Properties of Microservices

* each part should be independently replaceable
* built around business needs and functionalities
* has smart endpoints and dumb pipes
* components act individually avoiding the ESB model
* db management is decentralized
* infrastructure is automated
* MSs are continuously delivered
* they have a single responsibility
* are object oriented
* small
* monitored
* clustered

Client Layer

* can be about anything (web, mobile, IoT)
* knows nothing about service (or MS)
* interfaces with API layer
* works with existing protocols
* high reliability
* high security
* variety of programming languages

API Gateway

* exists on server side
* has to be modified to locate individual MS
* acts as a controller
* directs requests to encapsulated MS
* has standard APIs to redirect input to MS
* high security
* variety of programming languages

Microservice Layer

* small, lightweight pieces of software that only do one thing
* know about each other’s existence but not their functionality
* can be on different platforms
* can be in different languages
* receives input form the API, each other, or the back end
* can be individually replaced
* self-contained and independent
* each runs its own process without help

Back End

* still knows nothing of API and Client Layer
* can be a db(usually), web service, or any other destination
* knows only about the MS it communicates with
* can communicate back to the API via a MS
* should have messaging and logging capability
* may need to be modified to communicate with MSs
* talks to API
* never communicates directly with MS
* encapsulated
* data validation and cleansing

Advantages of Microservices

* ability to choose frameworks and tools
* individual MSs are interchangeable
* scale better than monoliths
* easier to debug
* support continuous integration
* are agile

Disadvantages of MSs

* distributed architectures are complicated
* refactoring monolithic services is messy
* parallel paths may be difficult to implement
* who will own the MS arch
* who will support it
* how will the MSs be deployed

Considerations-

increased complexity, network congestion and latency, data consistency, fault tolerance and resiliency, diagnostics, versioning, DevOps

Microservice Best Practice

* use common source control for all services
* try to mimic PROD in DEV
* push working code to the main branch often
* release less, release fast
* beware of shared libraries (they are needed, hard to updated, and hard to test)
* keep your services simple

Client Devices

web browsers, mobile devices, IoT, Web services, embedded devices, streaming sources, any other

Application Program Interface

* can be used to discover services
* act as a traffic cop
* message oriented
* multiplatform and multilingual
* interface with other APIs but never directly with MSs

Microservice Granularity – defines how exact or fine the grain of service is

a fine grain is more focused and has less functionality

atomic granularity refers to one large piece of code that contains all the functionality for a MS

Atomic Granularity

* one piece of code (not divided into smaller units)
* set of defined interfaces (because its hidden)
* two or three inputs and outputs
* code runs like a service
* each component runs in a separate process in own thread

Composite Granularity - large app broken down into smaller units

* has an outer service with a set of interfaces
* contains one or more ‘inner’ components
* runs as a separate process by default
* each inner component runs in a separate thread
* greater opportunities for reuse over atomic granularity

--------------------------------------------------------------

Microservices Architecture Design

transactional boundary- smallest unit of atomicity (should be as small as possible)

atomicity transaction is indivisible and irreducible series of db operations

(either all occur or none)

using DDD terminology we identify entities, value objects, aggregates

distributed database functionality- separate dbs for entity, values, and aggregate

big questions-

1. what is the domain? what is the reality?

what is data representing, no objective definition of data model, all about context

1. where are the transactional boundaries?
2. how should microservices communicate across boundaries? state/connection info/ failure info

keep true business invariants intact, maintain some form of consistency between aggregates

distributed systems are finicky, nonsychronized time boundaries

use events to communicate consistency (constant communication between microservices)

events are immutable structures that capture a point of time

1. what is we turn db inside out?

each have data capture and event handlers that give distributed, replicated event log to kafka, kafka passes to event handler

logging and monitoring- need to have holistic view of whats happening over dozens or hundreds of microservices

metrics are numerical values that can be analyzed

* node-level system metrics
* kubernetes metrics
* application metrics
* dependent service metrics

logs are record of events that occur while the application is running

logging considerations

* configuration and management

often automated or in the cloud

* ingestion rate
* cost (in resources)
* data fidelity (how true the data is)
* latency(how long it takes to get data from microservices into logging mechanism)
* storage
* dashboard and visualization (management techniques for bottlenecks or health of ms)

distributed tracing

* there is currently no standard HTTP header for correlation IDs
* should include correlation ID as metadata (info about log message)
* send a correlation context that includes richer information
* some services may still need to explicitly propagate the correlation headers
* consider how to aggregate logs

Technology Options

* Prometheus or InfluxDB
* Kibana or Grafana
* Fluentd and Elasticsearch
* Operations Management Suite (OMS) Log Analytics

Monolith vs Microservices

everything happens in Monolith boundaries

each MS has own private db (SQL or noSQL)

Microservices are event-driven

* a service publishes events when something notable happens
* other services subscribe to those events
* a server typically updates its own state
* use an event-driven approach to implement eventually consistent transactions

Order ManagementS ---create order---🡪 customer management s

OMService 🡨----- credit check approved 🡨------ CMService

ACID transactions are replaced by multi-step, event-driven, eventually consist workflow

API Gateway (single entry point) manages requests

traffic cop, divides client specific APIs and Protocol Translation(REST)

OR multiple API Gateway for Web/Mobile/Public

Monolith Service-based Architecture

* supports a variety of different clients including desktop browsers, mobile browsers, and native mobile applications
* applications may also expose an API for third parties to consume
* integrates with other applications via web services or a message broker
* applications handle requests (HTTP req and messages) by executing business logic

benefits of Monolith(simple to develop, deploy, and scale)

* new team members will quickly become productive
* application is easy to understand and modify
* want to practice continuous deployment
* you must run multiple copies of the application on multiple machines in order to satisfy scalability and availability requirements
* you want to take advantage of emerging technologies

negatives of Monolith

* overloaded IDE
* overloaded web container
* continuous deployment is difficult(1 small change, whole thing needs to be redeployed)
* scaling the application can be difficult
* requires a long-term commitment to a technology stack

Challenges of Change

* there is overhead associated with managing a large number of services as compared to just a few
* deployments distributed components need to be automated and well orchestrated
* monitoring infrastructure needs to support tracking communication flow across multiple services
* operations team needs to manage a large number of infrastructure units

Considerations and Realities of Change

* operational complexity
* deployment pipeline of distributed components
* monitoring distributed services
* testing distributed services
* communication between ds
* too many services/incorrect boundaries
* transferring functionality between microservices
* needs function and non-functional testing
* end-to-end testing of an entire system gets more complicated
* good knowledge of business domain is essential
* errors can be difficult to retrofit later
* less developers writing more code
* single vendor solutions will become alphabet soup
* more objects will be written that do less things

Elements of Change

* microservice architecture introduces its own complexity
* if you are not building a SaaS(Software as a Service, aka cloud) type services, you should start with a monolith
* its easier to split something up you already have

Business Reasons for Microservices

* programmer productivity
* faster development
* faster testing
* faster deployment
* easier to identify and track down bugs
* embrace failure (robustness)
* smart dynamic scaling
* high availability and robustness
* technical advantages directly translate into various business value propositions
* problems and costs in software development are related to the ability of the software system to change or evolve
* change is inevitable
* provide a competitive advantage to the businesses that adopt the change most effectively
* a system must embrace change in its fundamental design to keep evolving
* businesses change their models and directions
* technical solutions should represent business goals
* embracing a changing environment will maintain a competitive advantage

Opportunity Costs-

median cost of fixing software errors in the later phases (instead of earlier) of the software life cycle is greater by a factor of 50

Development Costs-

faster deployment, testing, and bug-fixing can significantly reduce lifetime costs of software projects

Optimization Costs-

further cost optimization come form the smarter utilization of infrastructure as individual parts of the entire solution can be scaled up and down independently

* evolvability

-rate of change

-speed of change

* time to market
* brand reputation
* reduced cost

-development

-infrastructure

* better hiring

Working In Parallel

* often a developing MS app is developed alongside a Monolith
* not one but two critical paths need to be defined
* different dev teams, will be used for each path
* developer skillsets will vary
* tools and vendors will not be congruent

Tools and Technologies

* Monolith path will have full stack tool implementation and support
* monolith path will need support but not development
* microservices path will employ alphabet soup of tools and technologies
* tools for each path will not work the same
* devs should not recreate the monolith with microservices

Requirement Types-

Function vs Non-functional

non-functional- things we need to get working that aren’t customer facing or functional

aka get Azure working with Kafka (time eaters in MS app)

Testing to validate that your system meets all the non-functional requirements

* concurrent requests
* transactions per second
* response times
* subservice failures

FR- Verbs, Mandatory, Captured in Use case, Product Feature, Easy to Capture

NFR- Attributes, Nonmandatory, Captured in Quality Attribute case, Product Properties, Difficult to Capture

* Applications and the ecosystems in which they run have changed drastically
* older apps used to run in closed, barely-distributed, named-pet-system environments
* with arrival of clouds and ms, ecosystem has changed drastically
* most testing is not even scoped in the functional requirement

NFR Testing Steps

----be agile, don’t wait until end

----plan

1. be requirement driven- locale all NFRs and map them to your tests
2. be fear driven- if something is new, test it out
3. prioritize

----setup

only prod-like truly counts – in Config, Data, Platform, and Monitoring(aka dev environment)

N.b run-state – Normal? peak Load? Failure/DR scenario? Latency?

----Perform:

run your NFR test

Microservices can be outsourced to cloud providers

Infrastructure as a Service (Iaas) (removes any operational overhead related to hardware)

Playform as a Service (PaaS) (additionally remove operational overhead)

SaaS(cloud based Software) made up of Docs, Files, and other Files

Moving to the Cloud

Modern software projects are embracing the cloud (Amazon Web Services, Microsoft Azure)

easy to deploy and operate software solutions on cloud infrastructure

feasibility of ms is supported by advances in virtulization

Amazon Web Services

CDN (content distribution network) allows us to push data geographically near users to reduce latency and access time to retrieve content

CloudFront Distributions

new distribution

web or RTMP (media files)

-------------------------------------------------------------------------

SOA (Service-Oriented Architecture) is platform-independent

Elements of XML are emphasized when used in SOA

SOA basics

Key Roles

* --Service consumer
* --Service provider
* --Service registry

Architectural approach that promotes

* reuseablity
* distributed deployment
* interoperability (ability of computer systems/software to exchange and make use of information)
* composablity (system design principle that deals with the inter-relationships of components) (a highly composable system provides components that an be selected and assembled in various combinations to satisfy specific user requirements)
* vendor independence
* programmatic interface/ Consumer

SOA is a collection of services

Typical SOA service is:

* logical representation of a repeatable activity/ set of services
* clearly defined
* self contained
* low dependency on other components/services “black box”
* platform independent (XML)
* discoverable
* quality of service characteristics defined in policies
* governable

Business Process composed of Activities/Tasks/Service fulfilled by Applications/Packages supported by Infrastructure

XML is a document format (eXtensible Markup Language)

* Office 2007+ and HTML are XML formats
* formal is usually called a schema
* data and attributes
* <tag attributeName=”AttributeValue”>data</tag>

SOA

* Envelopes
* HTTP
* for SOA, the emphasis in XML is on the elements rather than the attributes

SOAP- Simple Object Access Protocol: higher-level protocol (in XML format), is a way to encode multiple parameters or data values into a single XML document

WSDL- Web Service Description Language provides an easy way to deliver the metadata is running the services

metadata includes what services, operations, and data are available for use and what data is expected by the operations, and what data is returned by them (in XML format)

SOA technologies

XML- Extensible Markup Language, Used to Transport Data

Web Services- deliver loosely coupled services

SOAP- Simple Object Access Protocol

WSDL- Web Services Description Language

UDDI- Universal Description, Discovery and Integration global business registry that may include web services

WSI- Web Services Interoperability is a set of best practices for web services designed to operate across platforms

these standards fit within the TOGAF: The Open Group Architecture Framework, and enterprise architecture

Service Oriented Architecture

describes a system of software, its major components and info

includes

--organization of the system

--elements and interfaces and their behaviors

* scalability
* usage of elements
* reuse
* performance
* reliability

have to look at big picture

1. business processes- enterprise data
2. business services- consolidated data
3. integration services- integration data
4. enterprise resources- operational data

Typical Web Service-

exposes some core functionality

Utilizes- WSDL, XSD, XML, SOAP

SOA is a methodology, concerned with overall design (can be implemented with various technologies)

SOA Processes

New Process -> Model -> Implement -> Execute and Control -> Monitor and Optimize

People, Organization, Technology at Center

SOA Layers- Consumer to Provider (Best Practice Design)

1. Consumer Interfaces
2. Business Processes
3. Services
4. Service Components
5. Operational Systems

Integration, Quality of Service, Information, Governance

WCF- Windows Communication Foundation

WCF Service Application

System.Runtime.Serialization converts OO classes into String

System.ServiceModel provides for basic service functionality (usually through config file)

System.ServiceModel.Web provides functionality for receiving and sending messages

service contract defines what service will do

CompositeType

OperationalContract

XML

request-

in header- definition around what the service must do (specifies action)

in <s:Body> the operation is set with the message

XML serialization- process of converting operation parameters into XML

response-

deserialization – converting back

body matches request

Key Drivers for using WCF as SOA

* tracking
* monitoring
* extensibility
* use custom code to monitor activity
* use built in tracing and message logging
* use WCF performance counters

Windows Communication Foundation

services are stateless but we can store information within a database or file store

extensibility- message interception, message mapping, error handling

need strong tracking and monitoring for peak performance

Pii- Personal Identifiable Information

<messageLogging logEntireMessage=”true” logKnownPii=”true” logMalformedMessages=”true” logMessagesAtServiceLevel=”true” logMessagesAtTransportLevel=”true” maxMessagestoLog=”10000” />

<endToEndTracing propagateActivity=”true” activityTracing=”true” messageFlowTracing=”true />

<diagnostics performanceCounters=”All”>

WCF Credentials

DataMember = contains data

UserName.Password

Contract =

IIS = Internet Information Services

ClientCredentials.Username

security mode=”Transport” or

security mode=”Message”

transport clientCredential

basic sends username/password for credential

digest is basic plus password hashing

transport security focuses on network

message-level transport security focuses on WS Security standard around SOA communication

M-LS combines security claims, encryption and other plus flexibility

performance decrease with message-level security and does not support message streaming

OperationContract defines a specific activity that will receive and send messages

Three Main Service Patterns

1. Microservices Architecture

most granular, each microservice has sole ownership of their data

1. Service-oriented Architecture

loosely coupled resuseable code

1. Service-based Architecture

dozens of deployable services (may share data store)

Nonservice Patterns (common for monoliths)

* Layered arch
* Microkernel arch
* Event-driven arch
* Other

distributed architecture is-

elasticity, scalability, decoupled, complex + easy of deployment

harder to develop

Monolithic Architecture Patterns

* Layered arch
* Microkernel arch
* Pipes and Filters (Pipeline) arch

Distributed Architecture Patterns

* Event-driven a
* Space-based a (linear scalability)
* Microservices a (100s of small pieces of code)
* Service-oriented (predefined interface that is shared across program)
* Service-based (similar to microservices but not as many)

(communication types)

synchronous – waits for response

asynchronous –moves onto another task (for when you expect response to be delayed)

can be handled by single receiver or multiple receivers (must be asynchronous)

HTTP/s is synchronous

AMQP asynchronous

Communication Styles

* HTTP and Rest (rest is tightly coupled to HTTP protocol) most used
* Binary format communication mechanisms (WCF)
* Asynchronous message-based communication mechanism
* JSON or XML
* Binary formats

Decoupled Application Architecture

* each component to perform its tasks independently
* allow components to remain completely autonomous
* change in one services shouldn’t require a change in other
* a process of separating services
* functionality is more self-contained
* smaller code blocks are easier to mange an understand
* the parts of your application can evolve independently
* each service can be maintained by separate developer teams
* understand only a small part of your app

How to Decouple

* Achieve by simply adding a layer of technical abstraction
* a message queue or a well written interface
* message queues decouple your processes
* the sender and the receive agree on a common format for messages
* queues use the same message broker

Distribute Workload and Scalability

* message queue makes it easy to scale up a decouple app
* simply add more workers and receivers whose job is to work off the queues faster
* receiver can retrieve these message and process them
* message queues act as a buffer
* receivers can gradually drain the queue at their own pace

Decoupling Issues

Dependencies

* multiple teams work on same codebase
* there is little communication between teams
* will have dependency problems
* communication component still must be grown and improved as team size grows
* achieve decoupling by having a monolith composed of well-defined modules with the well-defined interfaces

Horizontal Scalability

* most companies adopt micro services too early
* no horizontal scalability at start
* the complexity that accompanies HS should only be a last resort
* first effort should be in taking reasonable steps to improve your app performance
* wisely use services that support your app

Rapid Provisioning

* developers should be involved in managing everything about their apps
* current architecture is not backed by a flexible infrastructure
* must be able to have flexibility on operations and deployment of services
* developers are usualy not allowed to provision
* rapid provisioning often means never provisioning

Increased Operational Overhead

* app can get slower
* ms often run on other machines and rquire a network hop betwee n your services
* some services need to contact multiple other services synchronously
* microservices have network overhead and other delays

Local development can be a nightmare

* containers wont save you from this level of complexity
* devs usually need to have all services running on their machine
* developing and testing new features is difficult
* development complexity is extremely wasteful

Properly Scoped Functionality

* partitioning the services along logical functionality lines
* mirror the dev org’s structure
* minimize a service to the amount of code that could be reimplemented by a team in a 2 week period
* Single Responsibility
* each ms should have well-defined API

Presenting an API

* typically done with REST web services API calls
* each services must be able to reliably send and receive data
* foundation of an API is exposing the service at a known location
* as individual services mature, they may add new functionality
* define what service must expose to operate properly (only what it must)

Traffic Management

* a service may run slowly
* service can be overwhelmed with calls and lack the processing power
* service might stop running to a software or hardware crash
* must be a way for calling and for called services to communicate status and coordinate the traffic loads
* API design should include the ability to send a response that indicates an overload

Data Offloading

* individual service instances are transient
* overall service must be available and continue operation
* the need for continuous operation is different from traditional apps
* insert a shared, memory-based cache system
* the caching system becomes another service in an app architecture

Docker is used to run microservice within a container

Service Dependencies

Managing Data Dependencies – critical questions

sharing db, handle volume, provide consistency, requires change of schema?

Problem Locality- related problems should be grouped together

Loose Coupling- should be able to modify 1 ms without affecting others

interaction between MS should be minimal, if not, try to merge into single service

Data Sharing-

1. Static Data – read but rarely or never modified
2. Mutable Data- boundaries between MS can be blurred (problem)

Distributed Logging Architecture

* centralize and externalize the storage of your logs
* log structured data
* generate correlation ID
* allow to change the logging level dynamically and make it asynchronous
* make it searchable

Architecture of Log Management

MSs -> Kafka (RabbitMQ or others) -> Logstash Cluster -> Elastic Search Cluster -> Kibana

MSs run on multiple hosts

messages generated by ms are distributed across multiple hosts

cumbersome process of grapping the logs

Microservices are explicitly built to expect failure so that when one service fails, the others keep working

Containerization provides a simple technology to package microservices into persistent deployable units

these containers need to be deployed, managed, and scaled in production

we have to have resilience, elasticity and recover

Microsoft Azure API Management

Developer Portal, API Gateway, Publisher Portal (connects to MSs and client) custom MS infrastructure

Amazon Web Services- Messaging (asynch and synchronous)

asynch

* Push-Pull : Amazon SQS
* Stream : Amazon Kinesis/DynamoDB Streams

synch

* Publish-Subscribe : Amazon SNS
* Request-Reply : Amazon API Gateway + AWS Lambda

Google Cloud

Management, Big Data, Storage, Compute, Connectivity, Dev Tools, Mobile

Docker- container management

virtual environment to spin up program within container, and then transfer to any system

Build / Ship / Run microservices in container

3 levels of independence

* decoupling via independent releases
* technological decoupling
* domain-based decoupling

How Do Clients Access Individual Services?

* network performance is different for different types of clients
* number of service instances and their locations change dynamically
* granularity of APIs is often different than client needs
* partitioning into services can change over time
* services might use a diverse set of protocols

Benefits of an API gateway

* security is implemented via the API Gateway
* insulates clients from how an app is partitioned into MSs
* insulates clients from the problem of determining the locations of services instances
* provides the optimal API for each client
* reduces the number of requests/roundtrips
* simplifies a client by moving logic for calling multiple services form the client to the API gw

Drawbacks

* increased complexity
* another moving part to be developed, deployed, and managed
* increased response time
* additional network hops through API gateway
* doesn’t always scale well

Direct Client-to-microservice Communication

* each ms has a public endpoint
* endpoints may have a different TCP port for each MS
* URL would map to the load balancer used in the cluster
* you could have an app deliver controller (ADC) between your microservices and the Internet
* adc acts as a transparent tier

Benefits of direct access service

* good design for a small ms-based app
* lends itself to server-side web app like MVC app
* nice for a small set of microservices
* simple design
* good design pattern when scalability is not an issue (wont grow)

Drawbacks of direct access service

* not good for large and complex ms-based apps
* cumbersome when handling dozens of ms types
* scales badly
* not a good choice for remote mobile apps or SPA web applications
* increases latency and complexity

Considerations for direct access

* How to handle cross-cutting concerns
* how can client apps minimize the number of requests to backend
* how can client apps communicate with non-Internet-friendly protocols?
* how can you shape a façade especially made for mobile apps?

Traditional/Hybrid Microservices

* applications are distributed
* apps are separately deployed
* maintina the concept of service components(within the monolith or external)
* may have 3-4 loosely coupled modules in one service component
* bounded context for the modules, if not the data
* service granularity
* shared db
* integration hub
* message bus

A good MS is kept in the dark about anything beyond its own functionality

Considerations of a Hybrid Design

* a monolith will only lose the benefit of component deployability
* monoliths can be composed of (relatively) loosely coupled modules/components
* monoliths also have many benfits that ms provide
* code can be quick to modify in monoliths
* monoliths can be fault tolerant

Remote Access Design Patterns

* composition -
* decomposition- by business capability or by a subdomain
* database per service pattern
* single service per host
* multiple services per host
* externalized configuration pattern (external API)

Microservices Observability Patterns

* log aggregation – taking the events and putting them into a stash and aggregate by specific functionality like transaction committing and rollback events
* application metrics – healthy of the overally system and what is running
* audit logging (observe events and debug stops) describes what it is doing
* distributed tracing – look at components running remotely
* exception tracking – emitting events in exception cases
* health check API – looking at any event logs and aggregating them
* log deployments and changes – keeps track of deployments/changes

UI Patterns

* Server-side page : Fragmentation Composition
* Client-side : UI Composition (converse of server-side) both in the middle

Point-to-Point Protocol (PPP) –data-link protocol that operates over a point-to-point network link connecting two communicating peers at the link level

* common for dial-up Internet access
* used in synchronous and asynch connections
* can dynamically configure and test remote network connections
* often used by clients to connect to networks and the Internet
* also provides encryption for passwords
* secure authentication of remote users

PPPoE (over Ethernet) and PPPoA (over Asynchronous transfer mode)

* more recent PPP implementations
* used by many DSL connections
* PPPoA is slightly faster than PPPoE
* using PPPoA does avoid the Maximum Transfer Unit(MTO) issue
* MTU for oA is 1500 byes, MTU for oE is 1492 byes

Point-to-Point Tunneling (PPTP)

* a Microsoft VPN layer 2 protocol
* increases security of PPP by providing tunneling and data encryption for PPP packets
* uses the same authentication types as PPP
* common VPN method among older Windows clients such as LINQ query provider
* has serious vulnerabilities and is no longer recommended by Microsoft
* easy to hack, should not be considered for use

Layer Two Tunneling Protocol (L2TP)

* Internet standard protocol combo of PPTP and Layer 2 Forwarding (L2F) from Cisco
* enables the tunneling of PPP sessions
* tunneling across a variety of network protocols such as IP, Frame Relay, and Asynchronous Transfer Mode (ATM)
* used to support virtual private networks
* does not provide any encryption
* saves the dial-up cost and overhead for any user connected remotely

Interprocess Communication (IPC) Mechanism – where multiple threads in many or one processes need to exchange data with the other

1. first dimension

* one-to-one – each client request is processed by exactly one service instance
* one-to-many – each request is processed by multiple service instance

1. second dimension

* synchronous – the client expects a timely response from the service and may even block while it waits
* asynchronous – the client will not block while waiting for a response, and the response, if any, isn’t necessarily sent right away

One-to-One Interactions

* request/response – a client makes a request to a service then waits for a response. client then expects the response to arrive. in a thread-based application, the thread that makes the rquest may even block while waiting
* notification (one way request) – client sends a request to a service, but no reply is expected
* request/asynch response – client sends a request to a service that replies asynchronous. Client does not block while waiting and is designed with the assumption that the response might not arrive soon

One-to-Many Interactions

* publish/subscribe- Client publishes a notification message, which is consumed by zero or more interested services
* publish/async responses – Client publishes a request message, then waits for responses form interested services

IPC Technologies-

* HTTP-based REST or Thrift (scalable model for developing distributed systems)
* AMQP(reliability and interoperability) or STOMP (text-based like HTTP)
* JSON or XML (for transmitting plain text)
* Apache Avro (reliable data serialization system- fast binary data format) or Protocol Buffers

Communication Assumptions and Practices

* Microservices must communicate using an IPC
* consider how the services interact
* consider how to specify the API for each service
* consider how to evolve the APIs
* consider how to handle partial failure

Converting Monolith to MS- assign API to each individual MS

Kontena- dev-friendly container and MS platform

* all you need to run containers in production
* powerful CLI tools for minotring and management
* deploy on any infrastructure
* may be used together with Kontena cloud
* 100% open source under Apache 2.0 license

Apache Kafka

* Kafka is an open source distributed message broker
* maintained by Apache software foundation (made by LinkedIn)
* fundamental job is to write messages to a log on disk
* queues in Kafka are called “topics”

Docker

* container platform to build, secure, and manage applications
* open sourced
* multi-architecture operations at scale
* built with open source technology
* easily integrates into existing environments
* conducive to rapidly changing environments

Yipee

* helps DevOps teams create, collaborate, and orchestrate
* flexibility in development and deployment
* orchestration flexibility
* visualize apps in the Yipee.io modeler
* provides a visual representation of networks and storage
* collaborate and share apps

Prometheus

* real-time monitoring system with alerts and flexible query language
* Prometheus was designed for a large number of nodes
* uses a convenient, graphical interface
* supports time-based tracking
* query language makes it easy to gather germane monitoring information quickly

logging is critical to MS

log4J services (download library)

ILog logger = LogManager.GetLogger(typeof(LogTest)); //create logger

BasicConfiguration.Configure() //main method

logger.Debug(“debug log”);

logger.Info()

logger.Warn()

logger.Error()

logger.Fatal();

Refactoring Patterns

Adapter Microservices Pattern

* adapts, as needed, between a business-oriented API build using RESTful or lightweight messaging techniques
* adapting is necessary when a development team does not have decentralized control over an app’s data source
* an adapter microservice wraps and translates existing services into an entity-based REST interface
* this type of microservice treats each new entity interface as a MS and builds, manages, and scales it independently
* moving from a functional (verb) approach to an entity (noun) approach
* functions exposed in a SOAP endpoint correspond one-to-one to CRUD operations on a single business object type
* functions map easily to a REST interface
* operations send corresponding SOAP messages to existing SOAP endpoint
* operations then translate the XML data types from the corresponding SOAP operations to JSON data types for the new REST interface

SOAP

1. XML-based message protocol
2. uses WSDL for communication between consumer and provider
3. invokes services by calling RPC method
4. does not return human-readable result
5. transfer is over HTTP (and uses other protocols such as SMPT, FTP, etc)
6. Javascript can call SOAP, but difficult to implement
7. performance is not as good as REST

REST

1. architectural style protocol
2. uses XML or JSON to send and receive data
3. calls services via URL path
4. result is readable as plain XML or JSON
5. transfer is over HTTP only
6. easy to call from Javascript
7. performance is better, less CPU intensive, leaner code

Strangler Application Pattern

* helps manage the refactoring of a monolithic application in stages
* central to transformation-existing applications into microservices
* uses structure of a web application
* splits up an app into different functional domains
* replaces those domains with a new ms-based implementation one domain at a time
* newly refactored app replaces original over time until monolith can be shut off
* Steps- Transform, Coexist, Eliminate

Strangler Pattern Implementation

* analyze the relationships between the screens in your existing UI
* look for the aspects of your UI that correspond to MSs
* size your chunk based on assumption that UI changes must be self-consistent
* choose whether to release an entire chunk at a time or a series of slivers

Mainstream Definition for Microservices

* apps are distributed
* apps are separately deployed
* maintains the concept of services components (may have multiple modules in one service comp)
* bounded context (for modules, not data)

Hybrid Microservices

* Service Granularity (loosely coupled monolith)
* Shared Database
* Integration Hub/Message Bus

Alternatives to Microservices

* monolithic app that has loosely coupled, modular components
* alternatives are the result of long-known, best architectural practices
* refactoring designs are very local to your app
* beg, borrow, steal the best designs that work for you

Proxy Microservice Design Pattern

Load Balancer 🡪 proxy 🡪 Service A, B, or C 🡪 Cache 🡪 DB

Chained Microservice Design Pattern

Load Balancer 🡪 Service A 🡪 Cache or Serv B 🡪 Serv C or Cache 🡪 DB

Cloud Infrastructure

Amazon Web Services, Microsoft Azure, OpenStack, Google Cloud Platform, Rackspace, etc

piecing together pieces of software from service-based software

Enterprise Service Bus determines where the request is along the chain of custody

service discovery architecture, looks into services to find correct path

examples of technology stacks for a MS design patter-

Spring Boot, Spring Cloud Stream, Spring Data Rest, Spring Data JPA, Spring MongoDB, Spring Cloud, Netflix Eureka, Netflix Ribbon, Apache Kafka

Needed Technical Knowledge

* Docker
* REST/HTTP/RAML/Swagger
* Asynchronous and event-driven programming
* Kubernetes for services orchestration
* gRPC for high-speed internal communication
* GraphQL for data orchestration
* Java and C#
* Spring Boot and Spring Cloud

Microservices Technical Requirements

* services should be loosely coupled
* each service is independent during the development and deployment phases
* deployment should be flexible regardless of the environments
* *data consistency should be achieved by asynchronous, nonblocking operations*
* or synchronous blocking if necessary

during functional requirements, identify technology (in design phase) for each part of development cycle

Sunsetting MSs - retirement process:

demand driven- what is calling our service? replace with another MS

supply driven – are other MS doing same thing? eliminate worse one

have to understand app or service in terms of data flows and events

data event interception is key (incrementally)

Spring Boot can be used to quickly build MSs

Spring Starter Project with REST(to use HTTP protocols) and Web, Web Services

Dependencies to add

* REST (and HAL) -to be accessible to HTTP protocols
  + Rest Repositories
  + Rest Repositories HAL Browser
  + Jersey
  + HATEOAS
* to connect to Web
  + Web
  + Web Services
* Discovery Services- Eureka Server
* Config Server – provide centralized ms management (ensures artifacts are accessible on GIT or any SVN backend)
* Eureka Discovery – helps client discover the services
* Pivotal Cloud Foundry – Config Client, Service Registry, Circuit Breaker
* Messaging – Cloud Bus (with ActiveMQ) , Cloud Stream
* Cloud AWS Capabilities – AWS Core, AWS JDBC, AWS Messaging (can be used to interact with various components that are available on Amazon Web Services)
* Cloud Providers
* Infrastructure Management

Database Architecture in Microservices

* services need to persists data
* database-per-service pattern used
* business transaction must query the data distributed over multiple services
* services must join multiple servers data
* database must sometimes be replicated and shared to scale
* different services may have different data storage needs
* must be able to be read quickly

Solutions

* MSs data persistence made private to services
* accessible via API
* service database included in service implementation
* provision database server for each service (deployed with service)
  + private-table-per-service
  + schema-per-service (increased sharing)
  + database-server-per-service

Implementing Data Server per Service

* Loosely coupled services
* services use best-suited db type
* difficulties implementing business transactions spanning multiple services
* difficult to implement data joining on multiple databases
* solution: command query responsibility segregation and API composition

ACID principles (meant for Monolith) provided by SQL

* Atomicity
* Consistency
* Isolation
* Durability
* Result- Simple database access approach
* difficult to implement with MS as services are isolated and there can be multiple servers in multiple systems with their own dbs and tables

CAP theorem

* Consistency
* Availability
* Partition
* combination of SQL and NoSQL
* complex database access approach
* event-driven data management
* polyglot persistence
  + - Application Servers
* MongoDB Key/Value RDBMS Graph

Spring Cloud architectural components

Principles driving the construction of ms app framework

* decompose systems into numerous distributed processes
* change configuration without redeploying all processes
* locate processes that our process needs to collaborate with
* choose a process instance and communicate with it

Spring Cloud Framework Components

* Spring Cloud Config server – manages configuration and communication with the SVN in order to manage versions
* Cloud bus – messaging activities
* Netflix – service discovery
* Circuit breakers – ensure our MS are resilient and do not face issues or problems
* Zuul – gateway service with dynamic routing, monitoring security, and more

Capability Spring Cloud with Kubernetes

|  |  |
| --- | --- |
| autoscaling and self-healing | Pod/Cluster Autoscaler, HealthIndicator, Scheduler |
| resilience and fault tolerance | HealthyIndicator, Hystrix, HealthCheck, ProcessCheck |
| distributed tracing | Zipkin |
| centralized metrics | Heapster, Prometheus, Grafana |
| centralized logging | EFK |
| job management | Spring Batch, Scheduled Job, (Grafana?) |
| load balance | Ribbon, Service |
| service discovery | Service, Eureka Service |
| configuration management | Externalized Configurations, ConfigMap, Secret |
| service logic | Apache Camel, Spring Framework |
| application packaging | Spring Boot Maven Plugin, (Spring Batch) |
| deployment and scheduling | Deployment Strategy, A/b, Canary, Scheduler Strategy |
| process isolation | Docker, Pods |
| environment management | Namespaces, Authorizations |
| resource management | CPU and Memory Limits, Namespace Resource Quotas |
| IaaS | GCE, Azure, CenturyLink, VMware, OpenStack |
| DevOps experience | Self-service, Multienvironment Capabilities |

Data Architecture Components

* NoSQL, RDBMS
* Host components- Jobs, Sink, Source, and YARN – responsible for managing jobs
* Cloud – cloud services can provide data infrastructure and containers
* Data Flow – info in how data flows

|  |  |  |
| --- | --- | --- |
| HOST | YARN | RDBMS |
| Job –SPI Implementation-> | Lattice Cluster | NoSQL |
| Sink | Cloud Foundry | Cloud Native |
| Process –Spring Cloud Data Flow-> |  | Other apps |
| Source |  |  |

Microservices and the REST Architecture

REST

* relies on stateless protocol
* simplified architectural style that works on principles of HTTP
* uniform interface
* resource interaction using HTTP verbs (get, put, post, delete)
* no overhead
* can use multiple languages

Constraints on REST

* client server –separates concerns for separate evolution of client and server apps
* stateless – facilitates stateless communication
* cache –enables message response and data to be labeled as cacheable (can use cache on client or server), reusable
* layered system – can use middleware or API, facilitates information restraint simplifying distributed architecture (which helps different layers to evolve separately)
* on-demand code –optional restraint

HTTP Request Structure

* GET /index.html HTTP:1.1HOST:example.com (resource, protocol, and host)
* headers have URI and version, body contains payload
* HTTP response- structure will contain the status or version code. there will be data, server, content type and length of content being retrieved
* then connection closes

Principles of RESTful Service

* POST – representation to be accepted for processing
* GET – representation of current state
* PUT – proposed modified state representation
* HEAD- metadata on the representation of current state

Microservices Access Using API Gateway and REST

Benefits

* client is independent of MS architecture
* provides API for each client
* reduces number of request/roundtrips
* moves logic for calling multiple services to simplify the client

Drawbacks

* more complex
* impacts performance due to additional hop through API gateway

http – port, context, and data

Monolith Problems

* continuous delivery difficulty
* team and project management difficulty
* high cost for scalability and performance
* minimal technology diversity
* less portable

Forces Driving Microservice Adoption

* Need new team members to become productive quickly
* Easy to understand and modify apps
* Reduces coupling and dependencies
* Continuous deployment/delivery
* Run multiple app copies on multiple machines for scalability and availability
* Take advantage of emerging technologies such as frameworks, programming languages, etc

Microservice Benefits

* Are small and easier to understand
* Faster IDE
* Apps start faster which speeds up deployments
* Easier-to-scale development
* Improved fault isolation (tolerance)
* Independent deployment
* No long-term commitment to technology stack

Microservice Challenges

* Complex distributed system
* IDEs do not support distributed applications development
* Testing complexity (especially for integration)
* Implementing an interservice communication mechanism (transaction management especially)
* Difficulty in implementing the use cases of multiple services
* Proper team coordination required to implement the use cases of multiple services

Decomposition of applications into smaller independent ms

Eureka Discovery- makes app discoverable

CAP

Consistency 🡨NoSQL🡪Availability🡨🡪Tolerance to Network Partition 🡨RDBMS🡪 Consistency

All 3 elements are not available

Data Consistency in Microservices

* Stronger consistency is hard in distributed services
* Even harder with microservices as they own their data
* Strong consistency only inside a microservice
* Dependency on the data scheme
* Dependency on the actual collected data
* Eventual data consistency because MS A must publish change to MS B to reflect A’s change

Data Consistency Types

* Strong Consistency

Subsequent accesses to a distributed system always return the updated value

* Weak Consistency
  + Used in distributed computing where subsequent accesses do not always return an updated value
  + Inconsistent responses
* Eventual Consistency
  + Special type of weak consistency method
  + Guarantees last update value return if no new updates are made to a given data item

Data Consistency- CAP

* CA – Consistent and available
* AP – Always available, but may be inconsistent
* CP – Always consistent, but may not be available
* Consistency : every read gets us the most recent write
* Availability : every node (if not failed) always executes queries
* Partition Tolerance : when connections between nodes are down, the other two (A and C) promises are kept

Query Pattern- command query pattern based on responsibility segregation

Command side- takes responsible of handling function like CRUD and Request, emits events when there are changes in the data

Query side – takes the responsibility for executing them against one or more materialized view that can be created by subscribing to the data stream or the events when data changes

CQRS – pattern will help us identify the event-source architecture

public String getAll(){  
 //implement mongodb query  
 //query  
 return "ALL";  
}  
public void createMessage(){  
 //statement to create  
 //command  
}

Business Capabilities -

* Services that are required in order to have proper understanding about businesses
* Typically related to the object of an organization in terms of business processes, structure, areas of operation, and are typically bound with context that are best identified by agile process

Organizational structure where we have different groups with different business capabilities

High-level domain model

One service for each business capability

Domain Decomposition-

Forces driving service decomposition

* Need for stable architecture
* Cohesive services
* Services conform common principles
* Testable services

Cross cutting/ MS chassis pattern

Spring Boot Dependency Injection

Externalized Configuration (keep sensitive info separate) – use application.properties file

API Gateway- granularity and network performance (can use Zuul, eureka discovery, and eureka server and web)

Cross-cutting concerns – metrics, service registration, discovery etc

Microservce chassis pattern evolves with the fundamental objective of reducing the time and complexity of microservices development

Spring Cloud and Spring Boot (dependency injection)

Externalize Components

Eternalized configuration patterns helps us identify sensitive information related to various components like database, server location, discovery, service location, etc which helps us configure and externalize them from the code

And is automatically refreshed for any change to the configuration

//@EnableEurekaServer  
// then application.properties file - change port to differnet spot to externalize config

Multiple services per host

CQRS and REST drive queries in microservice

Microservices Patterns

Systematic approach, Solution to known problems, Standardized solutions, Product maturity

* Core patterns
* Decomposition patterns – identify NFRs
* Cross-cutting patterns – facilitate robust and reuseability
* Deployment patterns
* Communication style patterns
* Service discovery patterns
* Reliability patterns
* Observability patterns
* UI patterns

Service Integration Contract Implementation

Test suite that simulates the environment

Spring Cloud Contract

* Open source reliable testing approach
* Solution for consumer driven contract approach
* Components of Spring Cloud Contract
  + Cloud Contract Verifier
  + Cloud Contract Stub Runner
  + Cloud Contract WireMock
* Alternate testing approach using Junit or Spock

Circuit Breaker Pattern

* Prevents service or network failure from cascading to other services
* Use Hystrix to implement circuit pattern

Token Pattern

* To facilitate authentication and authorization mechanisms
* API gateway
  + Single entry point for clients
  + Authenticates and validates requests
* JSON Web Token Libraries
  + Provides requester identity securely
  + Can be downloaded from jwt.io

Observability Patterns

* track what’s going on internally and understand the impact (observability)
* Log Aggregation (aggregates from various MSs)
  + Understand behavior of an app and troubleshoot problems
  + Use a centralized logging service that aggregates logs from each service instance
  + Users can search and analyze the logs
  + Users can configure alerts that are triggered when certain messages appear in the logs
* Application metrics
  + Have measurable or measured behavior that is desireable
  + Push/pull model
  + IMPORTANT METRICS
  + Application instances- indicate how many instances are required to properly run app
  + Error rates
  + Average response time – should check with this after any changes
  + Retrace satisfaction- apdech score
  + Request rate – to decide number of threads
  + Application availability – info on how available apps are
  + Garbage collection – internal feature
* Audit logging
  + Record that provides info about what has been done and has happened in lifespan of a particular server, service, or app
  + Gives info as to how cleanly our system has been behaving (resource crunches and usages for scalability)
* Distributed tracing
* Exception tracking
* Health check api
* Log deployments and changes
* Cloud watch is a part of Amazon Web Services

Logging

* import org.slf4j.Logger;
* private static final Logger log = LoggerFactory.getLogger(LoggingApplication.class)
* log.info(“application … “);

Distributed Tracing

* For identifying appropriate application behavior
* Identify issues to troubleshoot
* Assign request ID for external requests
* Include request ID in log messages
* Uses Sleuth(supports d t) in Spring cloud
* SPRING\_SLEUTH\_ENABLED: “true” (app.properties.yml)

Exception Tracking

* Service instances throw exception
* Exception contains- error messages and stack trace
* Centralized exception traffic tracking aggregates exceptions
* Ensures clean codes
* extends Exception {
* public void messageforIOException(){

Health Check API

* provision monitoring system that can generate right alerts and invoke load balancer
* Health endpoint
* Alert system
* Triggers service health
* Checks the status of- connection, infrastructure, disk space
* HealthIndicatorAutoConfiguration health = new HealthIndicatorAutoConfiguration
* New Health.Builder(new Status=”45567”)

Log Deployments

* Provides rich source of info to help decide on action
* Generates info about what it is doing to a log file in a standardized format
* Log file contains- Errors, Warnings, Information and Debug Messages
* Logging solution must have minimal runtime overhead

Minimizing Runtime Overhead

* Instrument a service to gather statistics of individual operations
* Aggregate metrics are centralized metrics service
* Models for aggregating
  + Metrics: Push/Pull
  + Libraries: JML, Yammer
  + Services: Cloud Watch
  + Visualization tool: Kibana

Logging Best Practices

* Have instance identifier on log entry
* Have field with UTC time to order message
* Generate a unique identifier for each transaction
* Aggregate log entries

Features of log and analytics

* Logs must be searchable
* Analytics on log helps identify problems
* Preunderstanding of error helps in better resolution
* ELK stack (Elstaticsearch, Logstash, and Kibana) helps in analytics of logs

Netflix

* Cloud-based ms architecture
* Moved from noncustomer facing to customer facing which led to MS adaption
* For availability, scale, and speed
* Broke their monolith into authorization, products/services, ordering, messaging and searching
* Spring Boot projects with Zuul API gateway, eureka discovery server, mongoDB
* Comes in through Edge Server (Zuul) with a API gateway and Ribbon load balancer before passing on to MSs with Eureka Service Registry and Service Discovery
* Hystrix (for 100% availability) a latency enforced tolerance library that controls interactions between MSs and checks health
* Ribbon – helps latency by reducing number of hops needed for services to communicate
* Discovery client will help services find other services through a registry
* API gateway can handle millions of requests and forward them
* Zuul – dynamic routing, stress testing, load shedding, static response handling authentication, security, insights, monitoring. Main function is to act as Gateway into API

Netflix Nebula –automated deployment

* Java based deployment and packaging app for testing and releasing MS
* Developed and deployed independently
* No need for intensive coordination
* Each service has own product and own life cycle
* Automated deployment- parallel or rolling restart (rollback)
* Needs Gradle to work
* Immutable server pattern

Spinnaker (akin to Asgard)

* supports releasing of software changes with high turnover
* open-source multi-cloud platform
* configurable pipelines, enhanced UI, robust API, flexible, resilient

Eureka – service discovery

* acts as a registry that coordinates other services in the system and knows which apps are reachable, like a db of operational and functional services
* Advantages of a registry
  + ensures that services are always reachable
  + handles self-registration
  + handles third-party registration
  + discovery is possible via client side and server side
* REST API in AWS Cloud
* helps Spinnaker with deployment
* Load Balancer
* Eureka client
* is a spring-boot java app

Ribbon – load balancer

* part of internal framework that handles communication between services
* prevents overloading of any single service by diverting traffic
* Availability and uptime- failure, load balancer (server and client), Eureka
* Features
  + Simple
  + Zone aware (helps with latency by connecting client to physically near server)
  + Resilient –reacts immediately to zone outage

NIWS – Netflix Internal Web Services

* handles the messaging part of communication
* provides serialization and deserialization data between services
* java client-server network

Archaius

* handles properties and environment variables
* .properties file, configuration management, deployment, context
* helps with updates and downtime, dynamic instances, runtime environment, and context
* gives true dynamic control, connection/time-out requests, logs, infrastructural components, and fine control

Containers (Docker)

* supports virtualization
* another process
* run several images
* Titus (type of Docker container) cloud, batch jobs, service jobs, Apache Mesos
* heyw

Zuul

* edge service in the cloud
* handles requests and forwards the request to a particular service
* use for dynamic routing, stress testing, load shedding, static response handling, authentication, and security
* acts as gateway (entry point) to the network of internal services
* can handle more than 50K requests per second
* complexity, volume, new features, new regions
* basic functionality run by filters (type, execution sequence, criteria, action)
* filter types- pre(authentication), routing, post (add HTTP headers to response and metric logging), error

Zuul 2 (now available)

* asynchronous
* better performance

Conductor

* handles workflows and resulting complexity
* workflow is made up of tasks (methods)
* each task has several processes which ensure task is completed on time
* processes handle things like communication with other processes, persistence, logging
* Features-
  + complex processes
  + JSON based
  + visibility and traceability
  + full control
  + reuse
  + user interface
  + synchronous processing
  + scalability

Typical Workflow

* request content -> inspect content -> decision -> encode -> publish

Event Sourcing

* what drives the application in the cloud
* event-driven
* immutable one-way messages
* metadata
* events that occur on a db are stored
* series of events are captured and recorded
* this allows- complete rebuild and helps with auditing

Event Sourcing Patter

* aggregate
* command handler – represents a request from client to change aggregate
* event handler – immutable representation of state of change
* repository
* event store - is aggregate of these events and then stored in db

Eureka handlers all incoming traffic from users and passes these requests onto other embedded MS

Spinnaker is pipeline delivery system for use by devs. it is how devs from development to product in a way that is seamless to the users

Ribbon is internal load balancer, it ensures that requests are appropriately sent to services that can handle that request and not create a bottleneck

Archaius gives ability to change properties of running apps on the fly

Design Thinking

Observer and Analyze

* gather UI requirements
* develop persona
* contextual interview – problems need to be solved
* evaluate GUI
* prototype

Envision and Design

* information architecture
* model and design interaction
* design visualization
* data visualization
* feasibility and usability tests
* UT specification

Evaluate and Refine

* feasibility and usability tests
* development support
* revise UI specification
* create visual asset
* review post development UI

Prototype

* implementation of conceptual and/or theoretical ideas
* prototypes explore the impact of ideas before an eventual implementation
* experimental models are built to test and validate ideas and to validate design assumptions and other aspects of ideas
* experimental models are built quickly at low cost and effort
* they are needed to understand the process and collect feedback across all stages of design

OOAD Principles behind Design Patterns

* prefer composition over inheritance
  + more flexible
  + less coupling
* program to interface rather than implementation
  + easier to test
  + change and replace implementation as needed
* UML artifacts used to depict design patterns
  + class diagram – name, attributes, behavior – defines participants
  + sequence diagram
  + associations
  + state machine

Benefit of SOLID principles

* principled dependencies management
* solid foundation for OOAD to build complicated design patterns
* results in following
  + robust
  + reusable
  + flexible
* solid base for designing and defining design patterns

how could you even make a big unwieldy interface when all the methods require implementation?

can you explain dependency inversion? I am having a hard time understanding it. Could you provide a concrete example?

There is training video on how to use Visual Studio (Microsoft, not Code), is that something to pay attention to? Is Visual Studio still widely used?