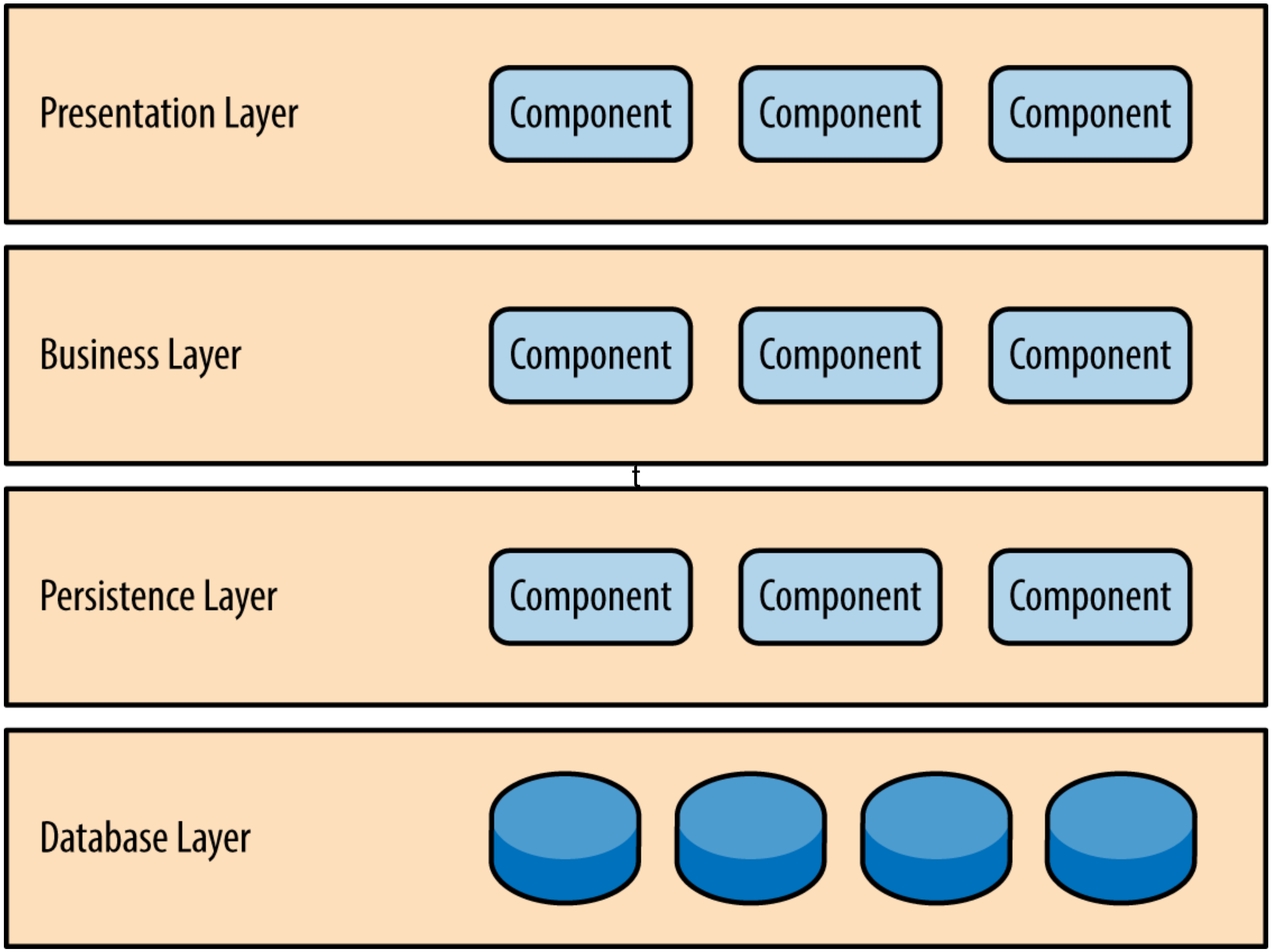
Architectural styles (1)

An architectural style, defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.

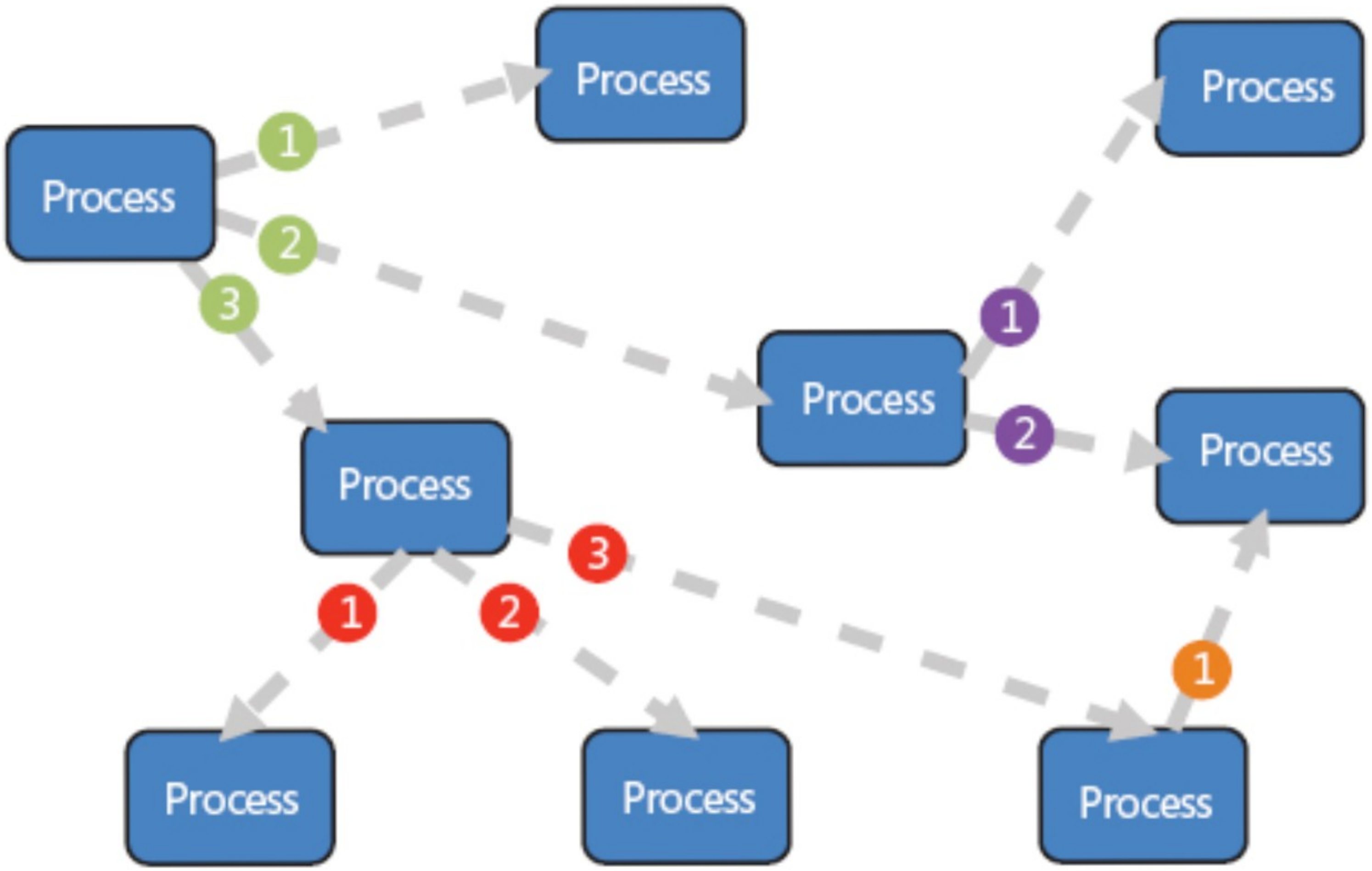
Architectural styles (2)

- Layered architecture
- Event driven architecture
- Hexagonal architecture (Ports & Adapters)

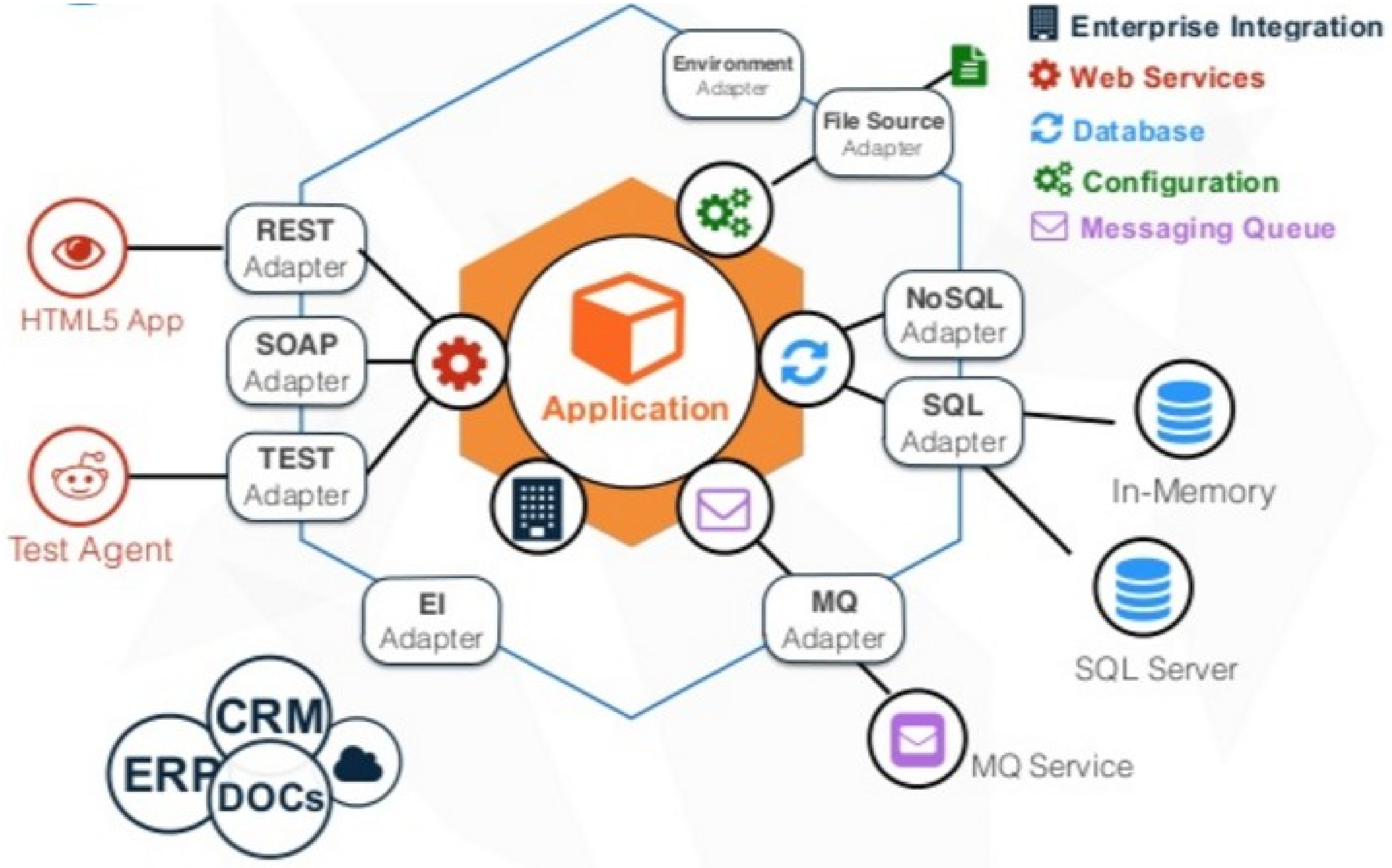
Layered architecture



Event driven architecture



Hexagonal architecture (Ports & Adapters)



Services that Works Together

- Relative small and focused on doing thing well
- Shaped by business boundaries
- Autonomous, deployed as isolated services
- Communicate via defined API

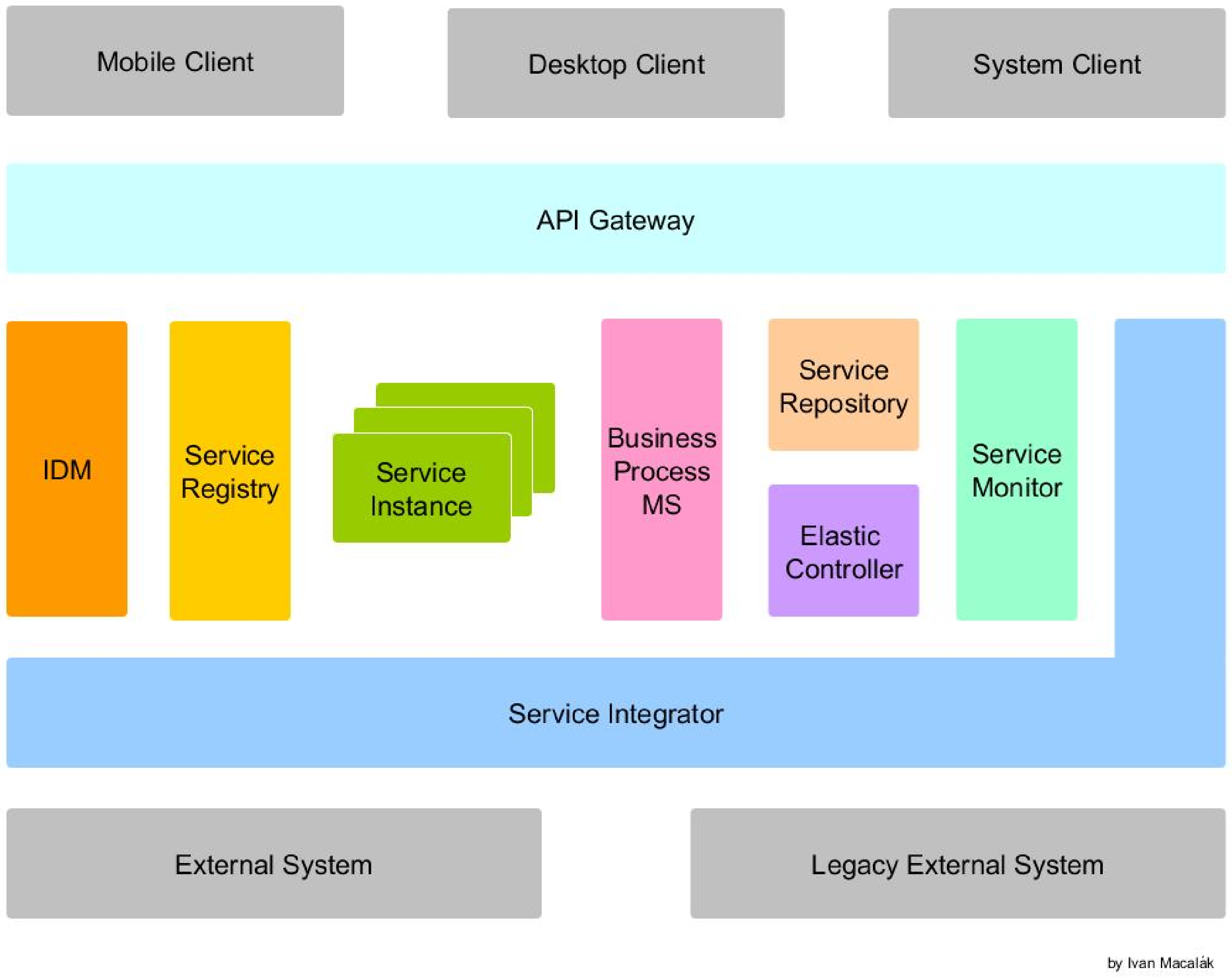
Benefits

- Small
  - Easier for a developer to understand
  - Starts faster
- Technology heterogenity
  - Eliminates any long-term commitment to a technology stack
- Resilience
  - Fault isolation
- Scaling
- Ease of deployment
  - Each service can be deployed independently of other services
  - Enables continuous delivery and deployment
- Replaceability

Drawbacks and Issues

- Complexity of creating a distributed system
- You must deal with failure / design for resiliency
- Operational complexity of deploying and managing a system comprised of many different services
- Increased resources consumption
- How to decompose application?
- How to maintain data consistency?
- You need initial investment

Microservices Reference Model



What about SOA?

Service Communication	
SOA	Microservices
Smart pipes Enterprise Service Bus	Dumb pipes Message broker
SOAP protocol	REST/gRPC

Service implementation	
SOA	Microservices
Larger monolithic application	Smaller independent services

Data	
SOA	Microservices
Global data model	Data model per service
Usually shared database	Database per service

Prerequisites (1)

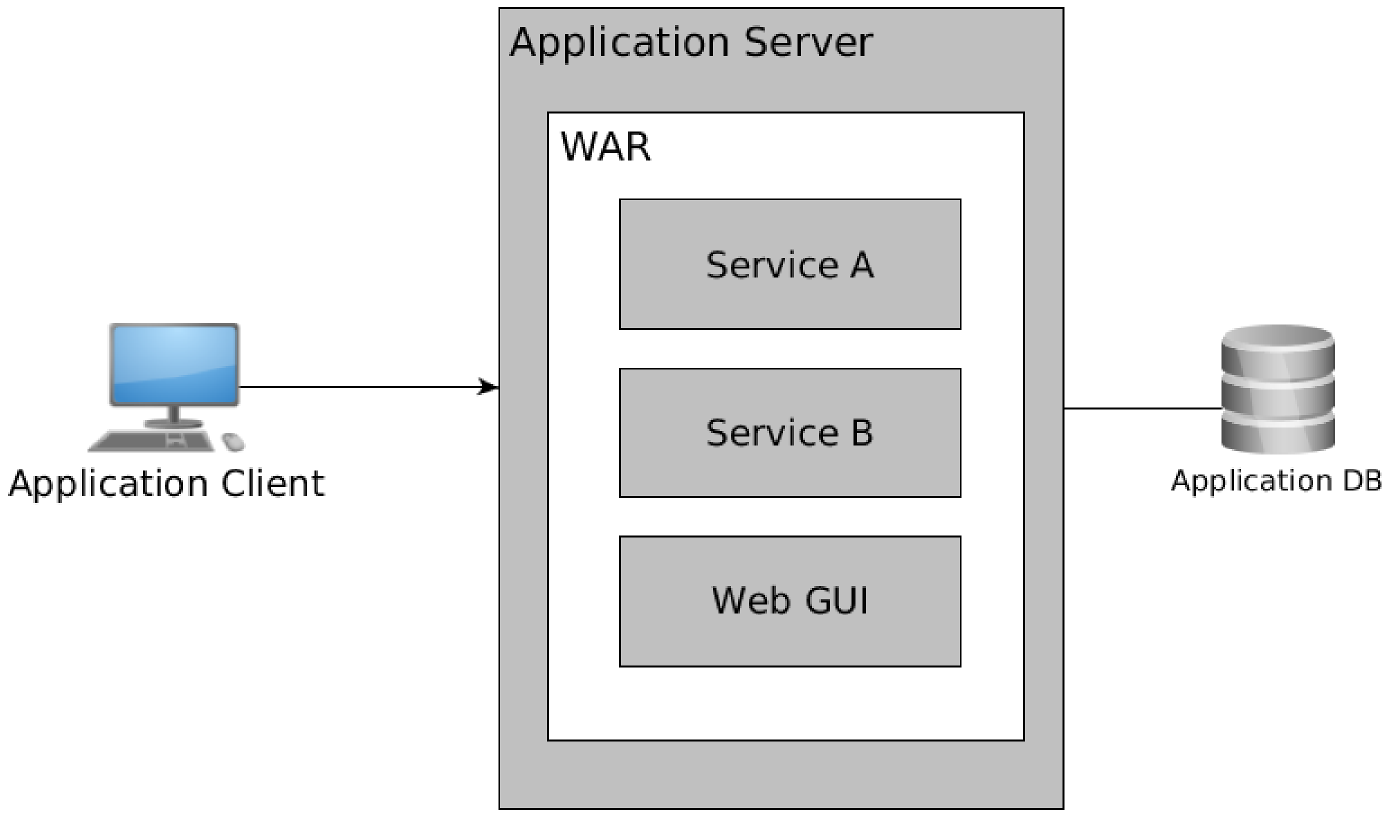
- Make sure the organization is compatible with the software architecture
- If your (microservices) architecture does not reflect the way your organization is structured, don’t even bother going that way!
- Your team should be cross-functional Everyone you need to build, maintain and get it into production must be part of the team

Conway’s Law: “Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations"

Prerequisites (2)

- DevOps process in-place
- Solid CI/CD practices
- Centralized Monitoring
- Centralized Logging/Tracing
- Metrics and Analytics

What about Monolitic Architecture? (1)



What about Monolitic Architecture? (2)

- Simple to develop and deploy, no complexity of distributed system
- Usually single WAR file deployed on application server
- Large code base become hard to maintain
  - Start to build up massive technical debt
  - Become hard to change without breaking stuff
- QA and test cycles take lots of time (expensive)
- Scaling out can be difficult and can vaste resources
- Hard to change the underlying technical stack
- Hard to scale development teams for big application

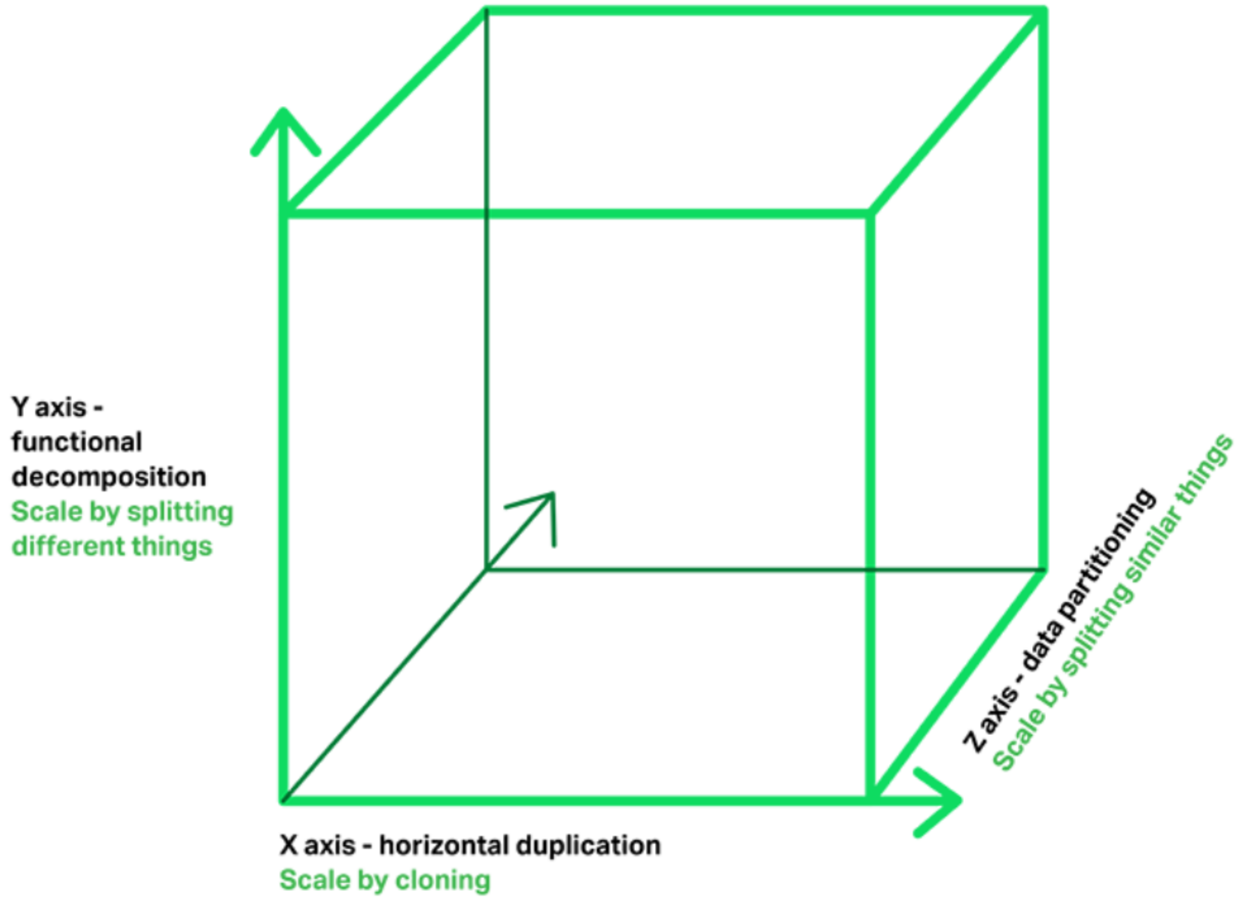
Module 2
Modeling Microservices

What is a Good Service?

- What is a service?
  - Loosely coupled
    - A change to one service should not require change to another
    - Such service knows as little as it needs to about the services with which it collaborates
  - High cohesive
    - Encapsulated related behavior and data
    - If there is need for change, the best is to do change in one place
- We need to find boundaries of problem domain
- What is the right size of a service?

Decomposition

3D model of scalability


Y axis - functional decomposition
Scale by splitting different things
X axis - horizontal duplication
Scale by cloning
Z axis - data partitioning
Scale by splitting similar things

x axis: scaling is for running multiple cloned copies of an application behind a load balancer.

y axis: scaling represents an application that is split by a function, service, or resource. Each service is responsible for one or more closely related functions.

z axis: scaling is commonly used to scale databases because the data is partitioned across a set of servers.

Decomposition by Use case pattern

- It is about starting, identifying and prioritizing UCs
- Partitioning of application into many small modules
- Each of them will accomplish at least one use case

Decomposition by Resources pattern

- Defining microservices based on resources as servers, storages, network components, databases they access or control
- Service as channel for access to individual resources

Decomposition by Business capability pattern

- A business capability is something that a business does in order to generate value
  - Customer management
  - Order management
- Define services corresponding to business capabilities

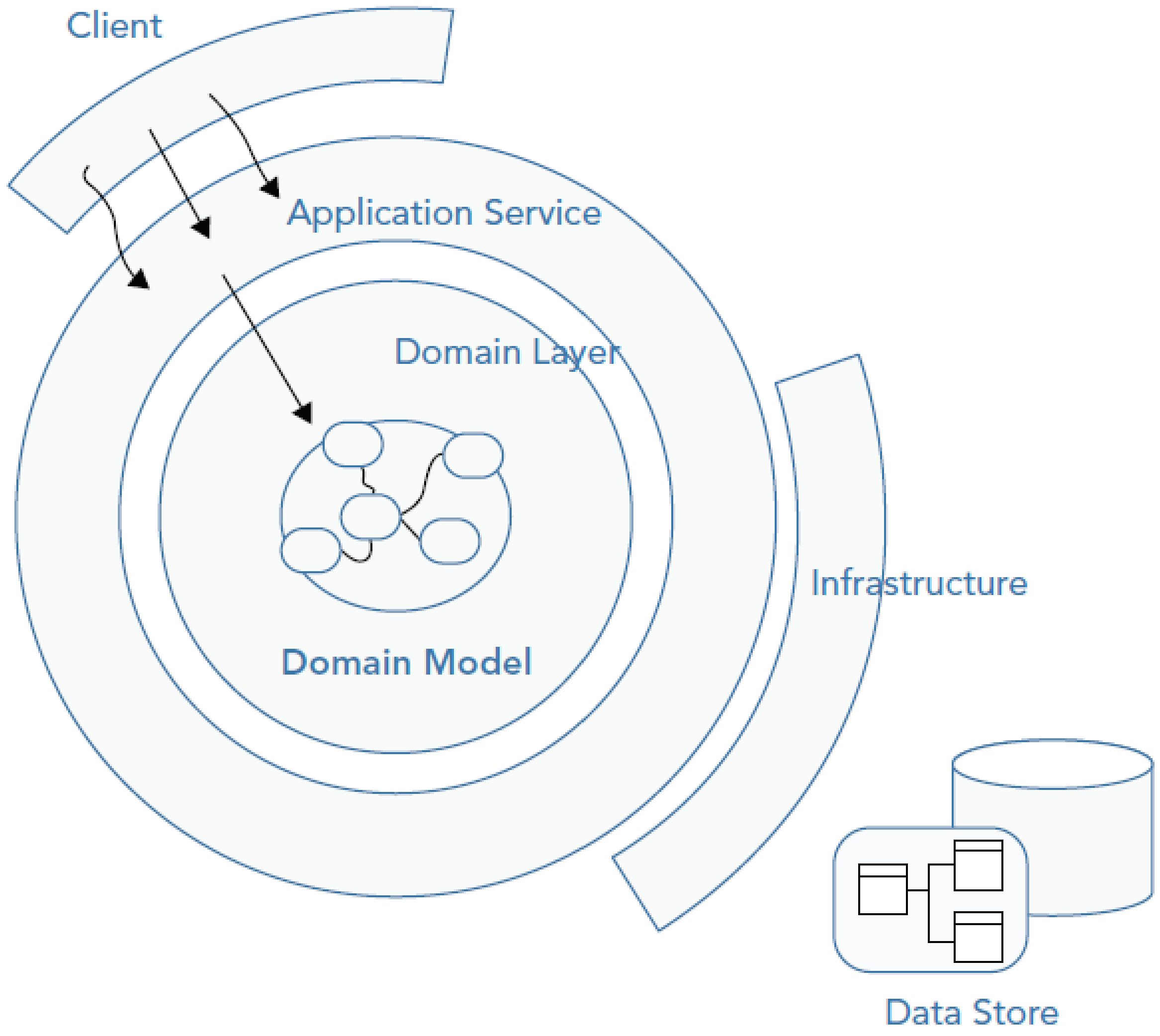
Decomposition by Subdomain pattern

- Define services corresponding to Domain Driven Design (DDD) subdomains
- DDD refers to the application's problem space as the domain
- Domain consists of multiple subdomains
- Each subdomain corresponds to a different part of the business. Online store example:
  - Product catalogue
  - Inventory management
  - Order management
  - Delivery management

Domain Driven Design

- Methodology for evolving a software system that closely aligns to business requirements
- Targeted to complex business domains
- You should have access to Domain Experts
- You should have an iterative process
- You should have a skilled and motivated team
- Not a silver bullet
- Not always the best solution

Domain Model Pattern



The Domain Vision Statement

- A shared understanding of what it is you are actually trying to create
- Should be brief, written in clear language and understood by business and tech people alike
- Should be factual, realistic, honest
- Should avoid superlatives and marketing speak
- Should avoid technical and implementation details

Domain

- A Domain is a Sphere of Knowledge, Influence or Activity
- A Domain is represented by the Ubiquitous Language
- A Domain encapsulates a Domain Model
- A Domain lives within a Bounded Context

The Ubiquitous Language

- The Ubiquitous Language is a shared language between the business and the development teams
- The UL comes from the business, and is enriched by the development teams
- You should create a Domain dictionary - defines a number of terms which emerge, directly or indirectly, from the problem space of a domain

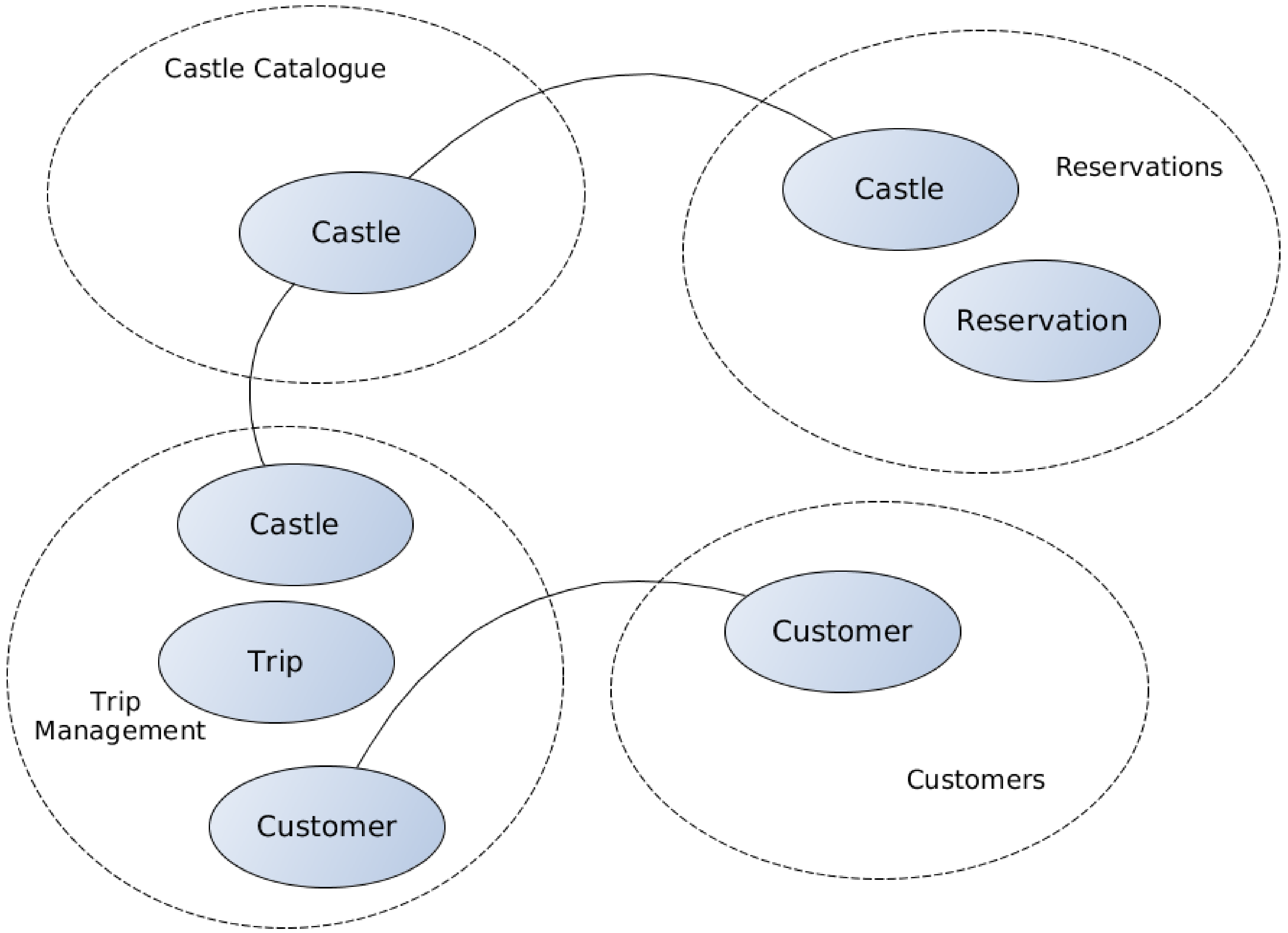
Domain Experts

- Domain Experts are the primary point of contact the development teams have with the business
- They are the Experts on their part of the business, not just users of the system
- They should have deep knowledge of the subject Domain

Bounded Context

- When you have multiple models you should consider Bounded Contexts
- Bounded Context provides capabilities to the rest of domain
- Each Bounded Context is a self contained “mini application” containing it’s own model, persistence and code base
- To map between Bounded Contexts you use a Context Map

Example of Context Map



Entities and Value Objects

- Entities
  - Entities are the “things” within your Model
  - An Entity is defined by being unique, and uniquely identifiable
    - `Castle` Entity
    - `Customer` Entity
- Value Objects
  - Value Objects are the “things” within your model that have no uniqueness
  - They are equal in all ways to another Value Object if all their properties match
  - Value Objects are interchangeable
    - `Guest` Entity

Factory

- An object or method that implements object creation logic that’s too complex to be done directly by a constructor. It can also hide the concrete classes that are instantiated.

Repository

- An object that provides access to persistent entities and encapsulates the mechanism for accessing the database.

Service

- An object that implements business logic that doesn’t belong in an entity or a value object.

Domain Model

- A Domain Model is a representation of the relationships between the Entities and Value Objects in your Domain
- It may look similar to UML or a class relationship diagram, but it is not one
- The Domain Model should be recognizable and understandable by the business

Aggregates

- An aggregate is a collection of items that are gathered together to form a total quantity
- An Aggregate Root is the root item containing a number of parts that form a whole
- An AR is more likely to match a Use Case than any model structure

Principles

- Single Responsibility Principle
- Common Closure Principle

System operations

- Commands – operations that usually modify data as create, update delete
  - `createReservation`
  - `modifyCastleDetail`
- Queries – operations that read data
  - `findCastle`
- They corresponds to
  - REST endpoints
  - RPC endpoints
  - Messaging endpoints

Domain events

- Events – communicate something that happened in the domain
  - `reservationCreated`
  - `castleDetailModified`

Challenges

- Network – latency, unstable
- Data – consistency across services

Module 3
Microservices Integration

Service Boundary

- Loosely coupled services should communicate over their boundaries with defined contract
- Internal implementation detail should stay hidden

Shared Database

- Service can view data of another service
  - Internal service implementation is not hidden
- Related service Is tight to DB technology
  - What if you decide to move from SQL to NoSQL
- Direct data manipulation bypasses business logic related with modification
- Problems with DB locking

RPC

- Used in monolithic architecture to get the scalability benefit of a distributed system
- Hides the complexity of remote call
  - Remote call looks like local call
  - But they do not, because there is a (unreliable) network
- RPC is mostly synchronous, so does not scale well
- Some RPC like Java RMI are tied to a specific platform, which can be a limitation
- It is harder to make it resilient and involves tight coupling

REST API

- REST endpoints
- Requests to access Resource
- Use HTTP operations/verbs
- Use HTTP status codes

Messaging

- Asynchronous nature which brings complexity
- Supports decoupling
- Supports event driven systems
- Scales well
- Message bus supports resiliency
- It is harder to debug
- You need another infrastructure element

Integrate trough Service API

- Focus on behavior/business logic not data
- API can invoke another related service or business logic
- API call is validated (business validation)

Interaction styles

- Synchronous - the client expects a timely response from the service and might even block while it waits.
- Asynchronous - the client doesn’t block, and the response, if any, isn’t necessarily sent immediately.
- One-to-one – each request is processed by exactly one service
  - request/response
  - async request/response
  - One-way notificatoin
- One-to-many - each request is processed by multiple services
  - publish/subscibe
  - publish/async responses

Synchronous vs Asynchronous

- Synchronous
  - Blocking calls
  - Request/Response
  - Easy to implement
- Asynchronous
  - Non blocking calls
  - Responsive
  - Long running jobs
  - Event based
  - Callbacks
  - Highly decoupled

Standardize on a common communication protocol

You can use both styles

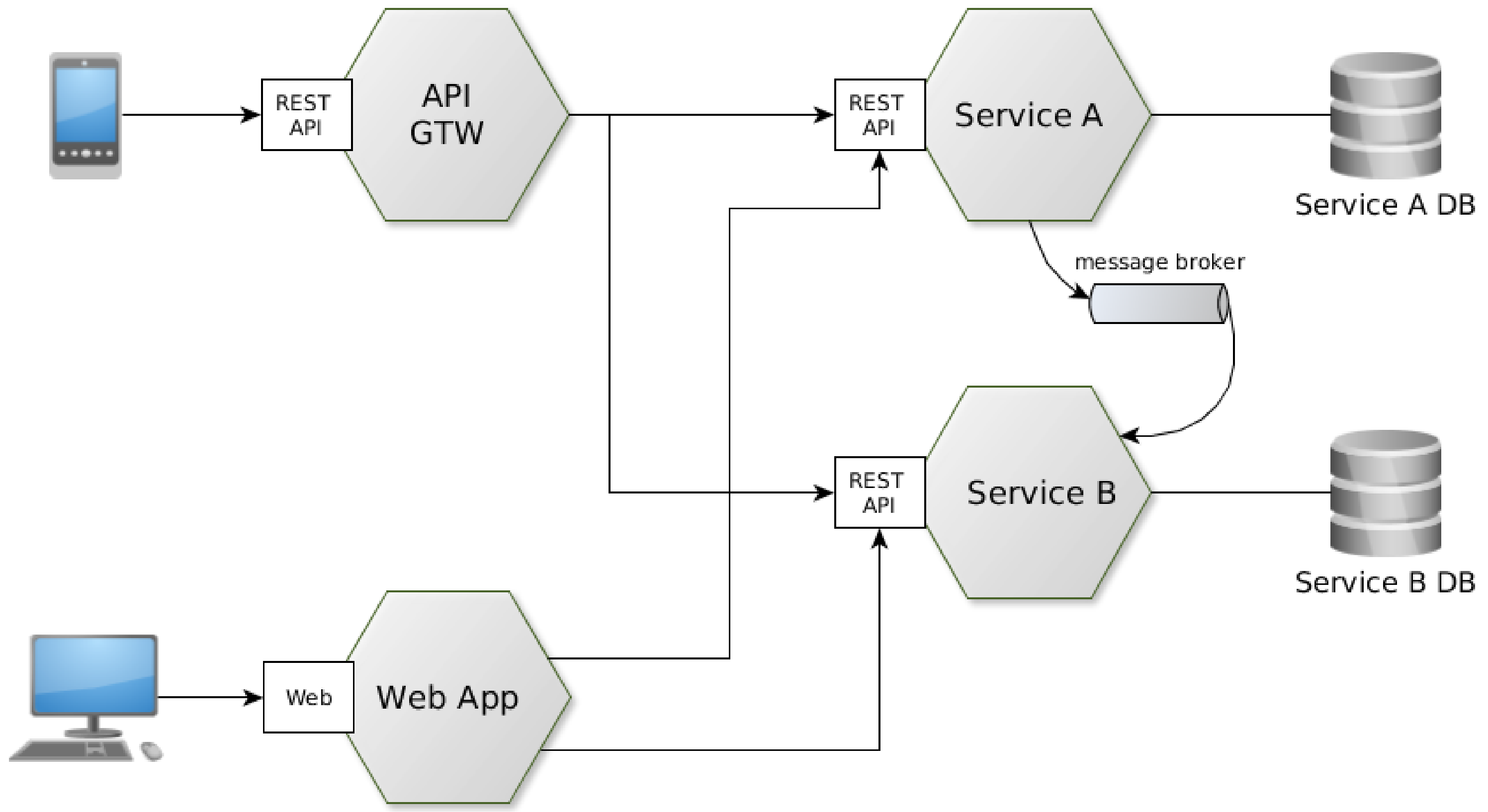
Message formats

- Text based formats
  - XML
  - JSON
- Binary formats
  - Apache Avro
  - Google Protocol Buffers

Orchestration vs Choreography

- How to handle business processes that span across boundary of individual services?
- Orchestration
  - Requires a central component
  - Explicit view of business process
- Choreography
  - Based on events consumed by services interested
  - Each service must understand its role
  - More loosely coupled and flexible
  - Implicit hidden business process

Microservices Communication Model



Communication patterns

- Remote procedure invocation
- Circuit breaker
- Client-side discovery
- Self registration
- Server-side discovery
- Asynchronous messaging
- Polling publisher

Service registry

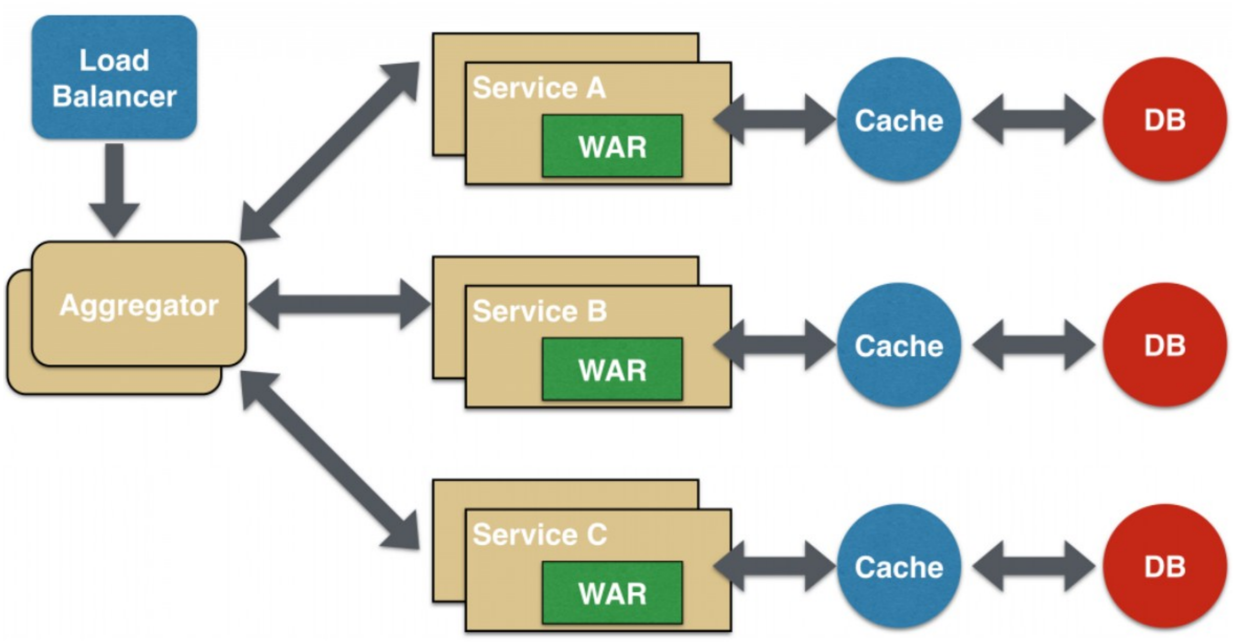
- Database of service instances and their locations
- Client of the service and/or routers can discover the location of service instances
  - Client-side discover
  - Server-side discovery
- Implementation
  - Netflix Eureka
  - HashiCorp Consul

Proxy microservice

- Microservice need not be exposed to the consumer directly
- Proxy can apply security and data transformation

Aggregator microservice

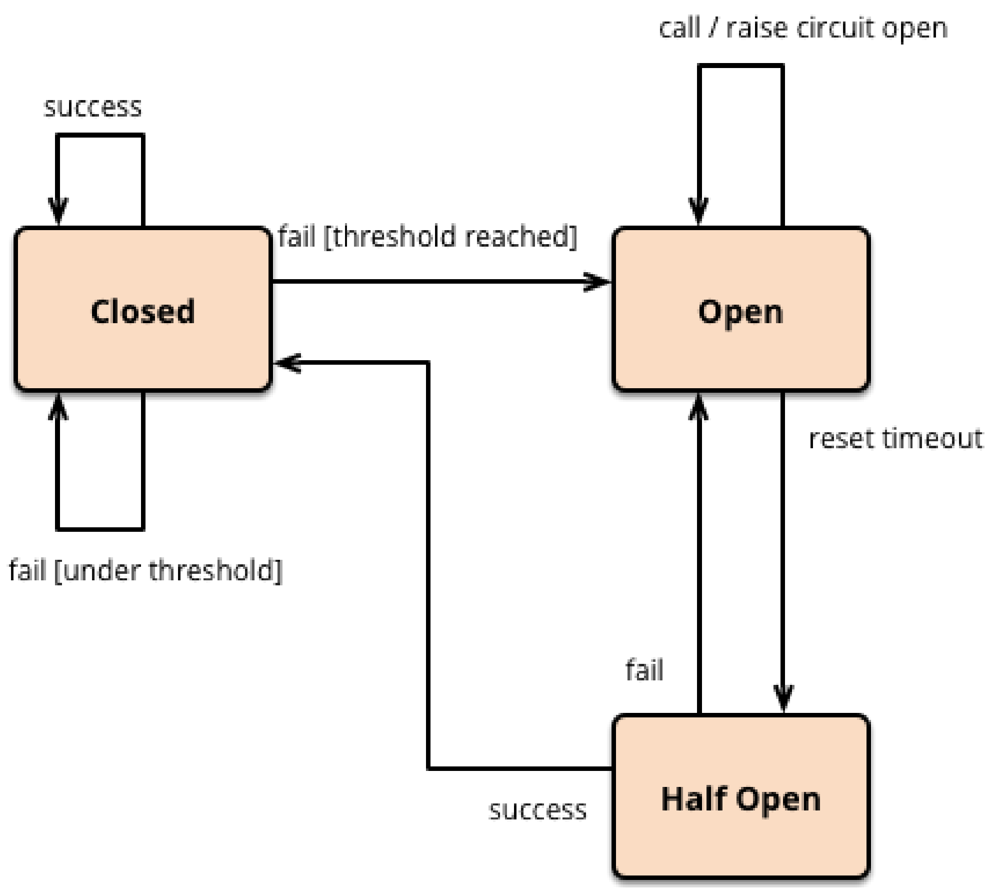
- To support business case/service, services must be aggregates
- Aggregator collects the data from each of the participating services



API Gateway

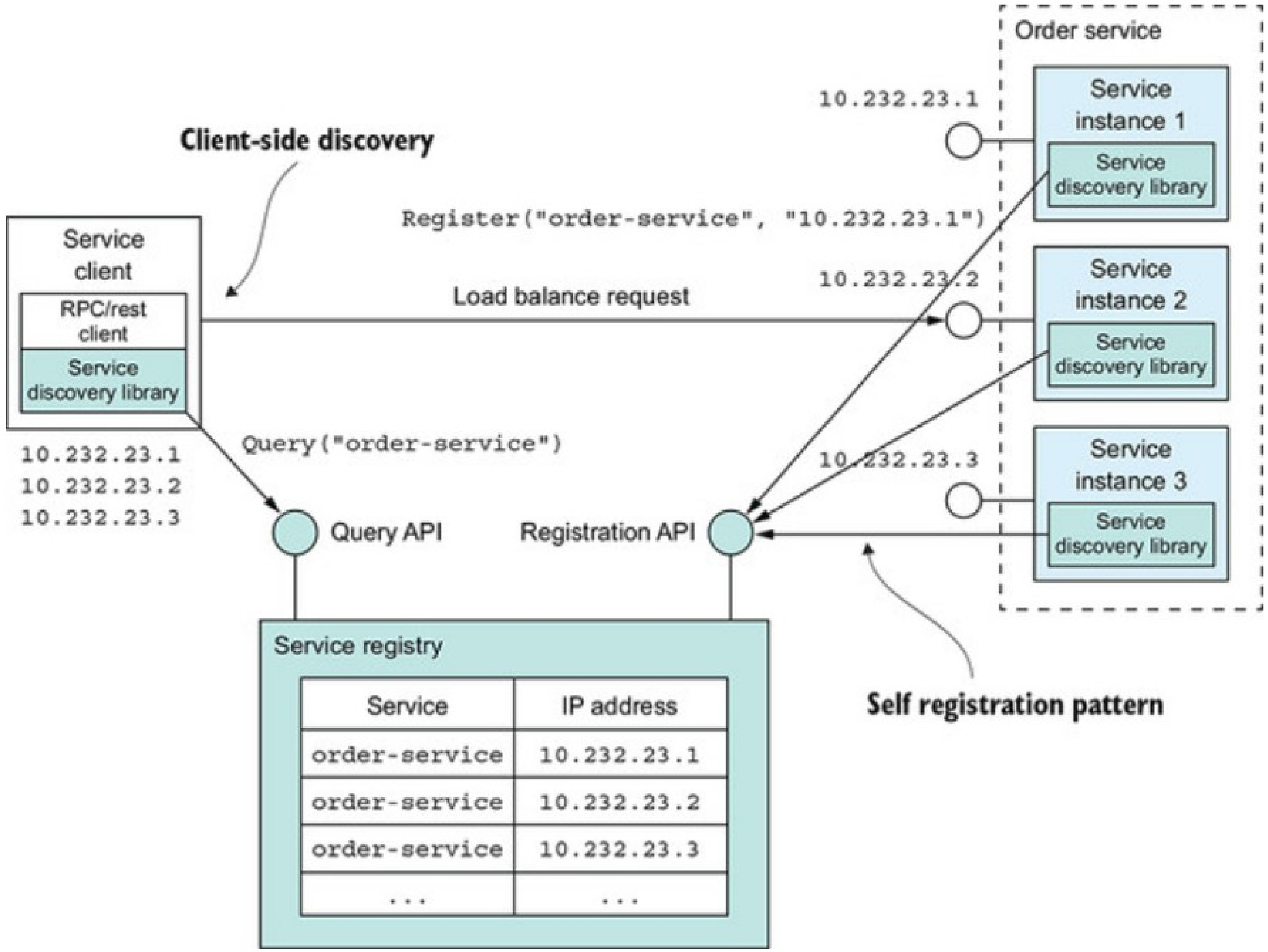
- Exposes API for clients of individual services
- Single entry point for all clients
- Insulates the clients from how the application is partitioned into microservices
- Insulates the clients from the problem of determining the locations of service instances
- Provides the optimal API for each client
- Reduces the number of requests/roundtrips
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally

Circuit breaker



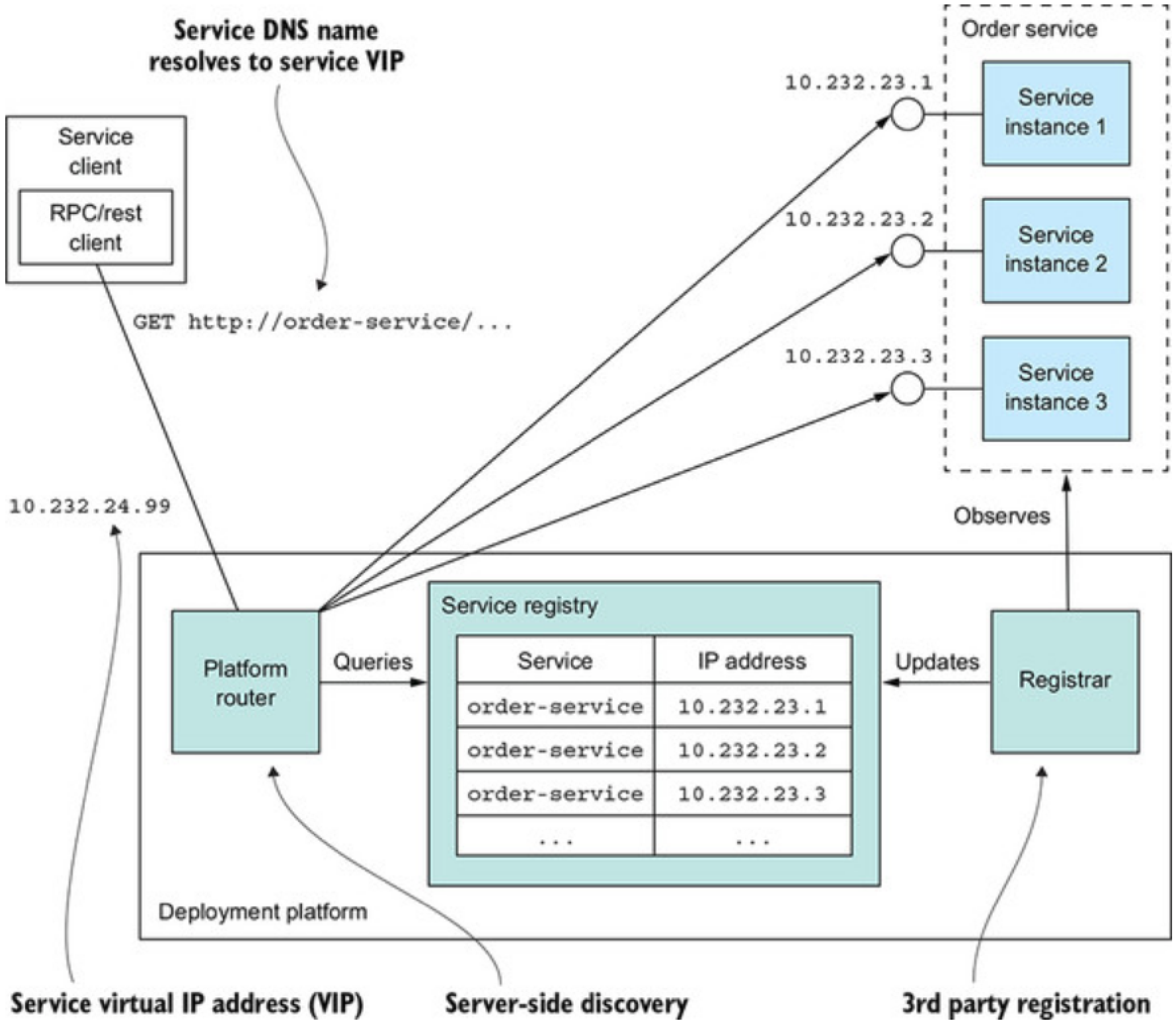
Track the number of successful and failed requests, and if the error rate exceeds some threshold, trip the circuit breaker so that further attempts fail immediately. A large number of requests failing suggests that the service is unavailable and that sending more requests is pointless. After a timeout period, the client should try again, and, if successful, close the circuit breaker.

Service discovery (1)



Service	IP address
order-service	10.232.23.1
order-service	10.232.23.2
order-service	10.232.23.3
...	...

Service discovery (2)



Service	IP address
order-service	10.232.23.1
order-service	10.232.23.2
order-service	10.232.23.3
...	...

Module 4
Introduction to Spring Boot

Spring Boot

- Java based framework for building microservices
- Pre-configured dependencies
- Auto configuration
- Starters
- Single executable jar
- Embedded servers
- http://start.spring.io for quick start

Spring Boot Starters

- Set of convenient dependency descriptors that you can include in your application
- Provides consistent dependencies that you need
- You just include dependency on required starter in project build tool (Gradle or Maven)
  - `spring-boot-starter-web` for web and RESTful applications
  - `spring-boot-starter-data-jpa` for using Spring Data JPA with Hibernate
  - `spring-boot-starter-test` for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito
  - …
- For quick application bootstrap use http://start.spring.io/

Spring Boot Auto-configuration

- Attempts to automatically configure your Spring application based on the jar dependencies that you have added
- You should add `@SpringBootApplication` annotation to one of your Spring Configuration classes

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Running Spring Boot Application

- Application is by default packed to single JAR
- Web application uses embedded HTTP server
- You can run application
  - From your favorite IDE
  - From CL simple as `java -jar myapplication.jar`
  - Using Gradle or Maven plugins `gradle bootRun`
- Supports hot swapping and other development tools as automatic restart

Spring Boot Application Configuration

- You an use `application.properties` or `application.yml` files to configure different application properties
- Default property file are included on classpath
- You can override these variables by environment properties
- Properties can be profile specific `application-{profile}.properties`

Spring Boot Application Logging

- By default Commons Logging for internal logging
- You can choose your own logging implementation as JUL, Log4j and Logback (default for starters)
- Messages are logged to console by default (`ERROR`, `WARN` and `INFO` levels)
- You can enable debug mode by starting application with `–debug` flag or include `debug=true` in `application.properties`
- Custom log configuration goes to `logback.xml`

Embedded Servlet Containers

- Spring Boot supports
  - Tomcat (default for web starter)
  - Jetty
  - Undertow
- You can customize containers in `application.properties`
  - `server.port`
  - `server.address, ...`

Embedded Databases Support

- Convenient during development
- Spring Boot auto-configuration detects
  - H2
  - HSQL
  - Derby
- You can enable H2 web console
  - `spring.h2.console.enabled=true`
  - optionally set
    - `spring.h2.console.path`
    - `security.user.role`
    - `security.basic.authorize-mode`
    - `security.basic.enabled`

Messaging Support

- Spring supports
  - JMS/ActiveMQ/Apache Artemis
  - AMQP/RabbitMQ
  - Apache Kafka

Spring Boot How To

- https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#howto

Module 5
Introduction to Spring Cloud

Cloud Native Applications

- Can be deployed into the cloud
- Self contained
- Robust
- Self healing
- Elastic

Spring Cloud

- Provides developers with common patterns in distributed systems
  - Configuration management
  - Service Registration and Discovery
  - Intelligent routing
  - Load balancing
  - Distributed messaging
- It builds on Spring Boot

Configuration Server

- Provides centralized and externalized configuration for services
- Central place to manage application properties
- Backed by a git repository or Vault
- Supports configuration versions for different environments
- REST API
- Property encription
- Push notifications

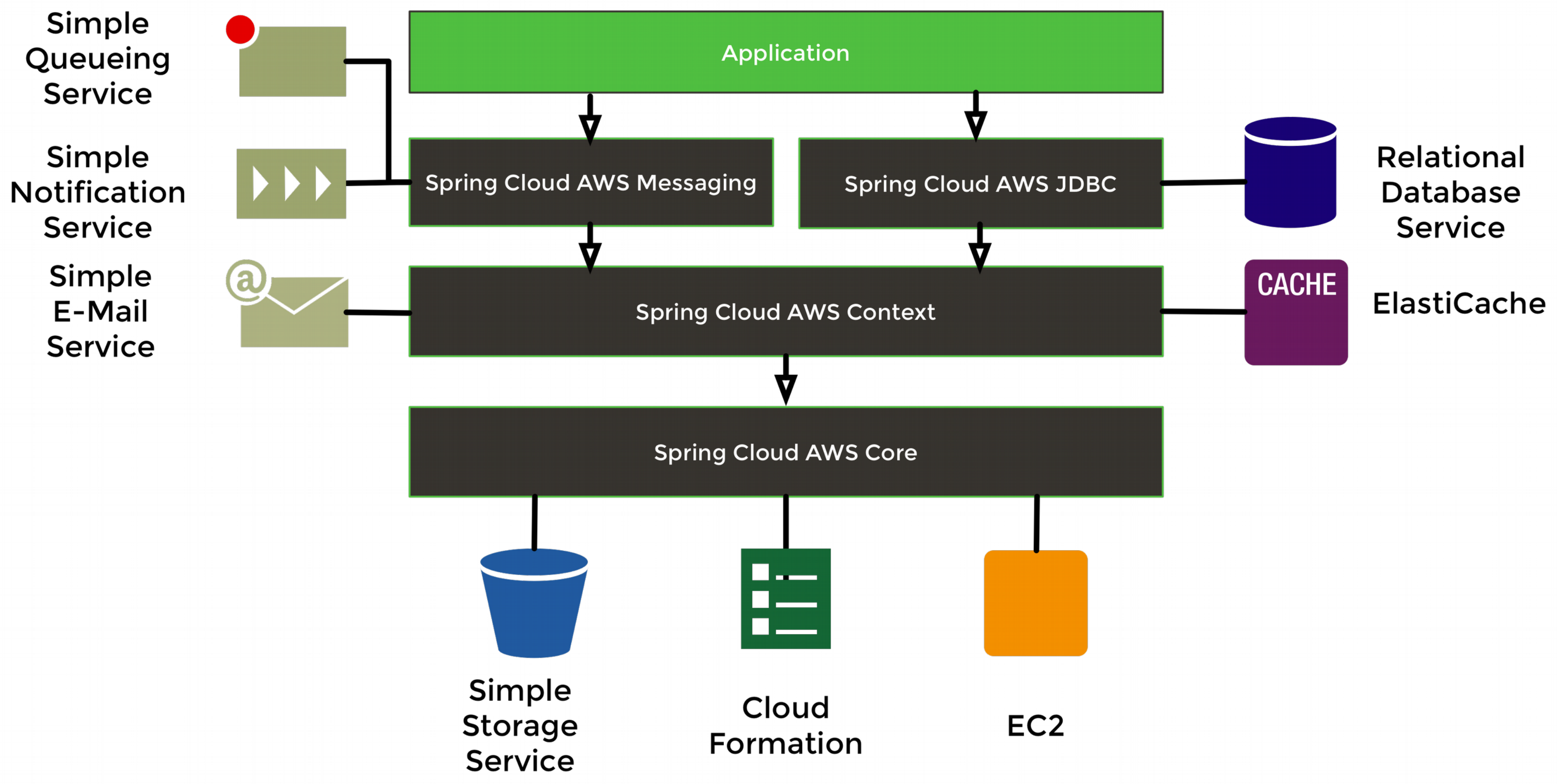
Service Registry and Discovery

- Services and their instances are hidden in the cloud
- Service Registry provides registration service for your microservices
- Spring Cloud Netflix integrates Netflix OSS with Spring Boot applications
- Netflix Eureka is Service Discovery Server and Client
- Eureka can be deployed on AWS
- Client integrates with Spring RestTemplate and Feign (declarative web service client)
- Resiliency support by running multiple Eureka instances

AWS Integration (1)

- Spring Cloud integration with hosted Amazon Web Services
  - Spring Messaging API implementation for SQS (Simple Queue Service)
  - Spring Cache API implementation for ElastiCache
  - Mapping of SNS (Simple Notification Service) endpoints
  - Integration with RDS (Relational Database Service)
  - Integration with S3 (Simple Storage Service)

AWS Integration (2)

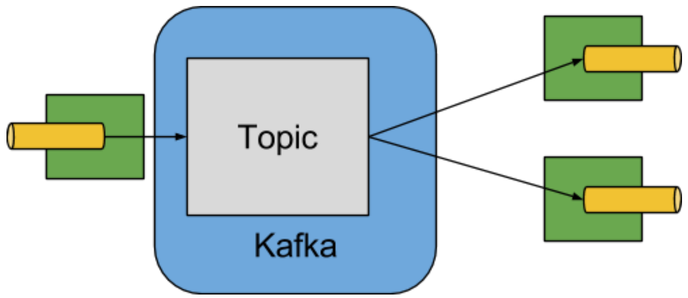


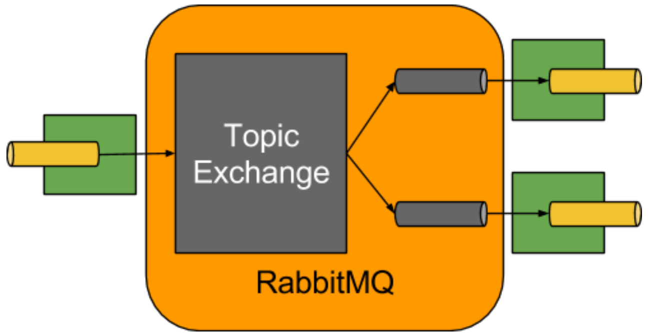
Client Side Load Balancing

- You scale service to achieve high availability, resiliency and required performance
- Load balancer distributes load to all service instances
- Spring Cloud integrates Netflix Ribbon client side Load balancer
- It is used also by Feign client
- It integrates with Eureka, so it balances requests to all discovered service instances

Event-Driven Message Bus

- Spring Cloud Stream for building message driven microservices
- Supports persistent Publish/Subscribe model
- Supports consumer groups and partitioning
- Abstracts message broker infrastructure (RabbitMQ, Appache Kafka, Redis and Gemfire)





Resilience

- Spring Cloud integrates the Netflix Hystrix Circuit Breaker
- Fall fast principle
- Avoids cascading failures
- Provides Hystrix Dashboard which presents Hystrix metrics and the health of each circuit breaker
- Provides Turbine Hystrix stream aggregator

Module 6
Microservices Deployment

Multiple service instances per host

- Multiple services on physical or virtual server
  - Multiple JVMs
- Possible issues
  - Competition for resources
  - Resource dependencies
  - Resource monitoring

Single service instance per host

- Service instances are fully isolated from one another
- There is no competition for resources and the issues being associated with dependencies are no more
- A service instance can consume at most the resources of a single host
- It is easy to monitor, manage, and redeploy each service instance
- It can decrease the resource utilization

Service instance per VM

- Single VM per service instance
- Cloud hosted VM services
- Service isolation at VM level
- Auto-scaling capabilities

Service instance per container

- Service isolation at (Docker) container level
- Container images hosted on dedicated hub/register
- Easily manageable and deployable
- Better resources (VM) utilization
- Cloud provided containerized platforms as Amazon ECS, Azure AKS, Red Hat Open Shift
  - Containers orchestration
  - Auto-scaling
  - Monitoring

Serverless deployment

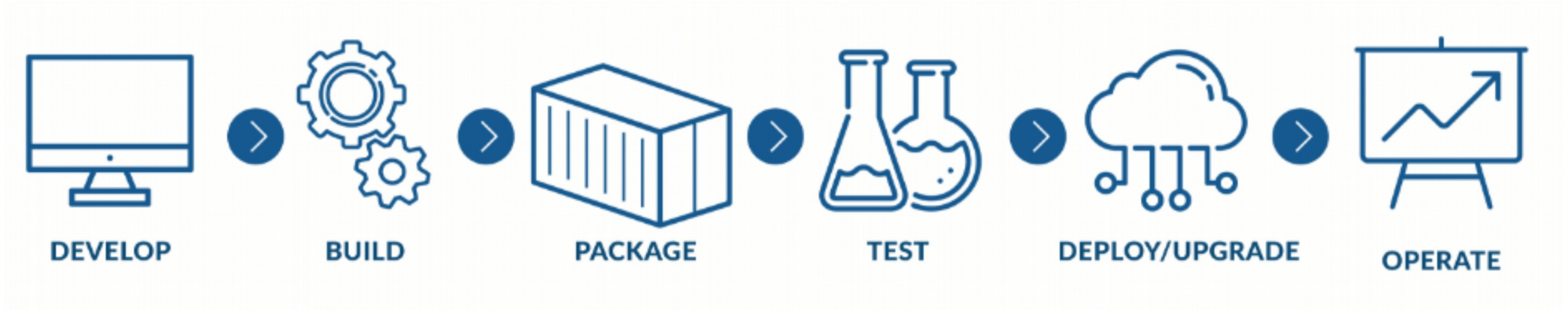
- Function as a Service (FaaS)
- Hides the concept of servers or virtual machines
- Infrastructure takes application service’s code and runs it
- No server management required
- Cloud services available
  - AWS Lambda Functions
  - Azure Functions
- Open source Apache OpenWhisk

Service deployment platform

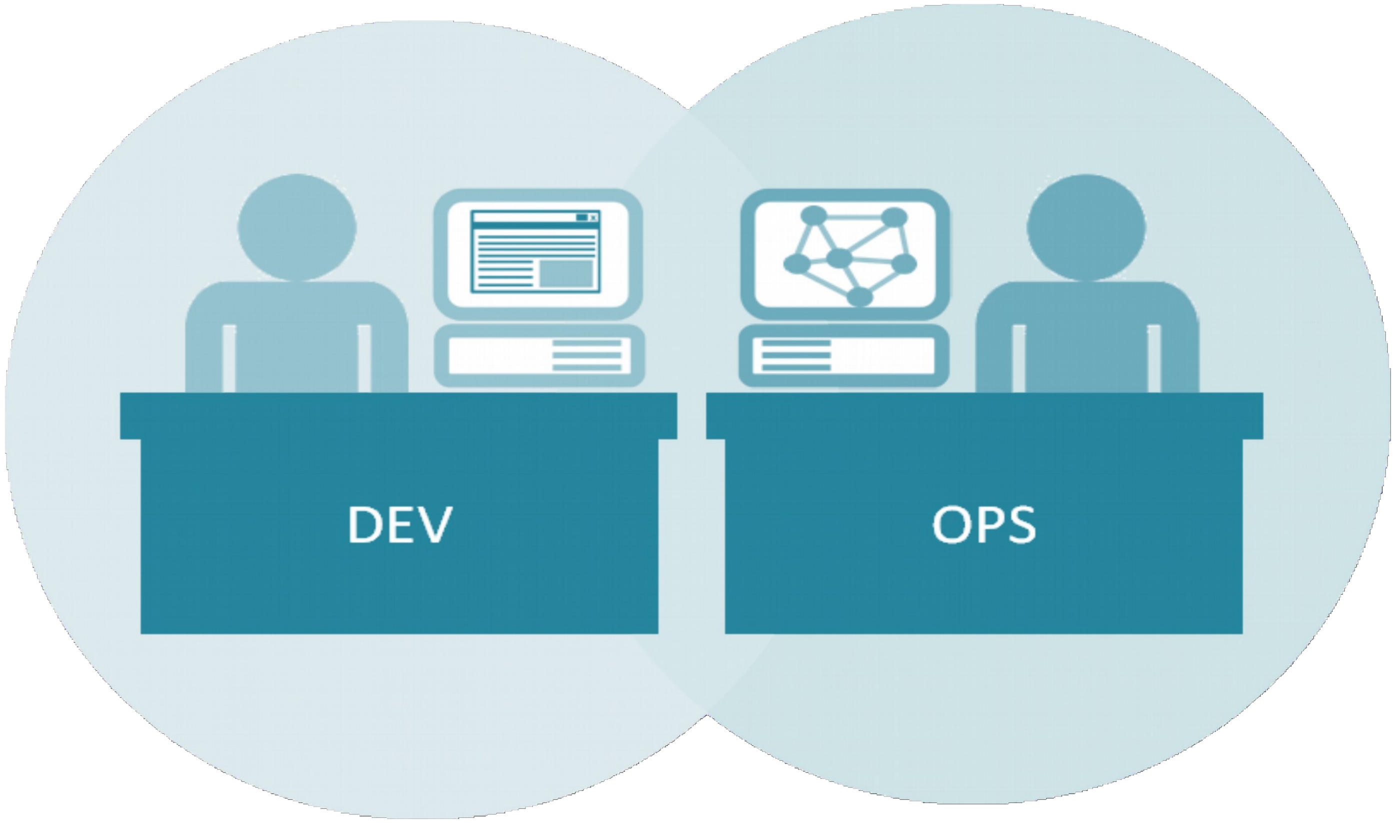
- Docker orchestration frameworks
  - Kubernetes
  - Rancher platform
  - Amazon EKS
  - Azure AKS
- PaaS
  - Red Hat OpenShift
  - Cloud Foundry
  - HashiCorp
- Supports CI/CD pipelines

Continuous Delivery

- From source code to production
- Deployment pipeline purpose
  - **Visibility:** All aspects of the delivery system - building,deploying, testing, and releasing – are visible to all team members promoting collaboration.
  - **Feedback:** Team members learn of problems as soon as they occur so that issues are fixed as soon as possible.
  - **Continually Deploy:** Through a fully automated process, you can deploy and release any version of the software to any environment.



DevOps



- Team Culture
- Communication
- Collaboration
- Automation
- Effectiveness

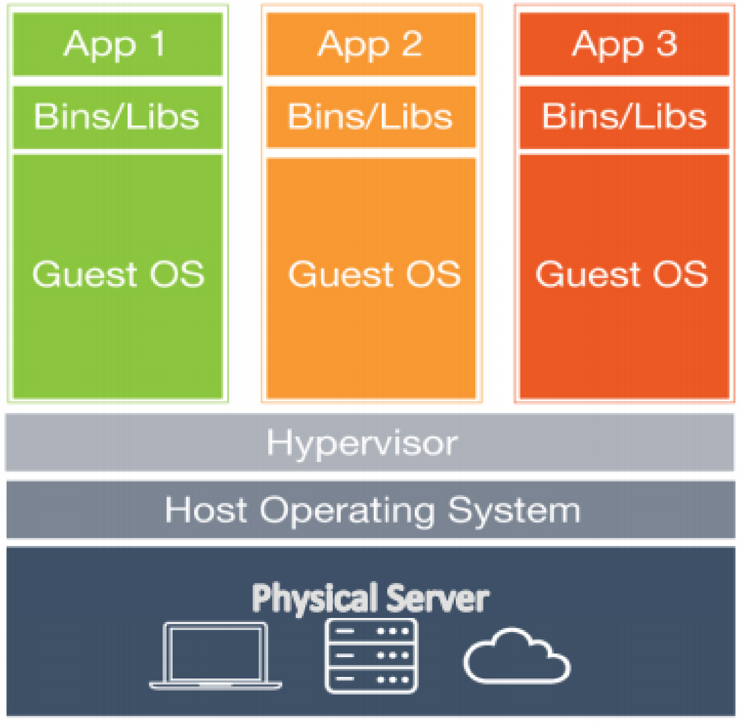
Spring Boot Application as Linux Service

- Spring Boot can be started as Linux service
- https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#deployment-service

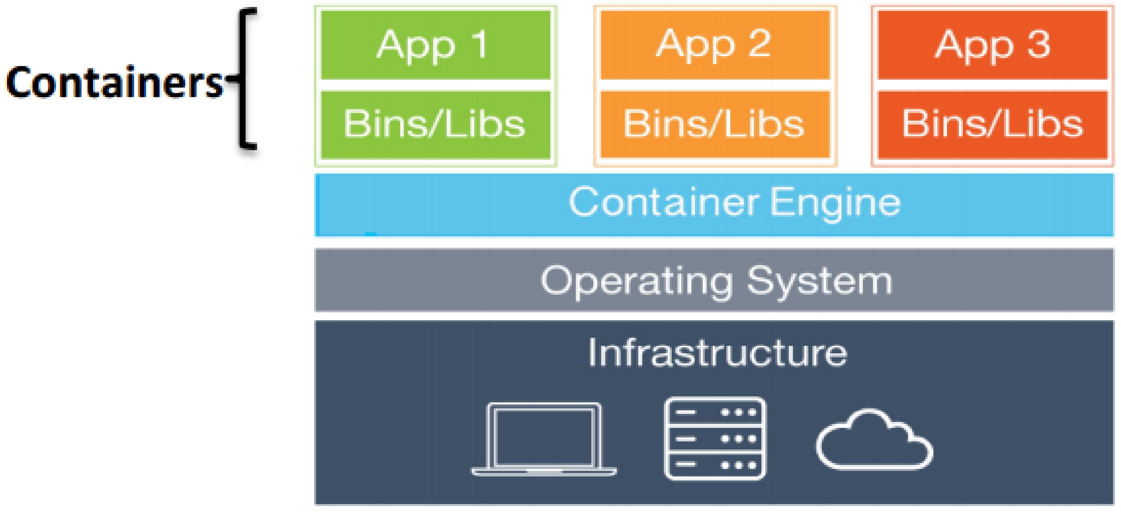
Containers

- Content Agnostic - Can encapsulate any payload and its dependencies.
- Hardware Agnostic - Using operating system primitives (e.g. LXC) can run consistently on virtually any hardware (VMs, bare metal, openstack, public IAAS, etc.) without modification.
- Content Isolation and Interaction - Resource, network, and content isolation. Avoids dependency hell.
- Automation - Standard operations to run, start, stop, commit, search, etc. Perfect for devops: CI, CD, autoscaling, hybrid clouds
- Highly efficient - Lightweight, virtually no perf or start-up penalty, quick to move and manipulate.
- Separation of duties - Developer worries about code. Ops worries about infrastructure.

Containerized Services

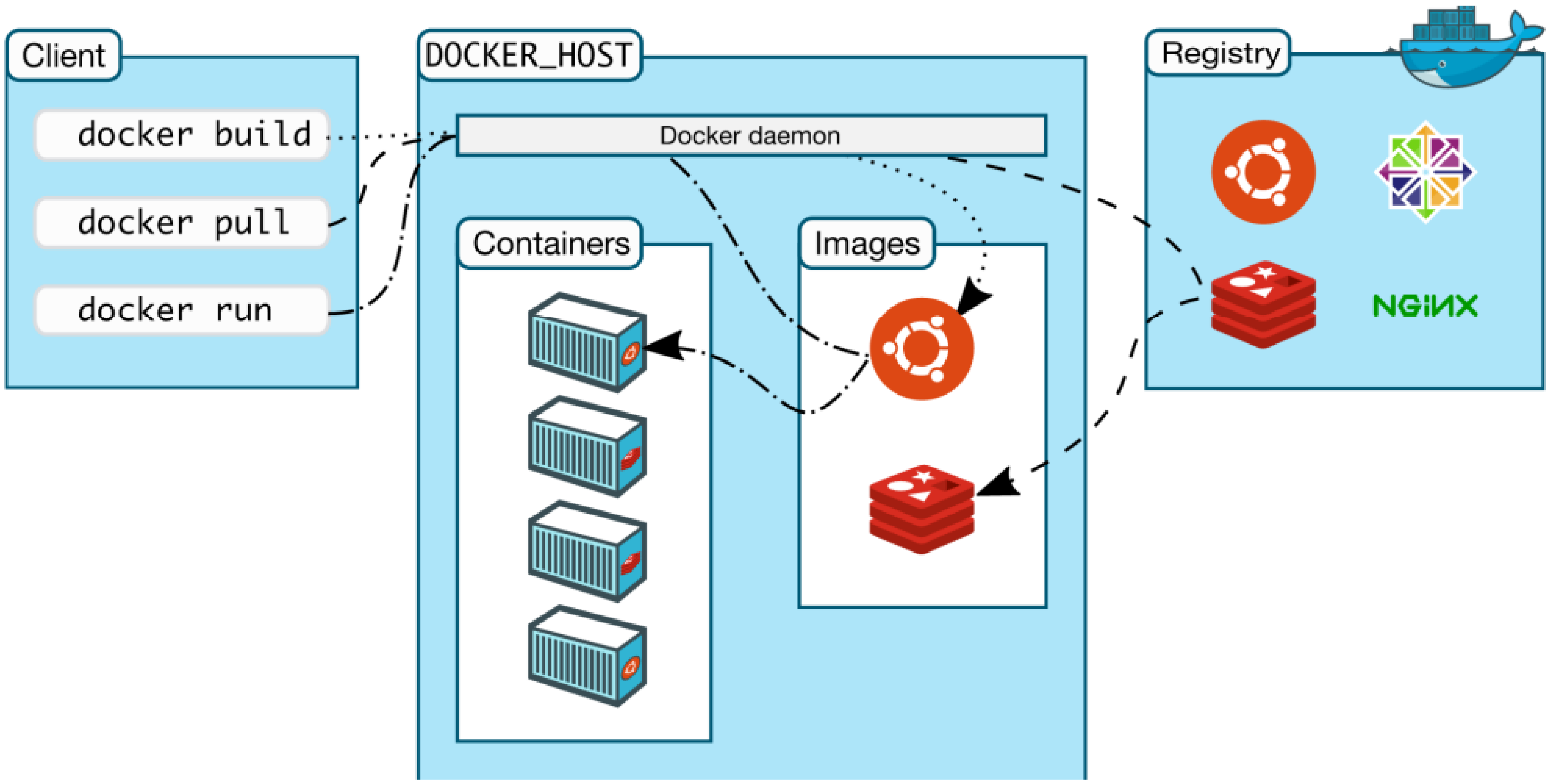


Hypervisor-based Virtualization



Container-based Virtualization

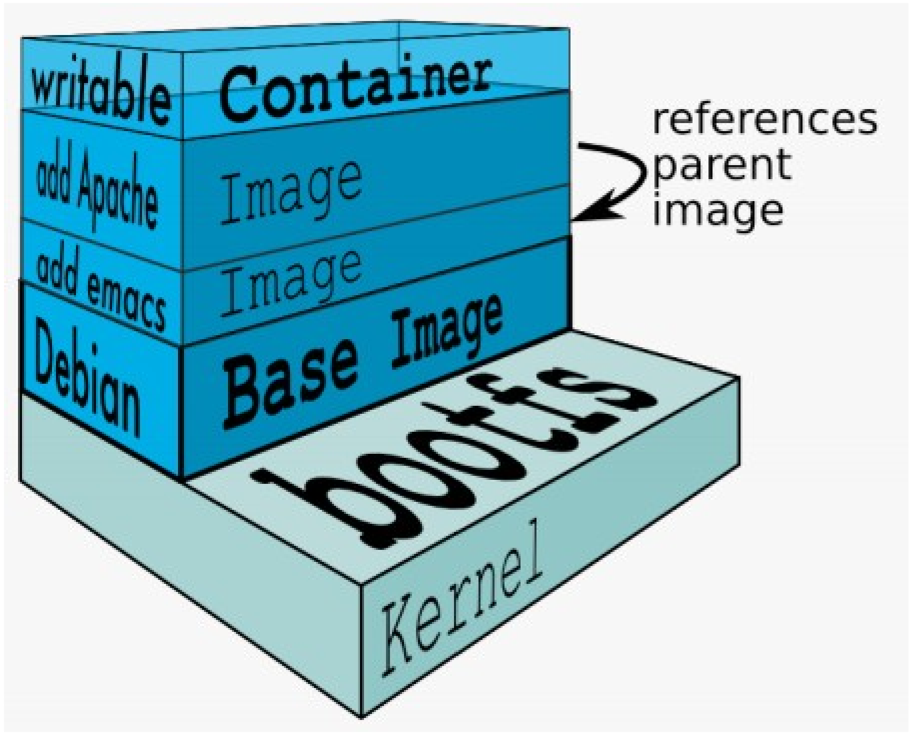
Docker Architecture



Docker Images

- Images are read-only templates used to create containers.
- Images are created with the docker build command, either by you or by other docker users.
- Images are composed of layers of other images.
- Images are stored in a Docker registry.

Docker Image Layers



Docker Containers

- If an image is a class, then a container is an instance of a class (runtime object).
- Containers are lightweight and portable encapsulations of an environment in which to run applications.
- Containers are created from images. Inside a container, it has all the binaries and dependencies needed to run the application.

Dockerfile and Instructions

- A Dockerfile is a text document that contains all the instructions users provide to assemble an image.
- Each instruction will create a new image layer to the image.
- Instructions specify what to do when building the image.
- It is recommended to chain the RUN instructions in the Dockerfile to reduce the number of image layers it creates.

Dockerfile CMD Instructions

- CMD instruction specifies what command you want to run when the container starts up.
- If you don't specify CMD instruction in the Dockerfile, Docker will use the default command defined in the base image.
- The CMD instruction doesn’t run when building the image, itonly runs when the container starts up.
- You can specify the command in either exec form which is preferred or in shell form.

Docker Container’s Links

- The main use for docker container links is when we build an application with a microservice architecture, we are able to run many independent components in different containers.
- Docker creates a secure tunnel between the containers that doesn’t need to expose any ports externally on the container.

Docker Compose

- Docker compose is a very handy tool to quickly get docker environment up and running.
- Docker compose uses yaml files to store the configuration of all the containers, which removes the burden to maintain our scripts for docker orchestration.

Docker Registries and Repositories

- A registry is where we store our images.
- You can host your own registry, or you can use Docker's public registry which is called DockerHub.
- Inside a registry, images are stored in repositories.
- Docker repository is a collection of different docker images with the same name, that have different tags, each tag usually represents a different version of the image.

PaaS

- OpenShift
- Rancher
- Amazon EC2 Container Service
- Azure Container Service

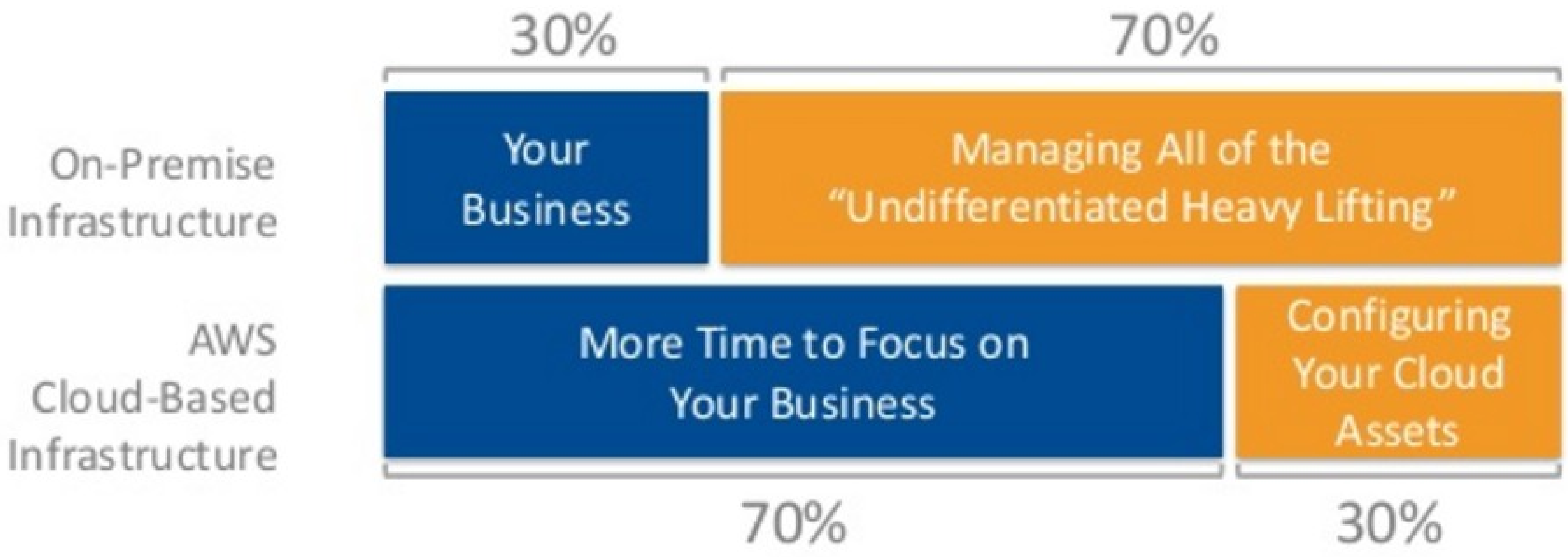
Boxfuse

- Spring Boot supports boxfuse to deploy application to VirtualBox and AWS
- https://boxfuse.com/

Cloud Computing (1)

- Cloud computing - access to technology resources managed by experts and available on-demand. You simply access these services over the internet, with no up-front costs and you pay only for the resources you use.
  - No capital expenditure
  - Pay as you go and pay only for what you use
  - True elastic capacity (scale up and down)
  - Improves time to market
  - You get to focus your engineering resources on what differentiates you vs. managing the undifferentiated infrastructure resources

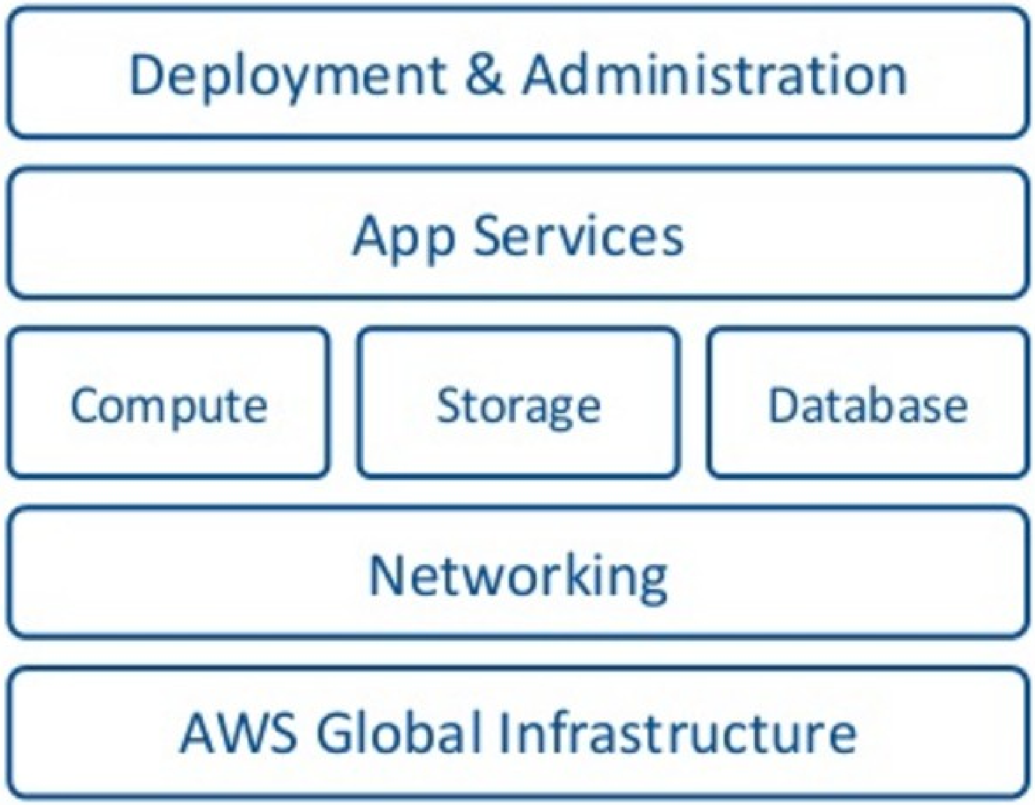
Cloud Computing (2)


30%
70%
On-Premise Infrastructure
Your Business
Managing All of the "Undifferentiated Heavy Lifting"
AWS Cloud-Based Infrastructure
More Time to Focus on Your Business
Configuring Your Cloud Assets
70%
30%

AWS Global Infrastructure

- Geographical regions - consist of one or more Availability Zones, are geographically dispersed, and will be in separate geographic areas or countries.
- Availability Zones - distinct locations that are engineered to be insulated from failures in other Availability Zones and provide inexpensive, low latency network connectivity to other Availability Zones in the same Region. By launching instances in separate Availability Zones, you can protect your applications from failure of a single location.
- Edge Locations - location is where end users access services located at AWS. They are located in most of the major cities around the world and are specifically used by CloudFront (CDN) to distribute content to end user to reduce latency.

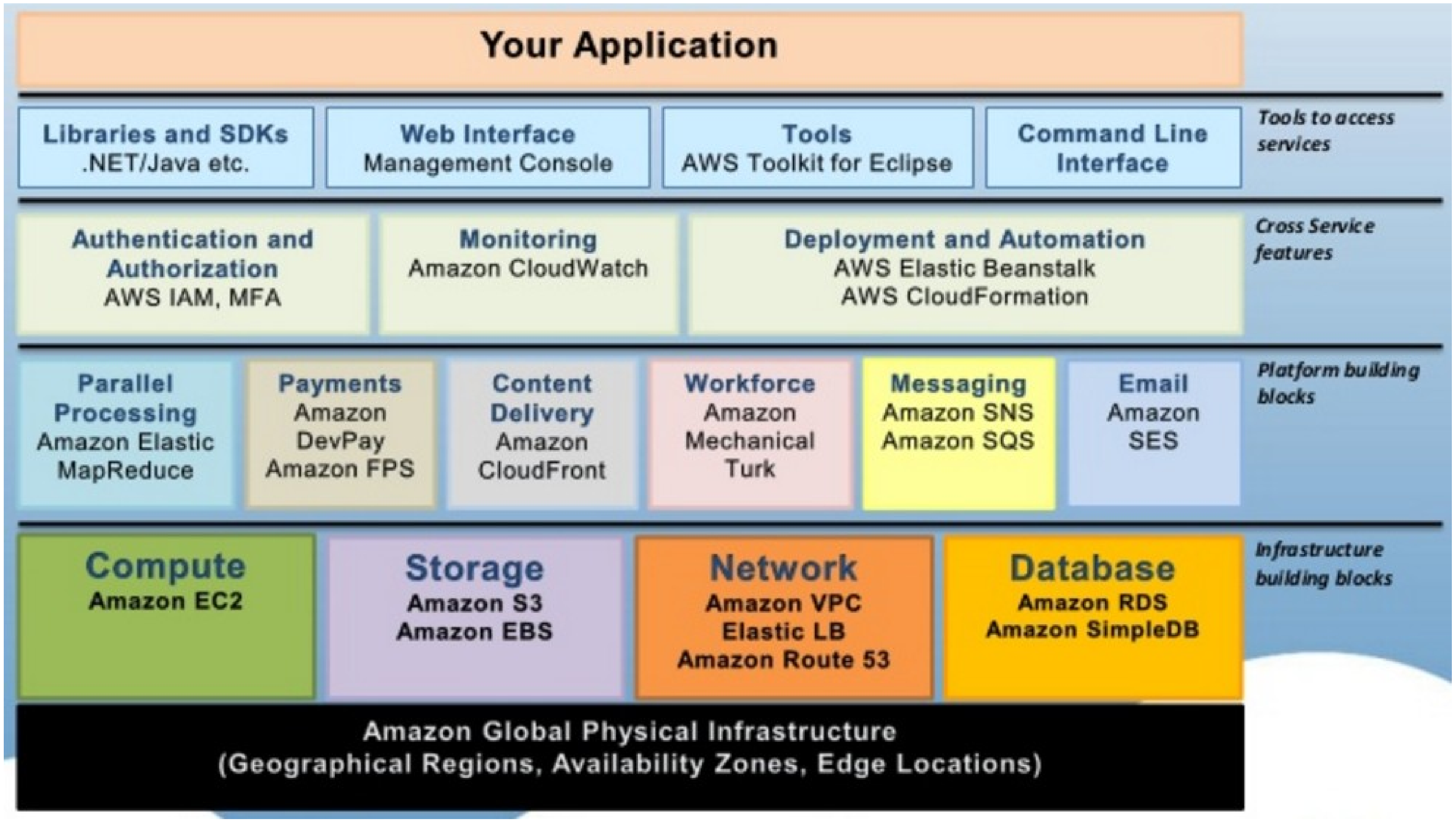
AWS Reference Model



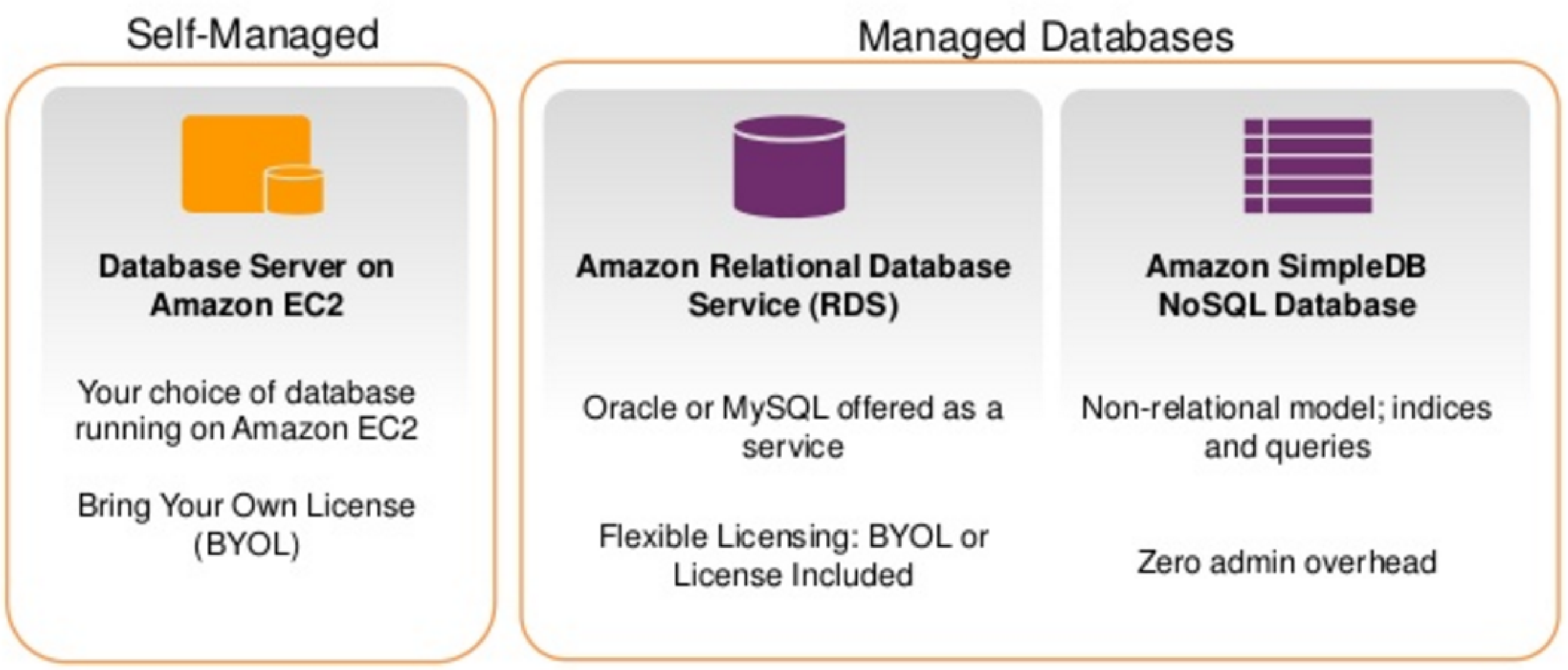
Amazon Web Services (1)

- Amazon EC2 (Amazon Elastic Compute Cloud)
- Amazon S3 (Amazon Simple Storage Service)
- Amazon EBS (Amazon Elastic Block Store)
- Amazon VPC (Amazon Virtual Private Cloud)
- Elastic LB (Elastic Load Balancing)
- Amazon Route 53 (Amazon DNS service)
- Amazon RDS (Amazon Relational Database)
- Amazon SimpleDB
- Amazon DynamoDB
- Amazon SQS (Amazon Simlpe Queue Service)
- Amazon SWF (Amazon Simple Workflow Service)

Amazon Web Services (2)


Your Application
Libraries and SDKs .NET/Java etc.
Web Interface Management Console
Tools AWS Toolkit for Eclipse
Command Line Interface
Tools to access services
Authentication and Authorization AWS IAM, MFA
Monitoring Amazon CloudWatch
Deployment and Automation AWS Elastic Beanstalk AWS CloudFormation
Cross Service features
Parallel Processing Amazon Elastic MapReduce
Payments Amazon DevPay Amazon FPS
Content Delivery Amazon CloudFront
Workforce Amazon Mechanical Turk
Messaging Amazon SNS Amazon SQS
Email Amazon SES
Platform building blocks
Compute Amazon EC2
Storage Amazon S3 Amazon EBS
Network Amazon VPC Elastic LB Amazon Route 53
Database Amazon RDS Amazon SimpleDB
Infrastructure building blocks
Amazon Global Physical Infrastructure (Geographical Regions, Availability Zones, Edge Locations)

Amazon Web Services (3)


Self-Managed
Managed Databases
Database Server on Amazon EC2
Your choice of database running on Amazon EC2
Bring Your Own License (BYOL)
Amazon Relational Database Service (RDS)
Oracle or MySQL offered as a service
Flexible Licensing: BYOL or License Included
Amazon SimpleDB NoSQL Database
Non-relational model; indices and queries
Zero admin overhead

Module 7
Monitoring Microservices

Importance of Monitoring

- Distributed loosely coupled services
- Asynchronous and event driven
- You should monitor health of service
- Services should provide useful metrics
- Services should log important messages

Spring Boot Actuator

- Provides useful HTTP endpoints for application monitoring and interaction as `/health` and `/metrics`

`env`	exposes environment properties
`health`	Shows application health information
`info`	Displays arbitrary application info
`metrics`	Shows 'metrics' information for the current application
`dump`	Performs a thread dump
`autoconfig`	Displays an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied
`auditevents`	Exposes audit events information for the current application
`loggers`	Shows and modifies the configuration of loggers in the application

Spring Admin Server

- Management and monitoring Spring Boot applications
- Supports Eureka discovering
- Uses Spring Boot Actuator endpoints

Monitoring Solutions

- Prometheus - https://prometheus.io/
- InfluxData - https://www.influxdata.com/
- Grafana - http://grafana.org/

Monitoring Solutions

- Prometheus - https://prometheus.io/
- InfluxData - https://www.influxdata.com/
- Grafana - http://grafana.org/

Log aggregation

- Services are distributed and running on multiple machines
- Each service logs about what it is doing in a standardized format
- You need to understand the behavior of an application as a whole and troubleshoot problems
- You need centralized logging solution

Logging Solutions

- Graylog - https://www.graylog.org/

Module 8
Microservices Security

Authentication

- Authentication is the process of determining whether someone or something is, in fact, who or what it is declared to be.
- Logically, authentication precedes authorization.
- It is usually done at API Gateway

Authorization

- Authorization is the process of giving someone permission to do or have something.
- In multi-user computer systems, a system administrator defines for the system which users are allowed access to the system and what privileges of use.
- Its is done on single service.

Single Sign-On

- You should avoid design where each service must contact Identity provider and Authenticate calls
- JWT – JSON Web Tokens
- OpenID Connect build on top of the OAuth 2.0

Module 9
Other Microservices Patterns

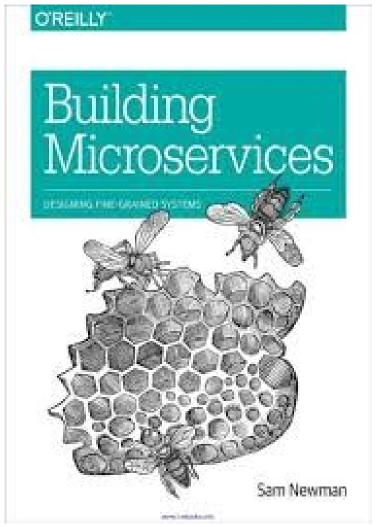
Externalized configuration

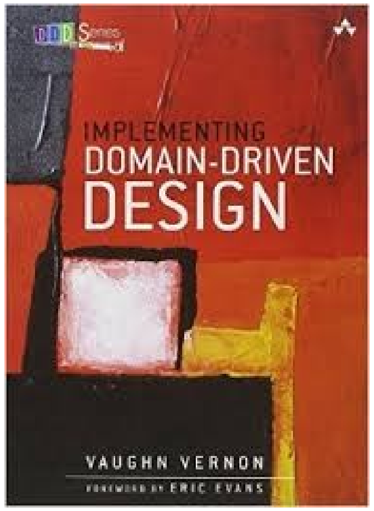
- Decouple configuration data from service
- Externalize and centralize configuration
- Supports multiple environments (DEV, QA, PROD)
- Configuration versioning
- Spring Boot Config Server

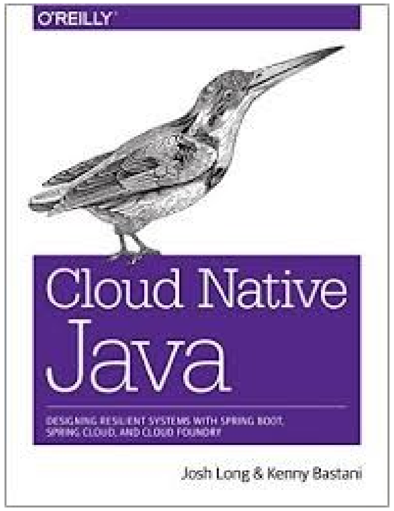
References

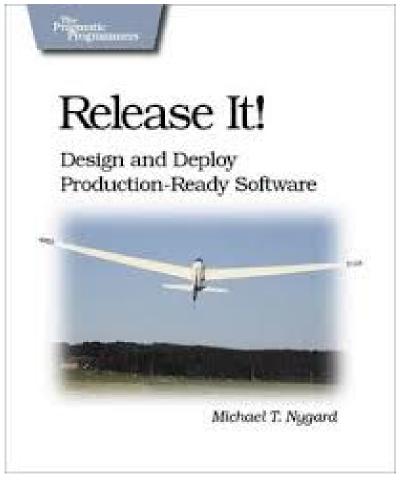
- http://microservices.io/
- https://microservices.io/patterns/
- https://martinfowler.com/articles/microservices.html
- https://martinfowler.com/microservices/

Books


O'REILLY
Building Microservices
DESIGNING FINE-GRAINED SYSTEMS
Sam Newman


DDD Series
IMPLEMENTING DOMAIN-DRIVEN DESIGN
VAUGHN VERNON
FOREWORD BY ERIC EVANS


O'REILLY
Cloud Native Java
DESIGNING RESILIENT SYSTEMS WITH SPRING BOOT, SPRING CLOUD, AND CLOUD FOUNDRY
Josh Long & Kenny Bastani


The Pragmatic Programmers
Release It!
Design and Deploy Production-Ready Software
Michael T. Nygard