microservices are small units of service created after breaking down monolithic apps- loose coupling between independent processes. done for

1. greater agility

2. scalability

3. availability

less code, better to maintain, scales up easily, independent deployment, better fault management

Spring Cloud

* provides MS infrastructure for
  + service registration
  + service discovery
  + configuration server
  + monitoring
* uses Spring Boot approach starters
* provides tools for devs to build distributed systems
* Spring Cloud Config Server –gives centralized control
* meant as one-stop-shop for MS

Spring Cloud Projects-

* SC Config, centralized management service for distributed systems, horizontally scalable, app configuration changes applied dynamically
* SC Bus, enables communication and messaging between services
* SC for Cloud Foundry, to deploy on Cloud Foundry
* SC Cluster, maintain cloud cluster
* SC Consul, better registry and discovery
* SC Stream,
* SC Data Flow, data flow management
* SC Task, manage and schedule tasks
* SC CLI, facilitates interaction with these components
* Event Bus to Client Bindings

Core Spring Cloud features

* service discovery – ensures services are discoverable
* circuit breaker – pattern to ensure services are granular and independent (faults are isolated)
* configuration server – centralized management
* API gateway – between client and Spring Cloud
* consumer-driven contract – driven by consumer policies
* business application – core where services can be exposed
* routing
* distributed messaging – intercommunication between services
* Auth Service- auth service provided by SC security using OAuth2 tokens
* Load Balancing- Netflix OSS provides tools for load balancing and failover
  + Ribbon- client-side load balancer
  + Hystrix- controls latency and failure

Cloud Governance

* schedule policies- part of evaluations
* controlled stack templates – build for underlying infrastructure (on premise or cloud)
* LDAP and SAML integration for RBAC -security
* auditing and budgeting – auditing internal activities
* resource tagging –optimization of resources
* dashboard and reporting – make user-friendly
* utilization – resource utilization for scalability

|  |  |
| --- | --- |
| Microservice Concern | Spring Cloud and Netflix OSS |
| configuration management | Config Server, Consul, Netflix Archaius |
| service discovery | Netflix Eureka, Hashicorp, Consul |
| load balancing | Netflix Ribbon |
| API gateway | Netflix Zuul |
| service security | Spring Cloud Security |
| centralized logging | ELK Stack (Elasticsearch, Logstash, Kibana) |
| centralized metrics | Netflix Spectator and Atlas |
| distributed tracing | Spring Cloud Sleuth, Zipkin |
| resilience and fault tolerance | Netflix Hystrix, Turbine, and Ribbon |
| autoscaling and self-healing |  |
| packaging, deployment, and scheduling | Spring Boot |
| job management | Spring Batch |
| singleton application | Spring Cloud Cluster |

Dependency options (from pivotal)

* spring-cloud-starter-bus-amqp
* spring-cloud-starter-cloudfoundry
* spring-cloud-starter-eureka
* spring-cloud-starter-eureka-server
* spring-cloud-starter-hystrix-dashboard
* spring-cloud-starter-turbine
* spring-cloud-starter-zuul

Spring Cloud provides out-of-the-box centralized management

MS collaborate at the process level while Spring collaborates at the component level

Client Discovery

* @EnableEurekaServer
* applications.properties or application.yml

1. eureka.instance.hostname=localhost
2. eureka.client.register-with-eureka=false /because this is the discoverer
3. eureka.client.fetch-registry=false /same
4. server.port=8080

* @EnableDiscoveryClient
* application.yml file-

spring:

application:

name: accounts-microservices

eureka:

client:

serviceURL:

defaultZone:

server:

port: 2346

SOA principles applied to Microservices

* distributed/versioned configuration – responsible for maintaining changes
* service registration and discovery routing – will use a registry server to register them and the client will be connecting to that registry server to discover the service
* service-to-service calls – must be able to communicate for when there is a collection of tasks in one user scenario
* load balancing
* circuit breaker
* asynchronous distributed messaging – to prevent blocks

ESB – Enterprise Service Bus

* integration tool – ensures integration patterns are implemented
* integration patterns provide a concrete approach of routing and mediating between services
* ESB coordinates the actions or events of the services from a set of heterogeneous systems that we must integrate and present to the upper layers
* integration diversification, centralized governance

Java Spring Cloud Microservice Example

* microservice architecture 12 factor app technology?
* service discovery – Eureka or Consul
  + Gateway Service: Zuul –incoming request management
  + Registry service: Eureka
  + Dynamic properties: Archaius
  + REST client: Ribbon – scalable and fault tolerant
* API gateway
* polyglot persistence – data and state management with different types of databases
  + key/value – rapid access for reads/writes
  + noSQL – high volume reads
  + SQL – transactional data
  + file – distributed file system
* integration testing using Docker

Microservice Architecture Principles

* OOP – object oriented programming
* Web service/API – expose app functionalities without a UI (provided by REST or web services)
* SOA (service oriented architecture)– structuring-related applications to work together rather than solving all problems in one app
* Systems- collection of parts that work together for a wider purpose
* Single Responsibility Principle
* Interface Segregation Principle (ISP) - code with defined boundaries

Interservice Dependencies in MS

* data sharing in MS can be complex
* services modeled as isolated units
* requires data sharing

Data Types

* Static data- usually read but rarely modified
* Mutable data- shared data, what to do when it changes
* common solutions
  + shared database
  + another microservice
  + event/subscription model
  + data pump model

Microservice Architecture

* the communication between apps and components creates a business solution
* modifications done efficiently
* organized around business domains
* smart endpoints (and dumb pipes)
* each MS provides an API endpoint (usually stateless REST API)
* use of APIs make communication standard
* easy for developers to consume
* most of technology is open source

System Design

* persistence layer or storage is encapsulated together with the processing part of the app- in one container we have the API that is exposed to other services and in other we have data access layer (actual data from a db)
* in a MS architecture, the persistence layer and store comes along with the module
* communication between modules should be well defined

SOA Implementation

* Layers
  + access or interface layer
  + business logic layer
  + service layer
  + data layer
  + service components
* Popular Styles
  + Gateway using APIs
  + Message Broker Style
  + Point-to-Point style (security risk)

Event Sourcing

* object state is determined by events
* read-only events stored
* events can be replayed
* advantages
  + answers “how” questions, gets the full picture
  + auditing
  + integrity
* 2-phase commit process (read and then write)
* current state of data
* (no) historical account of state
* databases are distributed
* NoSQL databases- no support for ACID transactions or 2PC (2-phase commit)
* events trigger other events

CQRS – Command Query Responsibility Segregation

* separating read operations from update operations
* gives advantages in performance, scalability, and security
* CQRS will help with- CRUD operations on same db could lead to mismatches, locking of records, other security issues
* one model initiates all queries
* a different model manipulates data
* separate physical databases give further enhanced performance, better scalability, reliable security

Microservice Architecture Patterns

* Aggregator Pattern
  + coordinates requests for all services
  + simple pattern
  + reduce inter-service communication
* Proxy Pattern (similar to aggregator)
  + requests direct by proxy service
  + good control of data
  + does not do processing unlike aggregator
* Chained Microservice Pattern
  + one entry point
  + every microservice depends on at least one other MS
  + used for very simple designs because of reduced speed
  + good security
* Branch Pattern
  + based on aggregator pattern
  + simultaneous requests
  + once a request comes in, it will spawn a chain of other requests
* Shared Data Pattern
  + anti-pattern because there is some coupling
  + good for converting from monoliths

Continuous Delivery

* shorter build, test, and deploy cycles
* services can be replicated almost instantly
* scaled using pools, clusters, and grids
* adapt to an agile environment

Dependencies in Microservices

* design with dependency in mind
* services calling each other
* data sharing- data can be static (easy sharing) or mutable
* one ms depends on another, may or may not involve access to db
* considerations- single db, large volumes, locking
* database changes

Cohesion and Coupling

* strive for low coupling but high cohesion
  + direct correlation between the two concepts
* choreography pattern
  + fewer direct requests between services
  + services wait for events taking place in the environment
  + react to events

Microservice Communication

* typically, API gateway handles all or most requests
  + web services free from any particular technology
  + more secure
  + loosely coupled
  + rolling updates possible
  + handles authentication
  + handles caching
  + handles static responses
* direct communication possible
* a load balancer can direct traffic
* client must know the service’s entry points
* may lead to tight coupling ?
* inter-service communication
  + mixture of protocols
  + different message formats
* XML, JSON, and binary transfer possible
* trouble handling partial failures

Inter-process Communication

* includes all communications within app that does not directly involve consumer
* interaction is based on –
  + one-to-many relationship
    - publish/subscriber
    - publish/async
  + one-to-one:
    - request/response
    - notification
    - request/async response
  + synchronous or asynchronous
* for object communication, mostly use asynchronous pattern using REST, JSON, and/or XML
* also synchronous pattern using REST

SOAP- glue between applications

* applications become accessible to each other
* uses XML and calls can be made over HTTP
* ignore strict hardware vendor restrictions
* run on any operating system
* can be written in any language

Docker

* container platform that can be used anywhere
* behaves like virtual machine but lean and lightweight (more efficient)
* can run multiple containers in one host
* secure
* Docker engine manages everything else inside of the container
* Docker Store (or Hub)- anyone can get/store images
  + cloud based registry services
  + keeps track of change management
* portable and lightweight (no hypervisor layer)
* easily replicated
* Advantages
  + faster start up time since it’s a process not a virtual machine
  + faster deployment
  + easier to manage
  + robust scaling
  + can use multiple operating systems running on same machine

Related apps

* OCI Open Container Initiative
  + attempt to create standards for the container industry
* Kubernetes
  + open source system for automating deployment, scaling, and management of containerized applications
* CoreOS – container security
* Mesos- Apache project for distributed Linux systems kernel
* Canonical – Ubuntu installation

Docker Installation

* docs.docker.com 🡪 Get Docker
* after install, to create new container-$ docker build –t NAME
* $ docker images
* $ docker run –p 81:80 NAME

Docker Toolbox

* includes Docker Machine- tool for configuring hosts
* Docker Engine(main part) – client server application that is made up of a Docker daemon, a REST API for interfacing the daemon, and a command line interface
* Docker Compose- tool for managing more than one running container app with YAML file
* Docker GUI (Kitematic) and Preconfigured Shell- legacy apps (no longer needed)
* Oracle VirtualBox – supports virtualization
* Docker Notary CLI – uses a key management and metadata for publishing and managing content that is verified creators can create their content and digitally sign it. aka you can trust that the content is what it says it is

Containers

* support virtualization
* just another process
* run several images
* images are lightweight and standalone
* image is like a template
* it includes the application plus required binaries
* prebuilt images already exist on the Docker Hub
* multiple instances of images can be creates
* a running instance of an image is called a container
* an image/container consists of layers
* it leverages all the resources, memory, and networking capabilities of the OS
* applications become portable, distributed, function specific, faster on startup

Dockerfile

* starts from a base image
* plain text file that contains instructions
* executed by the Docker daemon
* may include comments
* used by Docker Build
* label images with tags
* Dockerfile Basic Commands
  + FROM – loads an image
  + WORKDIR - sets up a working directory
  + EXPOSE – identifies a port for app to listen on
  + CMD – runs the app and passes parameters
  + ENV – sets up environmental variables
  + ADD – reference other files
  + ENTRYPOINT – targets and runs a specific application
  + MAINTAINER – depreciated but used for metadata
  + VOLUME – maps a directory on the host machine to a directory on the image
  + COPY – references other files, same as ADD

Deploying Microservices in Docker

* Docker Daemon – acts like a server process
  + builds and runs applications
  + listens for requests via a RESTful API
  + manages images, containers, network messages, and volumes
  + messages are sent to other daemon processes
* Docker client
  + sends commands to the deamon
* Docker registry
  + stores images
* Image
  + contains instructions for creating containers
* Docker run command
  + instantiates a container
  + provides a read/write file system
  + allocates an IP address
  + creates a network interface
  + starts up other services

Kubernetes

* open-source platform that helps with automatic deployment of our containers, scaling, and management when used with container apps (like Docker)
* also load balancing, monitoring of our apps, and more
* runs application containers on clusters (clusters can be run on physical or virtual machines or a mixture)
* logical objects- pods are single processes in a cluster
* service- defines a logical collection of pods
* volume- supports file retention in case of crash and file sharing between containers
* namespaces- used with very large teams and method to separate resources
* organizes a number of pods together and sets their access from the outside world
* have name of service and target port
* selector(name ) will be constantly checked
* services are discoverable via environment variables and DNS
* publishing services also possible

Connecting Applications and Services

* apps within container, container in pod, pod in node, node in cluster
* a node can be a physical or virtual machine
* node controls pods via services like
  + Docker
  + kubelet
  + kube-proxy
* pods contain running instances of container(s)
* a Kubernetes cluster is a combination of nodes
* a cluster contains a master
* Services control networking between pods, allows external access to pods
* kubectl gives us access to Node’s name and Node’s IP address
* a service views a collection of pods as a single entity
* an IP is given to the service
* pods communicate with the services
* incoming calls are automatically balanced
* kubectl is used to expose port on each app
* each container has a target port that accepts external requests
* the (selector) label makes the connection between the target port and app’s port
* a service controls pods that have reachable endpoints
* endpoints are contained in an endpoint object
* pods can be deleted and re-created anytime, but the endpoints will not change
  + or can use a DNS cluster add-on service to communicate between apps
  + running nslookup should return internal IP
  + signed certificates can then be used to secure the app via HTTPS

Kubernetes Cluster Administration

* install kubectl
* $ sudo curl –LO <https://asdf>
* $ sudo chmod +x
* $ sudo mv ./kubectl
* $ apt-get update
* $ sudo apt-get install –y kubelet kubeadm

EXTRA- install PHP from Apache

* $ sudo apt-get install docker-ce
* store.docker.com – search for PHP, select, scroll down and find Apache file
* create Dockerfile – FROM php:7.0-apache (copied from website)
* $ cd Documents/demo
* $ sudo docker build .
* $ docker run –name php7a –p 81:80 php:7.0-apache
* $ docker ps (shows is running)
* $ docker cp /home/sf/Documents/demo/api.php php7a:/var/www/html
* $ docker exec –it php7a /bin/bash
* open brower- go to local host port 81

Service Oriented Architecture (a few components)[slowest] –-> Service Based Architecture (about a dozen components) 🡪 Microservices Architecture (100s)[fastest]

Key Characteristics of Microservices

* domain-driven design (HR, sales, and production for example)
* polyglot (use different dbs and languages)
* single responsibility principle
* well defined and explicitly published interfaces
* functional decomposition
* any design pattern should take these into account

Pattern Language

* connects similar existing patterns together to solve a problem
* pattern structures have
  + name
  + context – explains the situation, requirements, componets
  + problem
  + forces – the actors involved and considerations
  + solution

Patterns

* Decomposition Patterns
  + Business capability- develop services based on business functions (inventory management, sales, production, etc)
  + Subdomain
    - Core – key differentiator for the business and most valuable part of app
    - Supporting – related to what business does
    - Generic – not specific to the business
* UI Patterns
  + server-side page fragment composition or server-side aggregation- individual teams provide data for pieces or fragments that will eventually be combined and form the UI page
  + teams supply complete user interface fragments, aka not just the data but the visual aspect as well
* Testing Patterns
  + service component test
  + service integration contract test
* Cross-cutting concerns
  + Microservice chassis
  + externalized configuration
* Security
  + access token
* Communication style
  + remote procedure invocation
  + messaging
  + domain-specific protocol

Domain-driven Domain (DDD)- best for complex situations

* each domain has a bounded context (each bounded context eventually becomes MS)
* methodology for building a business domain-centric application
* specific to domain and branching out to other scopes of the application
* focus on the center of a domain and domain logic
* creating software that addresses complex domain
* effectively refine conceptual model which addresses domain issues
* domain concepts implemented in software
* programming- no set methods or tools
* facilitation of collaboration between domain experts
* Bounded Contexts
  + central to DDD
  + the larger the domain, the more complex the design becomes
  + interrelated elements of a domain that have common connections or functions
* a guide to deciding on where the boundaries are- helps better separate the smaller pieces of MS using DDD concepts
* domain modeling helps identify and separate out interconnected parts

Designing for Failure

* Potential risks associates with MS
  + network calls and latency
  + increased complexity
  + external services
* graceful service degradation
  + contain failure
  + failover logic
* change management
  + automatic rollouts
  + use multiple production environments
* load balancing
  + gather health information to avoid unhealthy instances
  + use external system to monitor health and restart broken instances
* failover caching (failures are temporary so show can go on)
* retry logic
* want services to fail separately
  + use bulkhead pattern to protect services by separating them
  + use circuit breaker pattern opens when problem arises and stops requests to broken service – closes after specific time for service to recover
* fail quickly instead of waiting for timeouts
* test thoroughly and test often

Fundamentals for Success

* scoping functionality
  + prioritizing
  + logical functionality
  + organizational structure
  + set limits
* API presentation – exposed APIs (do last)
* Managing traffic
  + monitor
  + responsive API
  + graceful exit
  + volatility (delete and generate instances when needed)
* Offloading data
  + always on
  + shared redundancy
  + data caching
* Effective Monitoring
  + flexibility
  + storage and metrics
  + in real-time

Deployment Strategies

* Incremental – release new bits in small pieces
* Rolling – can rollback to last deployment if there is problem
* Parallel – same time, switch over
* Fault Tolerance
* Caching
* Automation
* Testing
* Integration
* Security
* CQRS
* Serverless Architecture

Microservices Service Discovery

* monoliths apps have static locations
* network location of these APIs must be known
* most services expose a REST of Thrift API
* MS deal with dynamically allocated IP addresses
  + instances start up and shut down often (scalable)
* registry of services – a db to handle requests to MS instances
* registry advantages
  + ensures that services are always reachable
  + handles self-registration
  + handles third-party registration
  + discovery possible via client and/or server side
* client-side discovery pattern
  + interrogates service registry
  + seeks out service locations
  + performs load-balancing tasks
  + less complicated
  + strong binding, client to service registry
* server-side discovery pattern
  + architecture includes load balancer
    - accepts incoming requests
    - interrogates the service registry
  + cloud-based load balancers
  + allows client code to be simpler
  + requires router
  + requires fewer network hops

Integration Testing

* feedback is important
* bugs can be detected quicker
* developers are updated
* short iterative cycles, extensive automated testing, low/zero cost to test, tests can replace documentation
* unit tests for business logic
* integration tests for environment
* integration tests depend on external resources and are complex
* separate testing suite for unit vs integration tests
* keep records
* test communication between services
* subsystem
* granularity

Continuous Delivery Pipelines

* software engineering pattern
* obtaining feedback
* process must be repeatable and reliable
* automate where possible
* version control
* include the persistence layer
* build binaries only once
* smoke test (basic testing that ensures software as a whole is functional)
* deploy to right environment (should match as close as possible)
* thorough integration (ongoing)

Continuous Delivery vs Continuous Integration

* CD pushes updates to users
  + CI updates code base
* product and test environments coordinate
  + triggers a build
  + risk reduced

Contract Testing

* ignore the body of components (already functionally tested) and look at whole app
* what happens from the user end
* simulates API calls to the product and make sure each response is as expected
* testing communication level, not behavior

Functional Testing

* unit as whole

Performance Testing

* deployment strategies
* event-driven architecture
* standardization
* logging
* monitoring
* cloud-based MSs
* Functional- JMeter, InfluxDB, Grafana
* Visualization

MS Scaling

* scalability vs performance
* complexity
* physical resources
* scale parts not the whole app
* X-axis scaling – divide work between involved MSs
  + load balancer
  + simple and effective
  + fine-tuning apps
  + disadvantage- each app has access to db which requires more memory
* Y-axis scaling – app split into multiple services
  + verb-based decomposition
  + noun-based decomposition
* Kubernetes automatically scales up (and down) a cluster
  + horizontal pod autoscaler
  + cluster autoscaling
* can depend on
  + machine resources (gradual increase/decrease)
  + high/low demand
  + queue length
  + business parameters
  + predictive
* advantages
  + eliminates downtime
  + flexibility
  + optimization and cost effectiveness
  + prioritize services

DB MS scaling

* state stored in tables
* capture events
* transactional records (must be persistent)
* transactional operations
* querying data
* normalization
* database variety
* three main models
  + table per service
  + schema per service – makes ownership of tables clear
  + database per service (higher overhead)
    - forces decoupling
    - freedom of database type
* application layer – use the app itself to manipulate data in db
  + performs queries
  + Command Query Responsibility Segregation CQRS
    - offers a different way of storing data
    - view based on data gleaned from events
    - event-driven db

Caching MSs

* caching- ability to store data or the content that was generated from a request
* read-write ratio – should cache data that is read more often than changed
* cache-hit ratio – a read can be satisfied by a cache
* stale- when data changes but cache does not reflect
* happens in memory (usually) for better performance
* locality of reference – certain data that is being accessed more often than other data
* local cache- cache written to disk

Monitoring Microservices

* logging – keeps track of data produced by MSs
* analytics – tools that make sense of the logs
* alerts – usually based on thresholds or limits, alerts us to any unhealthiness or failure
* main features:
  + monitor deployed containers and their contents
  + send messages via email or SMS
  + monitor services that automatically adapt
* monitoring several servers
  + end-user experience
  + end-to-end transaction tracing
  + application topology
  + service views
  + custom metric APIs
  + custom logging APIs
* MS monitoring hierarchy
* group of containers
* cluster nodes
  + CPU
  + Memory
  + swap space
  + file disk space
* processes
* container itself
* closed systems
* visibility limited
* built-in monitoring code
  + docker stats
* kernel level cGroups stats
  + memory includes gauges and counters
  + CPU usage by user and system
  + network metrics cia IPtables
* cAdvisor
  + interprets Docker stats
  + visual
* Prometheus – support for Kubernetes
* Agentless System Crawler
* SCOUT – hosted
* Data Dog
* Sensu Monitoring
  + open source server

ELK stack - Elasticsearch, Logstash, and Kibana

* Elasticsearch
  + analyzes the applications data and monitors activity
  + looks at the requests, how long the requests take, etc
  + helps define trends
  + RESTful interface
  + has libraries that can be added on
  + powerful search and querying feature
* Logstash
  + analyzes logs from the app
  + gives analysis in a graphical display
  + parses files like JSON traffic or NGINX or Apache log files
  + filters, outputs
  + handles variety of data from multiple sources
  + usually sends data to Elasticsearch (always in ELK stack)
* Kibana
  + graphical tool and dashboard
  + displays log data in simple rows
  + can run queries
  + monitors the system for peak periods and downtimes

MS Logging Tools

* Single or Multiple MS
* multiple log files
* historical data
* centralized logging

Logging Best Practices

* service identification
* use standard times
* request/transaction IDs
* grouping
* relevance
* event correlation
* track deployment
* set up alerts

Log Management

* Docker- two types of logs:
  + infrastructure management
    - system as a whole
    - is the app working?
    - overall response time
  + application
    - is each service performing optimally?
* Docker Engine logs-
  + logs are fed into the operating system manager
  + we can send it to a centralized logging service within system
  + Universal Control Plane – helps manage Docker systems
* Docker Trusted Registry
  + used to register images
* UCP and DTR system logs
* infrastructure logs include
  + monitoring
  + auditing
  + reporting
  + configuration details
* application log drivers- JSON, syslog, journal, gelf, fluentd, splunk
* Prometheus software .yml file needs-
  + target server
  + name of the job
  + time interval to check logs

MS General Security Consideration

* larger attack surface
* network is both vertical and lateral
  + east-west traffic – chattiness inside app between components
* securing APIs
* tools to secure against MS concerns-
  + standardization – Transport Layer Security TLS
  + least-privilege – MS given read only access if it isn’t used to modify data
  + application level
  + documentation
  + logging
  + authentication and authorization -like OAuth
  + encryption
  + automatic security updates
  + employ micro-firewall
  + API gateway
  + monitor (in real time)

MS Security Layers

* authorization
* authentication
* access

Security Best Practices for MS

* micro-firewall
* standardization
* encryption
* logging
* fine grain authorization
  + secrets
  + trusted perimeter
  + HTTPS

JSON web tokens- small lightweight, stateless, encrypted JSON object

* user identity
* authentication
* data transfer
* no need to reauthorize
* can configure access to various services
* channel does not need to be secure

Authentication and Authorization

* authentication- does this client have valid credentials (1st level)
* authorization- does the client have permission to perform the request
* access – extra authentication step- authority to perform actions
* API Gateway- 1st line of defense for MS
  + OAuth + JSON Web Token
  + access to resources
  + expiration/revocation (can be an issue)
* TLS
  + microservice certificate
  + application-level authentication
  + short-lived certificates
  + can use with OAuth
  + gives external and internal authentication

Fine Grain Authorization

* MS more vulnerable because of larger attack service and east-west traffic
* authentication- process of confirming the identity of the client or user
* authorization- process to decide if the user is allowed to perform the requested action (happens after authentication)
* trusted perimeter (zone where we trust all network activity)
* involve SSO- single sign-on system for internal interaction behind firewall
* HTTPS- better authentication than HTTP but adds overhead
* hash-based message authentication code- request body is hashed together with private key and sent along with requests and then the receiver recreates original hash
* secrets tools- to keep passwords, API keys, and certificates
* communication patterns- use to identify breach in security
* segmentation and isolation
* data flow encryption
* automation

Service to Service Auth’s

* challenges
  + constant change to MS
  + hidden resources
  + transaction trail
  + increased surface area
* solutions
  + service discovery – establish baseline patterns
  + isolated segment
  + encryption
  + automation
  + JWT
  + or SSL/TLS mutual authentication
    - trust
    - bi-directional
    - whitelist

Open Source MS products

* Containers
* Docker
  + Swarm – manages Docker containers
  + Kubernetes
* Monitoring
  + Prometheus
  + ELK
  + cAdvisor
* Istio – vendor-neutral way to connect, secure, manage, and monitor applications

Azure Service Fabric Overview

* for distributed system
* design time – everything needed to bring an app to market, XML files
* runtime – managing apps via clusters of nodes
* stateless and stateful
* partitioning, scaling, and availability
* SDK available
* containers (or processes)
* reliable services
* reliable actors – small independent units with single thread
* ASP.NET core
* guest executables
* lifecycle management
* independent scaling
* rolling upgrade
* uptime
* resource utilization
* Infrastructure Services
  + cluster manager – listens for RESTful HTTP traffic
  + failover manager – detects when nodes fail
  + naming – maps service instances to endpoints
  + fault analysis – test system for failure
  + image store – application packages
  + upgrade – automatic

Polyglots and MS – different dbs for different MS

* polyglot programming
* polyglot persistence
* Neo4j
* domain-driven design

Object Communication

* REST (protocol)
* JSON
* XML