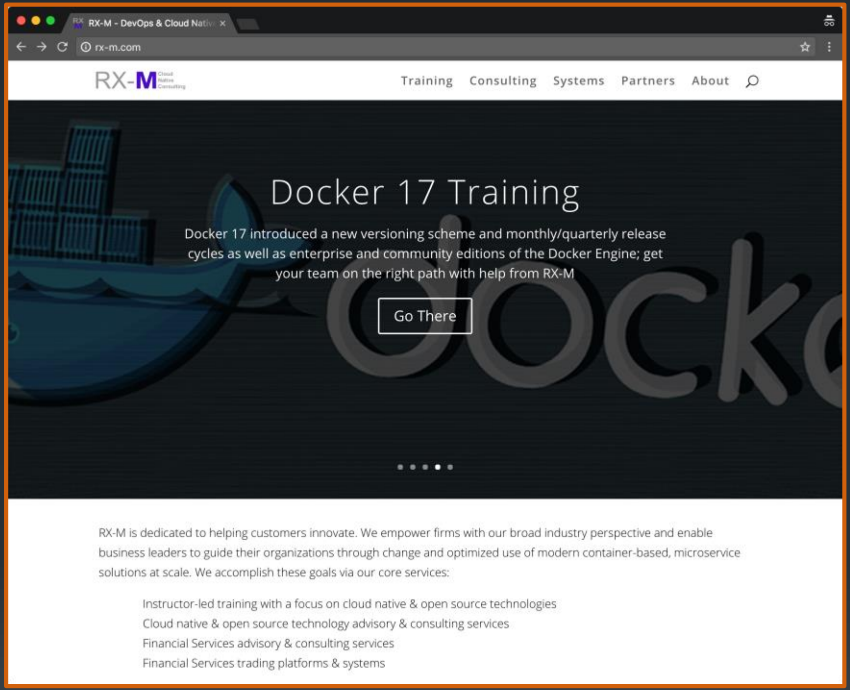
RX-M Cloud Native Consulting

Building Cloud Native Applications on Cloud Foundry

An in depth look at the microservices architecture pattern, containers and Cloud Foundry

RX-M Cloud Native Training

- Microservice Oriented
  - Microservices Foundation [3 Day]
  - Microservices on AWS [3 Day]
  - Microservices on Azure [3 Day]
  - Microservices on GCP [3 Day]
  - Microservices on Bluemix [3 Day]
  - Microservices on Oracle Cloud [3 Day]
  - Building Microservices with Go [3 Day]
  - Building Microservices with Apache Thrift [3 Day]
  - Building Microservices with gRPC [3 Day]
- Container Packaged
  - Docker Foundation [3 Day]
  - Docker Advanced [2 Day]
  - OCI [2 Day]
  - Container Technology [2 Day]
  - CRI-O [2 Day]
- Dynamically Managed
  - Docker Orch. (Compose/Swarm) [2 Day]
  - Kubernetes Foundation [2 Day]
  - Kubernetes Advanced [3 Day]
  - Kubernetes for Developers [3 Day]
  - Mesos Foundation [2 Day]
  - Nomad [2 Day]



RX-M Cloud Native Consulting

Overview

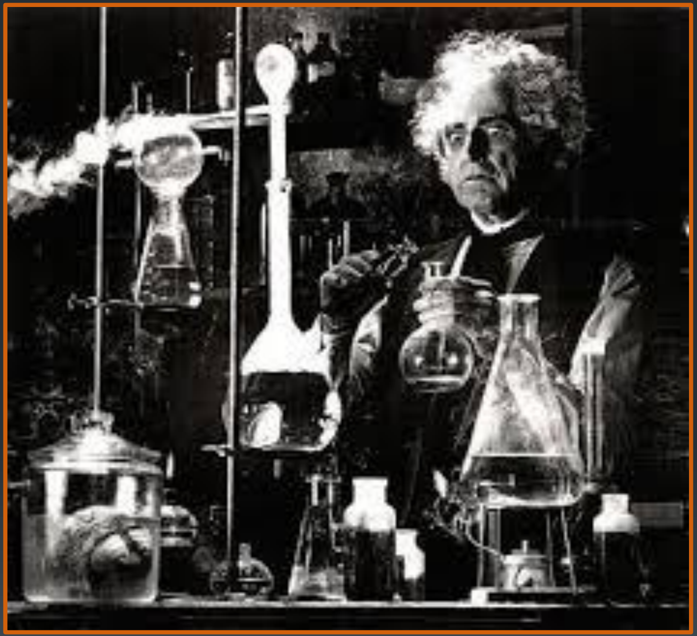
1. Microservice Overview
2. Microservice Communications I - Client/Server
3. Container Packaging
4. Microservice Communications II - Messaging
5. Cloud Native Transactions and Event Sourcing
6. Stateless Services and Polyglot Persistence
7. Microservice Orchestration
8. FaaS
9. API Gateways
10. Cloud Foundry Container Runtime
11. Pods
12. Cloud Foundry Application Runtime
13. Cloud Foundry Application Runtime Architecture

Administrative Info

- Format: Lecture/Labs/Discussion
- Schedule: 9:00AM – 5:00PM
  15 minute break, AM & PM
  1 hour lunch at noon
  Lab work after each module
- Location: Fire exits, Restrooms, Security, other matters
- Attendees: Name/Role/Experience/Goals for the Course

Lecture and Lab

- Our Goals in this class are two fold:
  1. Introduce concepts and ecosystems
     - Covering concepts and where things fit in the world is the primary purpose of the lecture/discussion sessions
     - The instructor will take you on a tour of the museum
       - Like a museum tour, you should interact with the instructor (tour guide), ask questions, discuss
       - Like a museum tour, you will not have time to read the slides during the tour, instead, the instructor will discuss and point out the highlights of the slides (exhibits) which will be waiting for you to read in depth later should you like to dig deeper
  2. Impart practical experience
     - This is the primary purpose of the labs
     - Classes rarely have time for complete real world projects so think of the labs as thought experiments
       - Like hands on exhibits at the museum



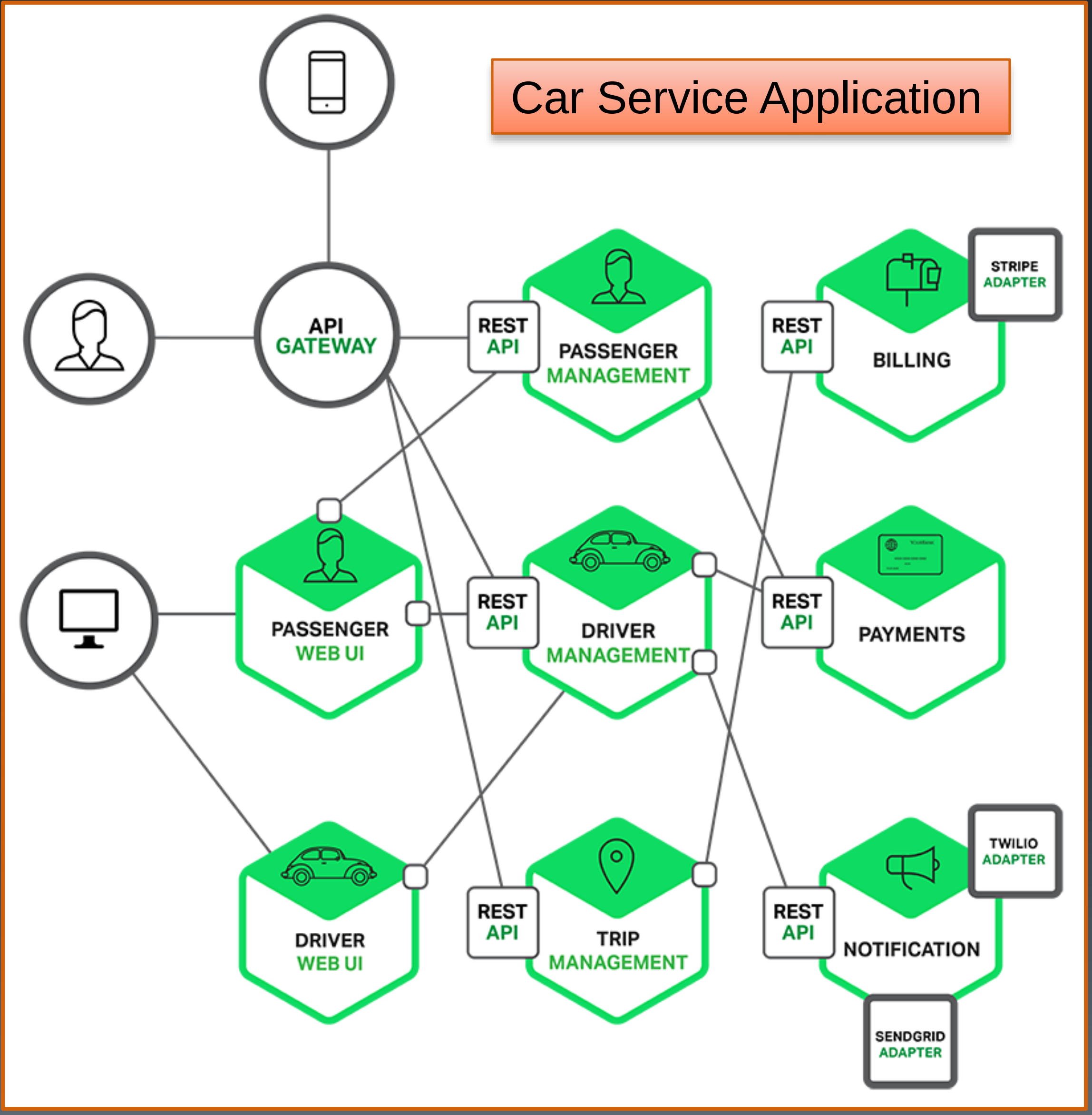
1: Microservice Overview

Objectives

- Define microservice
- Explain the microservice architecture (MSA)
- List the perceived benefits of microservices
- Explore the down sides of microservices
- Describe processes and approaches used successfully with microservices

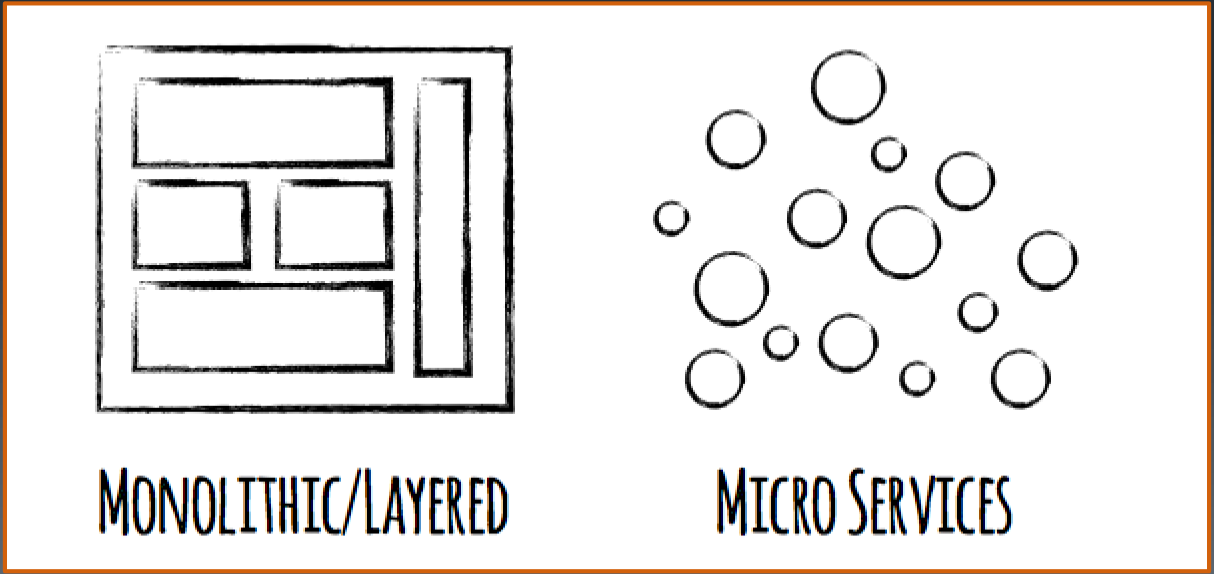
What is a microservice?

- Microservice
  - A fine grained atomically deployable service accessed via a platform agnostic network API
- Microservice Architecture (MSA)
  - An architectural pattern used to build distributed software systems
  - Processes communicate over networks
  - Composed of business aligned services
  - Supports design evolution and self organization
  - An approach favoring symmetry over hierarchy (peer to peer not layers)
  - An architectural approach that defines an application, not an enterprise
- A specialization of Service-Oriented Architectures (SOA)
  - MSA is the first realization of SOA designed to complement Agile and DevOps methodologies (both of which matured after SOA)



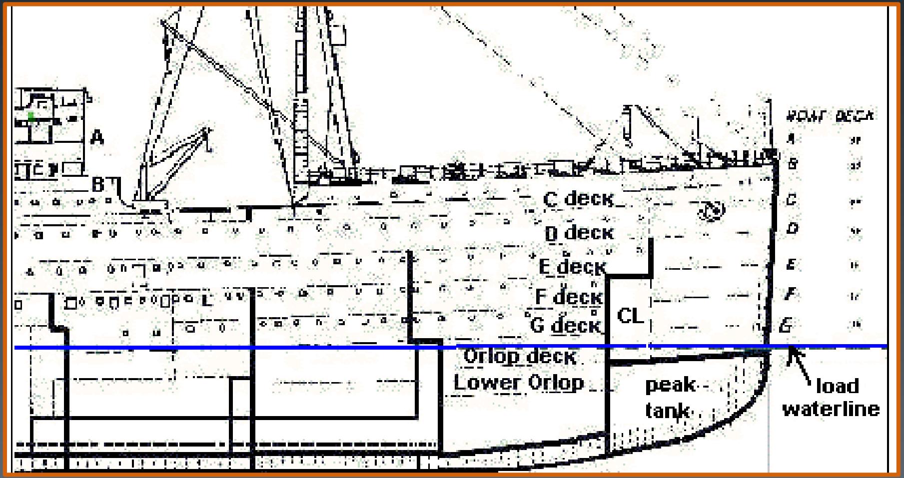
Microservice Attributes

- Small crisply defined services
- Decomposed by business/problem domain, not technology
- Well defined interfaces, completely abstract implementations
- Light-weight, technology-agnostic inter-service communications
- Polyglot friendly
- Changes are single service and incremental
- Services are atomically deployable and designed to be replaced
- Services are not changed they are replaced (and can be rolled back)
- Services are loosely coupled (discovering each other dynamically if need be)
- Single responsibility (high cohesion)
- Persistent state is encapsulated behind service interfaces and managed using cluster friendly network distributed schemes
- State is decomposed by service not centralized or shared



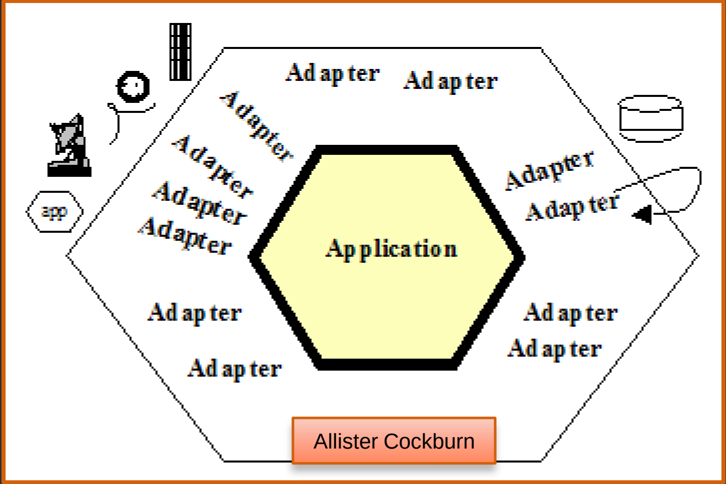
Microservices benefits

- Deliver software faster
  - A change to a microservice typically requires only the deployment of the microservice
  - Monolithic applications require large deployments for small changes
- Fine grained scaling
  - Microservices allow you to scale only the parts that need to be scaled
- Flexibility to embrace newer technologies
  - Risk is a key barrier to adopting new technologies
  - With a monolithic application a new programming language, database, or framework will impact a large amount of the system
  - With microservices this can be a small experiment and rolled back easily and without large cost if it fails
- Organizational Alignment
  - Each team can own one or more services, reducing coordination overhead
- Respond faster to change
  - Microservices are easier to rewrite wholesale, making it possible to try different ideas and track changes in business structure and patterns
- Composability
  - Microservices are atomic enough to be easily consumed by a range of clients for different purposes
- Resilience
  - When a single microservice fails only a small part of the application is inoperable
  - Enables microservice applications to operate more effectively in the face of partial failure
    - Resilience engineering bulkhead concept

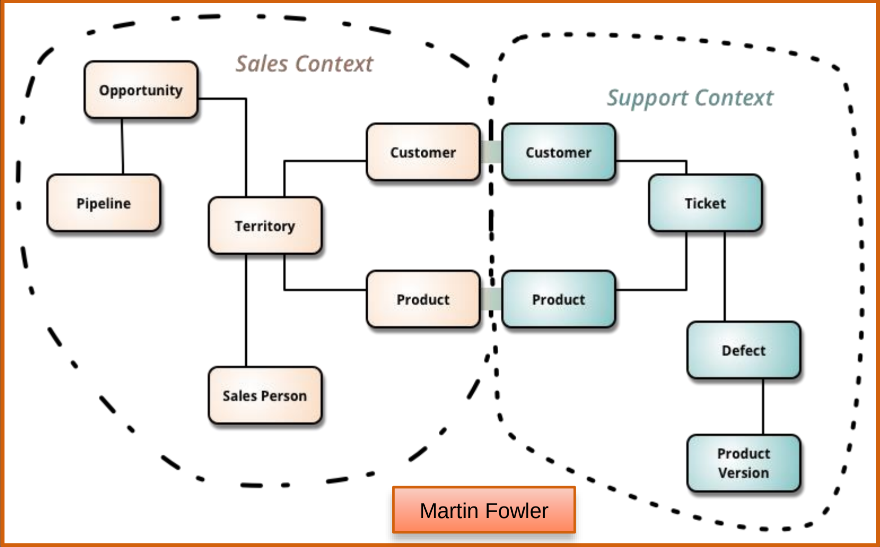


Architecture Evolution

- There is no perfect system architecture
  - Only those who have built a complete system can clearly reflect upon what the perfect solution for a given problem might be
- Microservice architectures allow the right architecture to emerge naturally over time
  - While controlling risk and cost
- Microservice based systems focus on business goals
  - Services align with the business and its needs
    - Bounded Context
  - Business changes are likely to have a one to one correspondence services which need to be changed in the solution
- Minimizing service scope limits the effect of bad design decisions
  - Old services with bad designs can be replaced wholesale, even using completely different languages, frameworks and state management solutions
- Microservice interfaces can evolve in backward compatible ways
  - REST, Thrift, gRPC/ProtoBuf all allow for backward compatible interface evolution
- Interfaces that are found to be unsound can be superseded progressively
- Microservices are typically designed without either a UI or a database
  - Allows them to be easily tested
  - Allows them to work when the database becomes unavailable or changes
  - Allows applications to collaborate in varying and flexible ways, independent of UI design and evolution



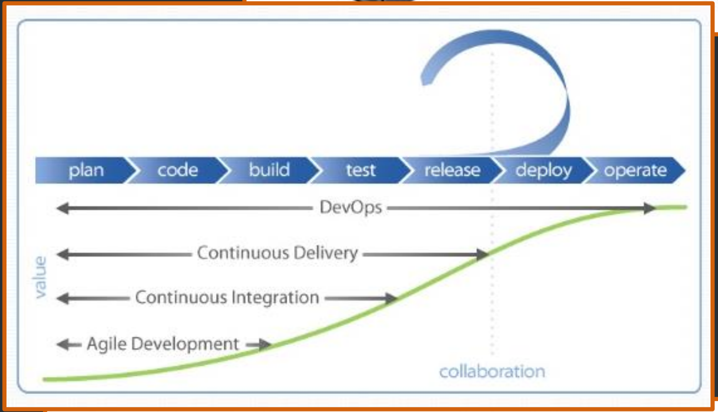
Allister Cockburn

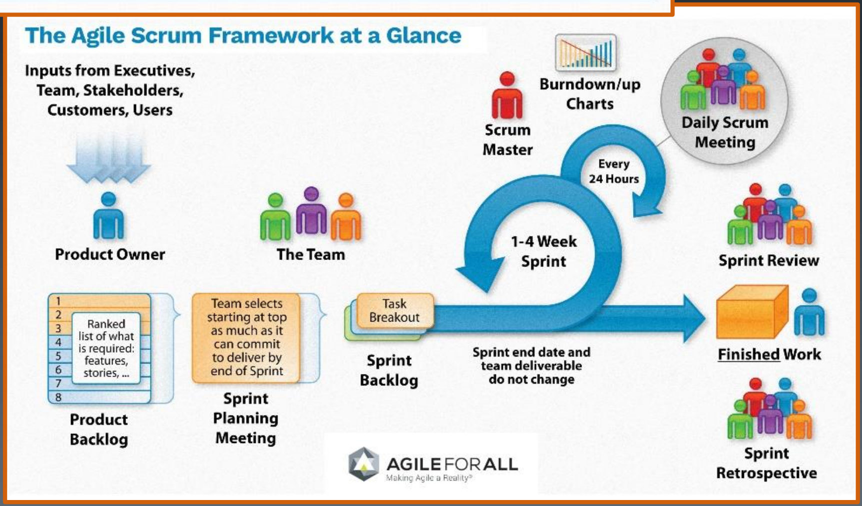


Martin Fowler

Microservice Precursors

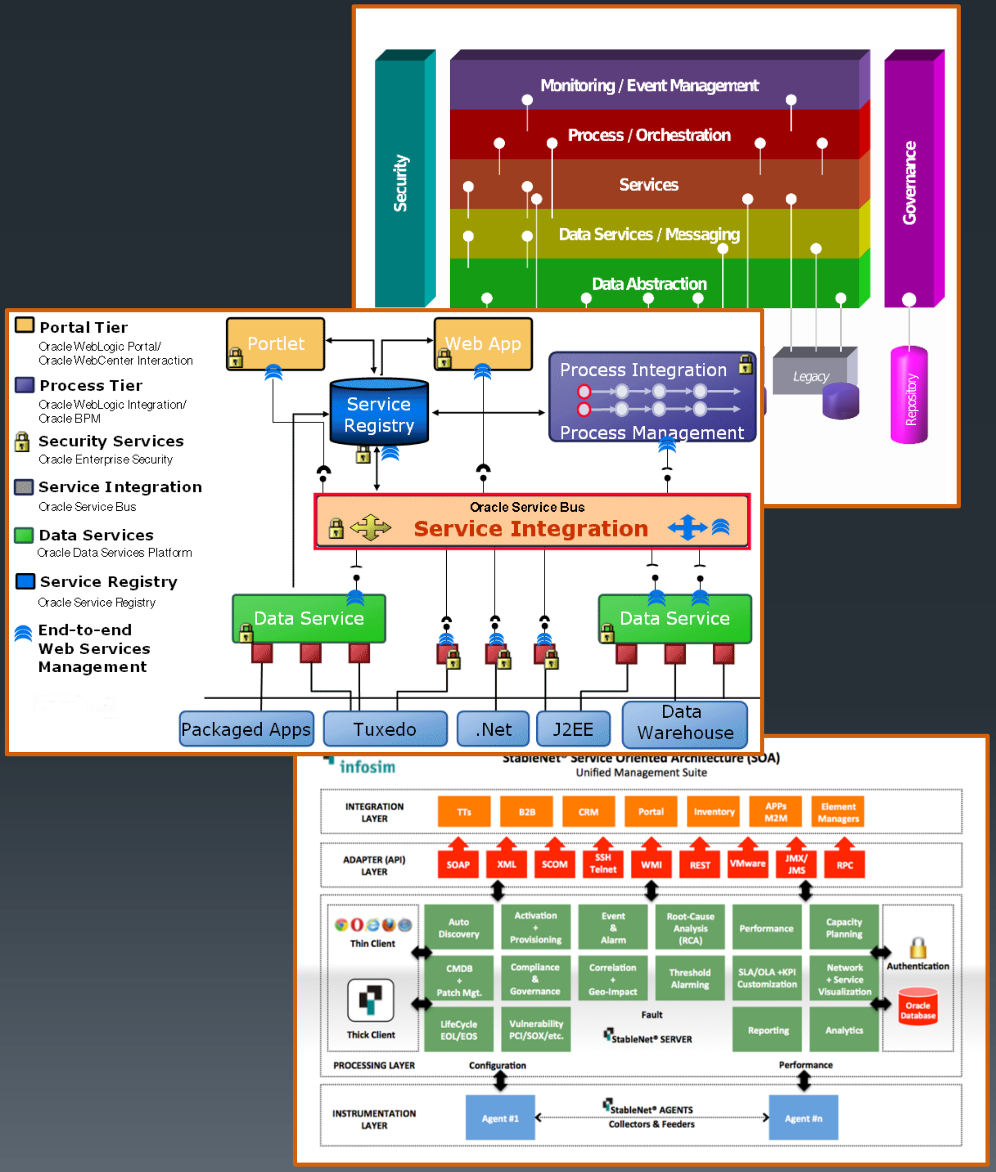
- Service Oriented Architecture (SOA)
  - Promoted the construction of applications from networked services using vender agnostic interfaces
- Domain-Driven Design and its precursors
  - Instilled the importance of representing the real world in code
- Hexagonal Architecture
  - Allister Cockburn argues for avoiding layered architectures where business logic can hide
- Agile
  - Describes the benefits of small teams owning the full lifecycle of their services
- Continuous Integration and Delivery
  - Demonstrated the benefits of moving software to production early and often, treating every check-in as a release candidate
- Virtualization
  - Enabled dynamic provisioning of systems
- DevOps
  - Infrastructure automation (Puppet, Chef, Ansible, Salt, etc.) and the unified views of development and operations give us ways to handle dynamic infrastructure and CI/CD at scale





SOA

- Service-Oriented Architecture (SOA)
  - An architectural style where services are provided through a communication protocol over a network
  - A fundamental principle is independence
    - of vendors
    - of products
    - of technologies
- A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently
  - e.g retrieving a credit card statement online
- The four properties of a service:
  1. Represents a business activity
  2. It is self-contained
  3. It is a black box for its consumers
  4. It may consist of other underlying services
- Different services can be used in conjunction to provide the functionality of a large software application
- The goal _was_ flexibility and independence
  - Many SOA solutions were heavy weight and scaled poorly
    - SOAP
      - Oasis offers >30 standards for SOAP
      - The W3C SOAP 1.2 standard is >100 pages long
    - Enterprise Service Buses
    - Layering
    - Etc.

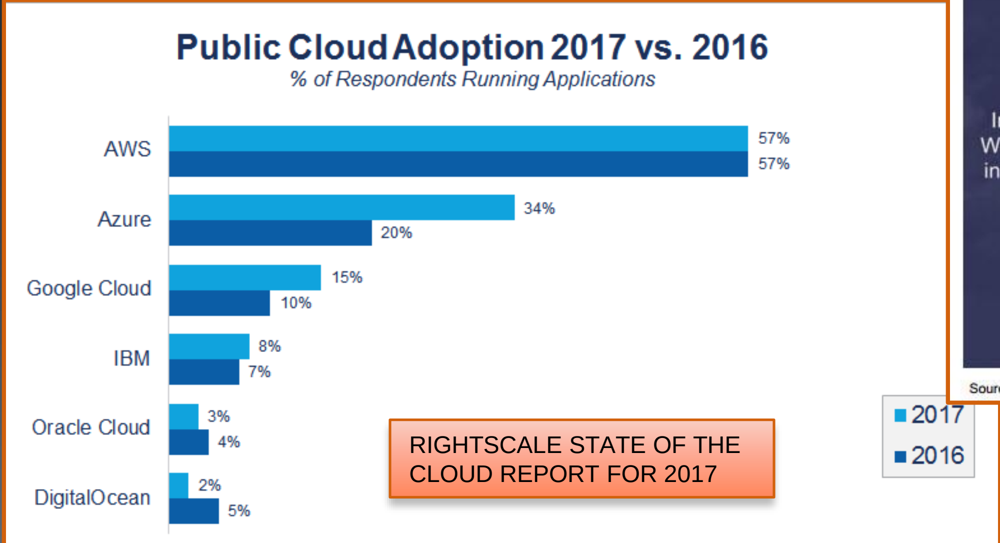


Growing dominance of the Cloud

- The 10/2015 Cisco GCI research report predicts that by 2019:
  - Global data center traffic will grow nearly 3-fold
  - Global data center traffic will reach 10.4 zettabytes per year
  - 83 percent of all data center traffic will come from the cloud
  - 4 out of 5 data center workloads will be processed in the cloud
- This mass migration to the cloud creates challenges for development teams:
  1. How to build the right software for the cloud
  2. … and therefore, how clouds work and how to effectively leverage them

Figure 9. SaaS Most Highly Deployed Global Cloud Service by 2018

Source: Cisco Global Cloud Index, 2013–2018



RIGHTSCALE STATE OF THE CLOUD REPORT FOR 2017

12 Factor Apps

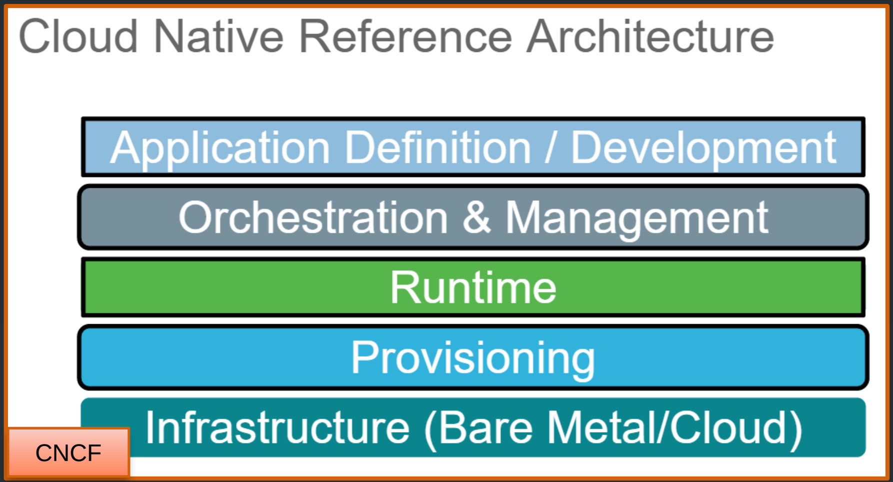
- Codebase
  - Each deployable app is tracked as one codebase in revision control (may have many deployed instances across multiple environments)
- Dependencies
  - An app explicitly declares and isolates dependencies via appropriate tooling (e.g., Maven, Bundler, NPM) rather than depending on implicit dependencies in its deployment environment
- Config
  - Configuration, or anything that is likely to differ between deployment environments (e.g., development, staging, production) is injected via operating system-level environment variables
- Backing services
  - Backing services, such as databases or message brokers, are treated as attached resources and consumed identically across all environments
- Build, release, run
  - The stages of building an artifact, combining that artifact with configuration, and starting one or more processes from that artifact/configuration combination, are strictly separated
- Processes
  - The app executes as one or more stateless processes (e.g., master/workers) that share nothing, state is externalized to backing services (cache, object store, etc.)
- Port binding
  - The app is self-contained and exports any/all services via port binding (including HTTP)
- Concurrency
  - Concurrency is usually accomplished by scaling out app processes horizontally (though processes may also multiplex work via internally managed threads if desired)
- Disposability
  - Robustness is maximized via processes that start up quickly and shut down gracefully, these aspects allow for rapid elastic scaling, deployment of changes, and recovery from crashes
- Dev/prod parity
  - Continuous delivery and deployment are enabled by keeping development, staging, and production environments as similar as possible
- Logs
  - Rather than managing logfiles, treat logs as event streams, allowing the execution environment to collect, aggregate, index, and analyze the events via centralized services
- Admin processes
  - Administrative or managements tasks, such as database migrations, are executed as one-off processes in environments identical to the app’s long-running processes

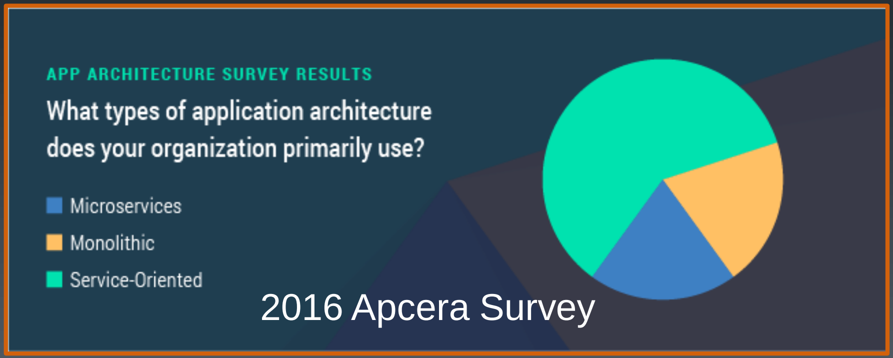
12 Factor Apps

- Codebase
  - Each deployable app is tracked as one codebase in revision control (may have many deployed instances across multiple environments)
  - Microservices: SAME
- Dependencies
  - An app explicitly declares and isolates dependencies via appropriate tooling (e.g., Maven, Bundler, NPM) rather than depending on implicit dependencies in its deployment environment
  - Microservices: DIFF   This conflates Build Operations with Distribution. In a PaaS you push code in a CaaS you push container images. Local activities use one build and PaaS activities use another. The image history should be immutable and no build should be required to recall any prior image for dev, testing or production deployment.
- Config
  - Configuration, or anything that is likely to differ between deployment environments (e.g., development, staging, production) is injected via operating system-level environment variables
  - Microservices: DIFF   Microservices should rely principally on dynamic discovery mechanisms (DNS, etc.) for environmental variations.
- Backing services
  - Backing services, such as databases or message brokers, are treated as attached resources and consumed identically across all environments
  - Microservices: SAME
- Build, release, run
  - The stages of building an artifact, combining that artifact with configuration, and starting one or more processes from that artifact/configuration combination, are strictly separated
  - Microservices: DIFF   Building and packaging operations should be atomic with microservices, an unpackaged service is not deployable/testable/etc.
- Processes
  - The app executes as one or more stateless processes (e.g., master/workers) that share nothing, state is externalized to backing services (cache, object store, etc.)
  - Microservices: SAME
- Port binding
  - The app is self-contained and exports any/all services via port binding (including HTTP)
  - Microservices: DIFF   Containerized microservices can listen on any port they desire and benefit from consistant port usage across replicas
- Concurrency
  - Concurrency is usually accomplished by scaling out app processes horizontally (though processes may also multiplex work via internally managed threads if desired)
  - Microservices: SAME
- Disposability
  - Robustness is maximized via processes that start up quickly and shut down gracefully, these aspects allow for rapid elastic scaling, deployment of changes, and recovery from crashes
  - Microservices: SAME
- Dev/prod parity
  - Continuous delivery and deployment are enabled by keeping development, staging, and production environments as similar as possible
  - Microservices: SAME
- Logs
  - Rather than managing logfiles, treat logs as event streams, allowing the execution environment to collect, aggregate, index, and analyze the events via centralized services
  - Microservices: SAME
- Admin processes
  - Administrative or managements tasks, such as database migrations, are executed as one-off processes in environments identical to the app’s long-running processes
  - Microservices: SAME (Blue/Green approach)

Next Generation Models

- Cloud Native Computing
  - A new computing paradigm optimized for modern distributed systems environments
  - Capable of scaling to tens of thousands of self healing multi-tenant nodes
- Cloud native systems have the following properties:
  - Container packaged
    - Running applications and processes in software containers as an isolated unit of application deployment, and as a mechanism to achieve high levels of resource isolation
    - Improves overall developer experience, fosters code and component reuse and simplify operations for cloud native applications
  - Dynamically managed
    - Actively scheduled and actively managed by a central orchestrating process
    - Radically improve machine efficiency and resource utilization while reducing the cost associated with maintenance and operations
  - Micro-services oriented
    - Loosely coupled with dependencies explicitly described (e.g. through service endpoints)
    - Significantly increase the overall agility and maintainability of applications





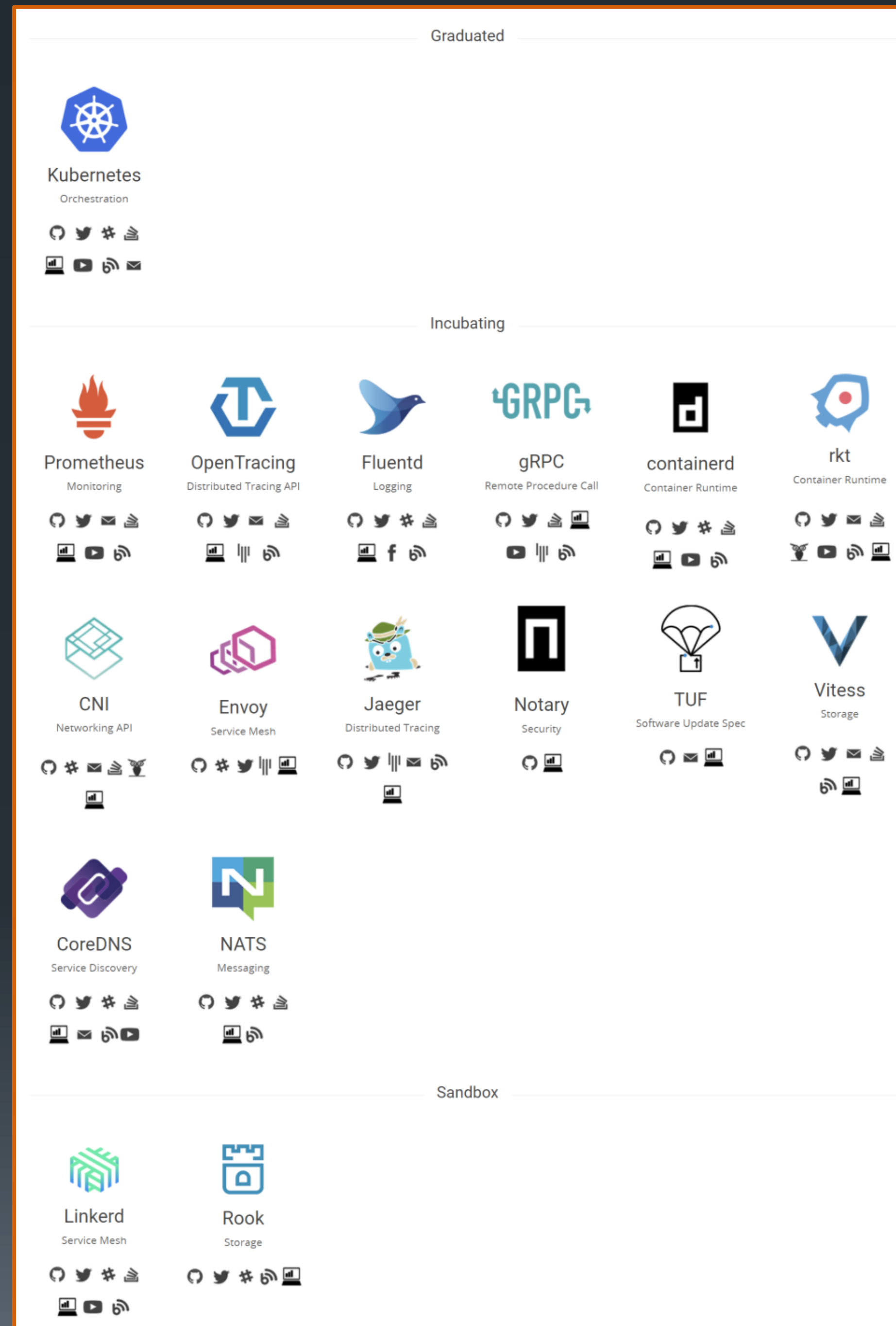
2016 Apcera Survey

Cloud Native System Traits

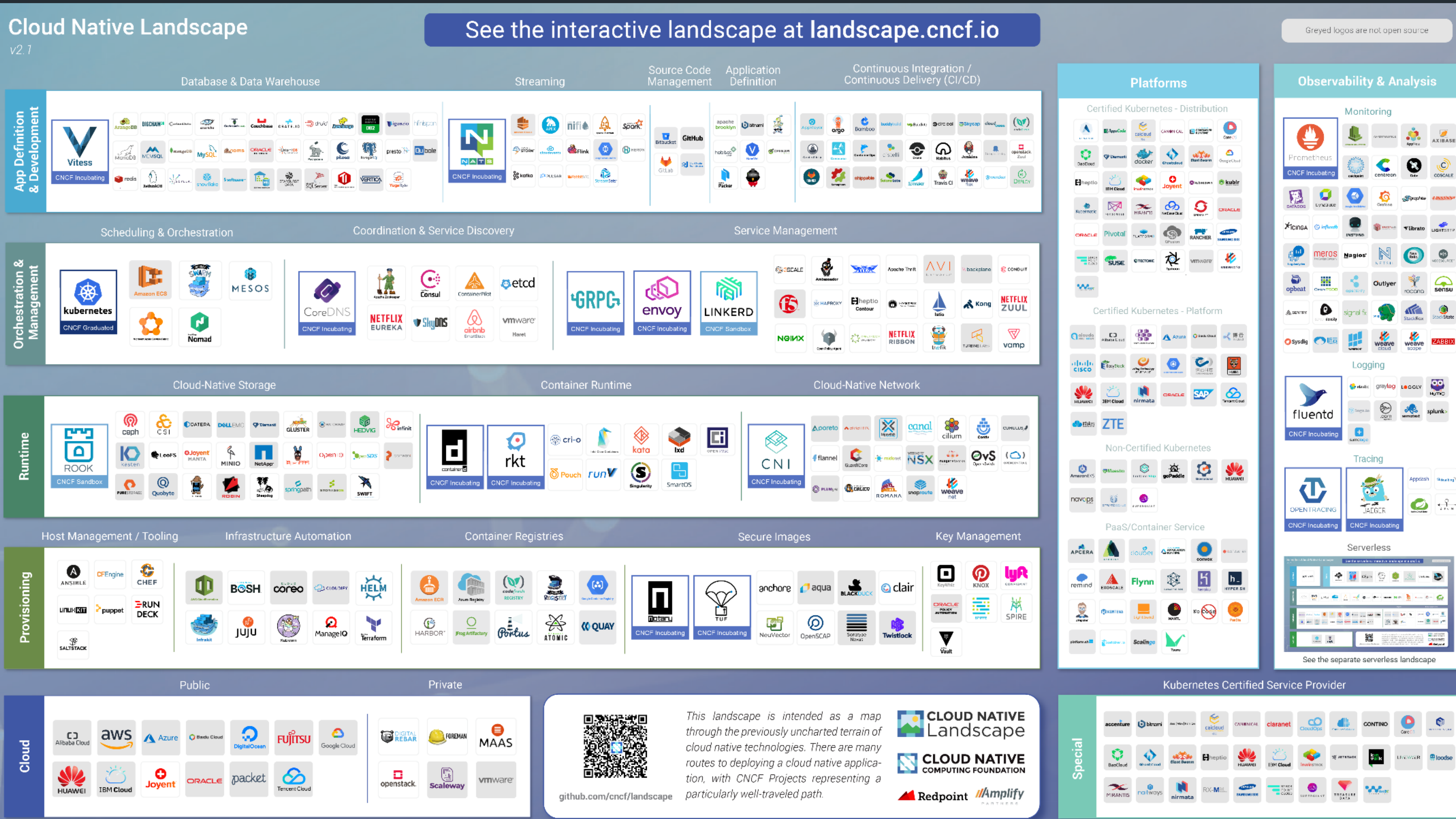
- In 2018 the CNCF refocused the official CNCF definition of cloud native systems of a more abstract set of attributes
  - Operability: Expose control of application/system lifecycle
  - Observability: Provide meaningful signals for observing state, health, and performance
  - Elasticity: Grow and shrink to fit in available resources and to meet fluctuating demand
  - Resilience: Fast automatic recovery from failures
  - Agility: Fast deployment, iteration, and reconfiguration.

Container Orchestration Initiatives

- CNCF [circa 7/2015] http://cncf.io
  - Cloud Native Computing Foundation
    - Hosted by the Linux Foundation
  - Cloud Native Applications are: container packaged, dynamically managed and micro-services oriented
  - Mission: To create and drive the adoption of a new set of common container technologies informed by technical merit and end user value, and inspired by Internet-scale computing
  - Aims to host the reference stack of technologies around container orchestration
    - Google contributed Kubernetes to the kick off the foundation, more recent additions:
      - Prometheus, Fluentd, CoreDNS, Linkerd, Jaeger, OpenTracing, gRPC, Containerd, rkt, Envoy, CNI, Notary, TUF, Vitess, NATS, Rook
  - 80+ Members including:
    - Cisco, CoreOS, Docker, Fujitsu, Google, Huawei, IBM, Intel, Joyent, Mesosphere, Red Hat, Samsung SDS, Supernap, AT&T, NetApp Amihan, Apcera, Apigee Aporeto, Apprenda, AVI Networks, Caicloud, Canonical, Capital One, Centrify, ChaoSuan, Chef, Cloudsoft, Container Solutions, Crunchy, Dao Cloud, Deis, Digital Ocean, Easy Stack, eBay, Eldarion, Exoscale, Galactic Fog, Goldman Sachs, Gronau, Heptio, Iguaz.io, Infoblox, Juniper, Livewyer, Loodse, Minio, Mirantis, NCSoft, NEC, Nginx, Packet, Plexistor, Portworx, Rancher, RX-M, Stack Point Cloud, Storage OS, Sysdig, Tigera, Treasure Data, Twistlock, Twitter, Univa, Virtuozzo, Vmware, Weaveworks, Box, Ticketmaster, Zhejiang University

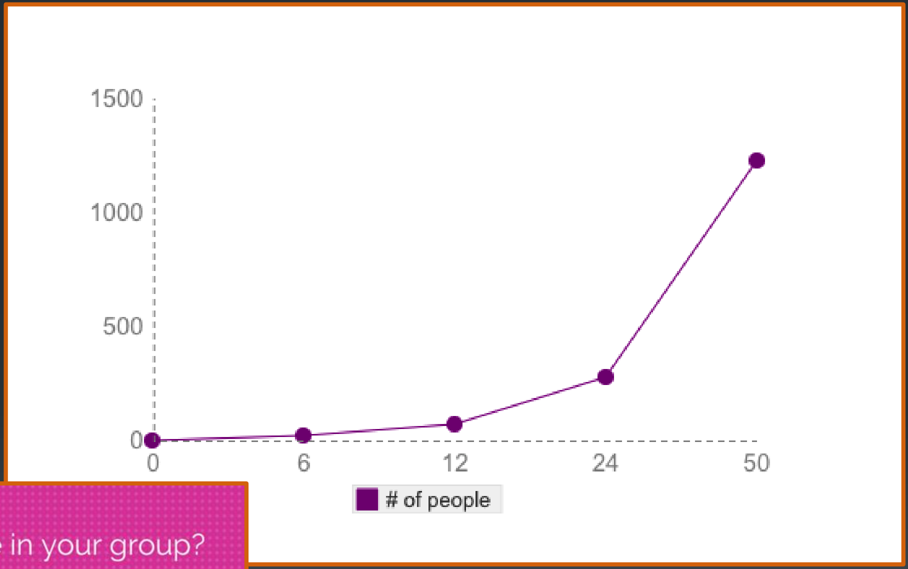


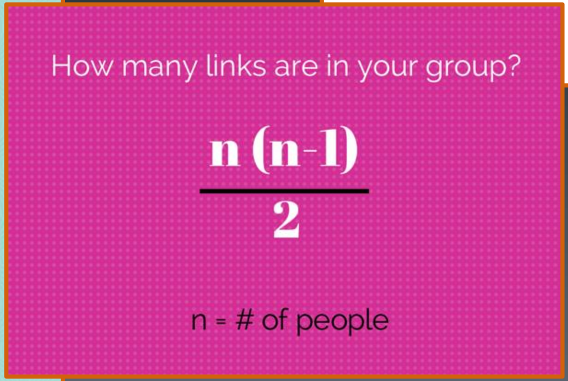
Players in the space



How micro is micro

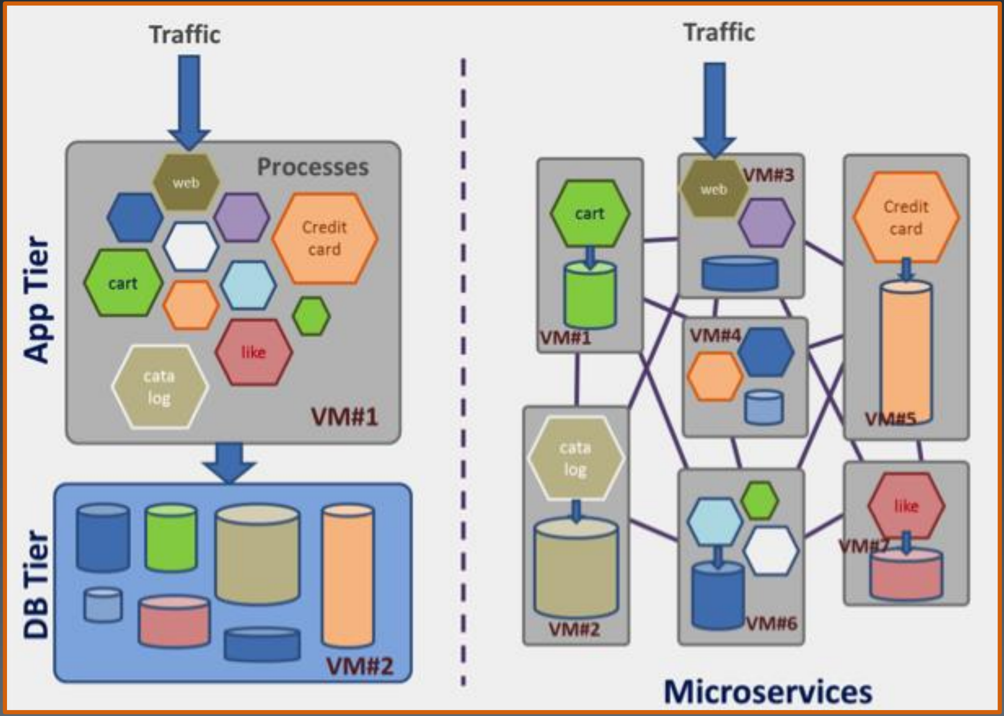
- You should be able to rewrite a microservice from scratch in 2 weeks
- A microservice should have a single crisp responsibility
- A microservice should be owned by a single team





Microservice challenges

- Explosion in the number of processes to manage
  - Requires reliable deployment solutions and automated dynamic orchestration to manage
- New and not well understood by many teams
  - Easy to slip into old habits making things worse not better
- Heavy network utilization and increased latency
  - Microservices communicate through out-of-process network interfaces
  - Slower, perhaps much slower, that in process function calls, though mitigated by async communications in some cases
- Small to medium applications may be harder to deploy and manage
  - No transactions, no single integration database
- Integration is no longer anyone in development's problem
  - Isolating teams within a single process may lead to poor integration practices in development making life harder for test and production



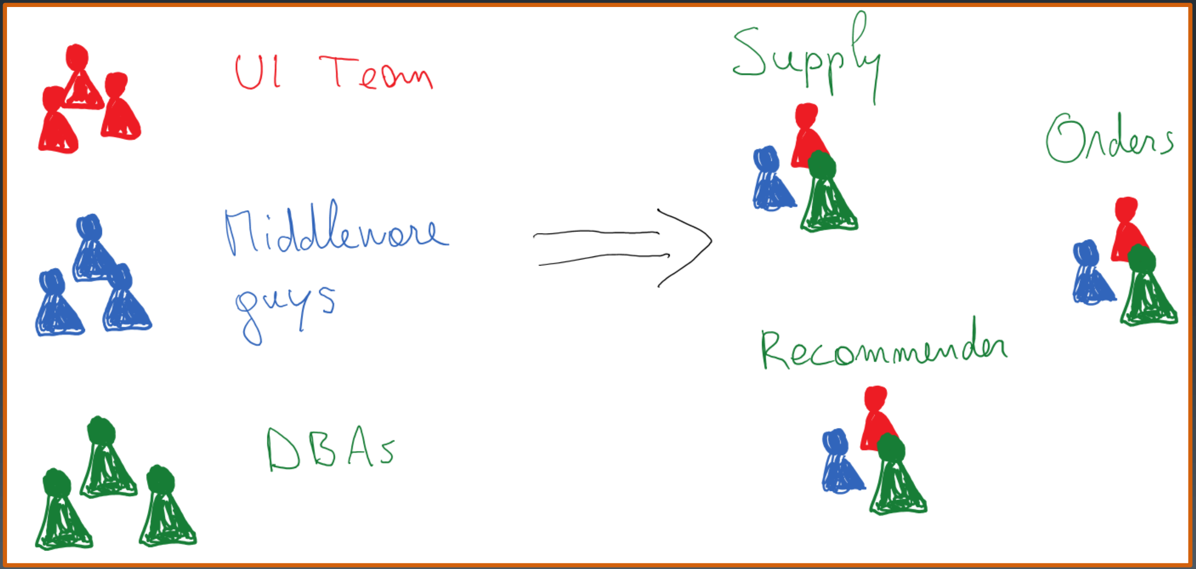
Anti-patterns

- Packing multiple services onto the same package
  - Single service packages can be cost prohibitive with VMs but works well with containers
- Too much cross service code sharing
  - Services can become coupled to internal representations and/or code
  - Decreases autonomy and requires additional coordination when making changes
- Too much polyglotism
  - Too many languages, stores or platform tool choices can make operations costly and problematic
- Shared database between different services
  - Creates tight coupling between services
  - Schema updates force updates across services
- Hard Coded Endpoints
  - Services should discover one another
- Transactions (particularly distributed)
  - Transactions require tight coupling of components
  - Scalable, loosely coupled distributed applications use eventual consistency, atomic updates and other approaches to consistency
- Persistent messaging
  - Message transactions and/or delivery guarantees
- Service instance identity
  - Microservices should implement a service, not embody it
  - End points should support elastic, scalable, multi-instance services
- Selective message delivery
  - Implies service instance identity and dependency

Microservices in Practice

- Evolution at 3 major tech firms:
- eBay
  - 5th generation today
    - Monolithic Perl
    - Monolithic C++
    - Java
    - microservices
- Twitter
  - 3rd generation today
    - Monolithic Rails
    - JS / Rails / Scala
    - microservices
- Amazon
  - Nth generation today
    - Monolithic C++
    - Perl / C++
    - Java / Scala
    - microservices

Very small autonomous teams achieve great things



Division of Labor in a Cloud Native World

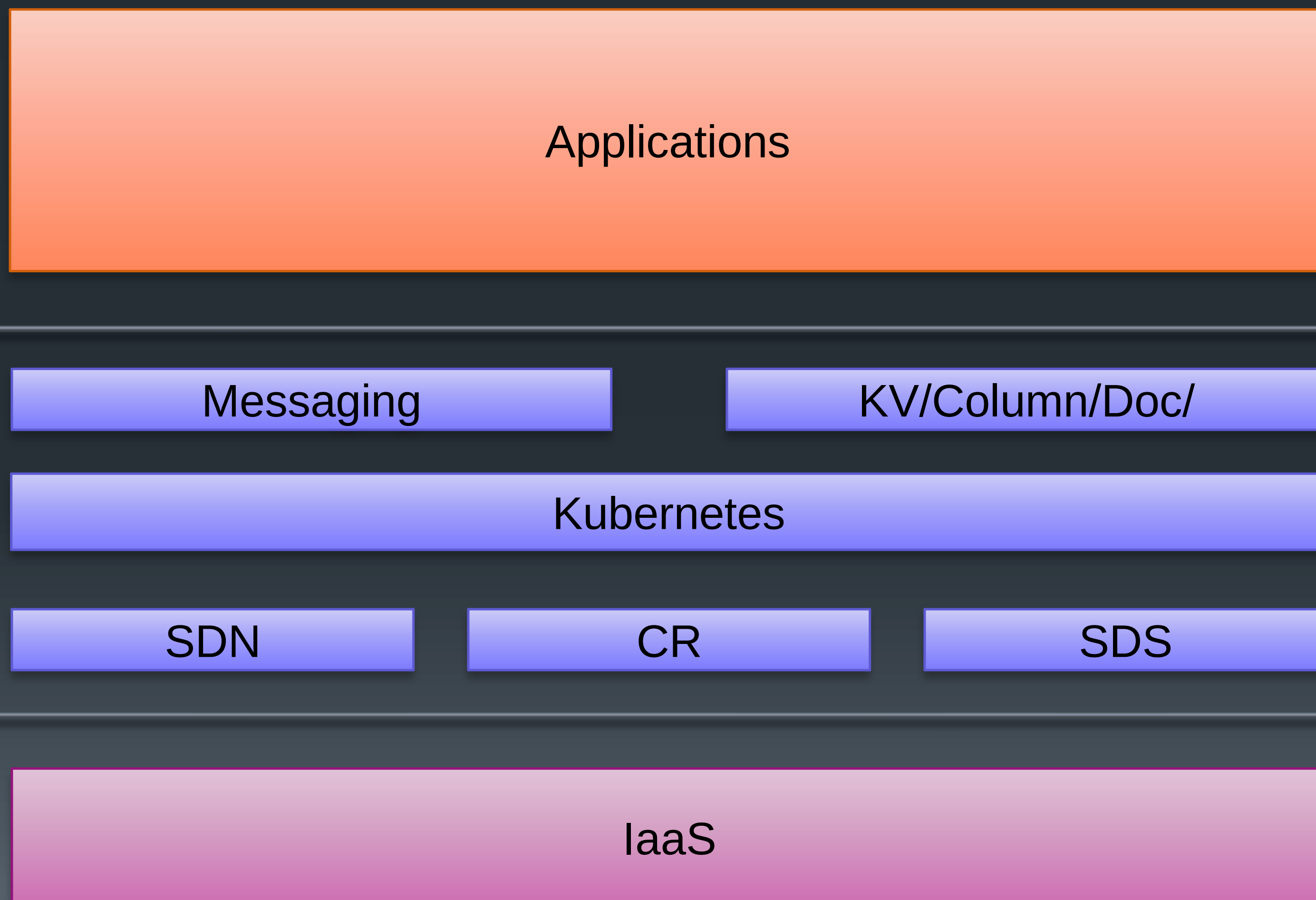
Applications
Operators: DevOps
fn/Containers/Configs
>> Deployments / Services / Jobs >>

>> StatefulSets >>

Application Platform
Operators: SREs

>> DaemonSets >>

Cloud Platform
Operators: CREs



Summary

- Microservice Architecture refers to an architectural pattern derived from SOA but focused on small single responsibility services with ubiquitous network based interfaces
- While microservices offer a range of benefits they have proven particularly powerful for organizations seeking:
  - Extreme scale
  - Low technology lock in and ability to adopt new productive tools dynamically
  - Rapid CI/CD support
- Microservice also carry risks, particularly when incomplete or inappropriate processes, tools or designs are applied

Lab 1

- Configuring a microservice development environment and creating a hello world microservice

2: Communication I

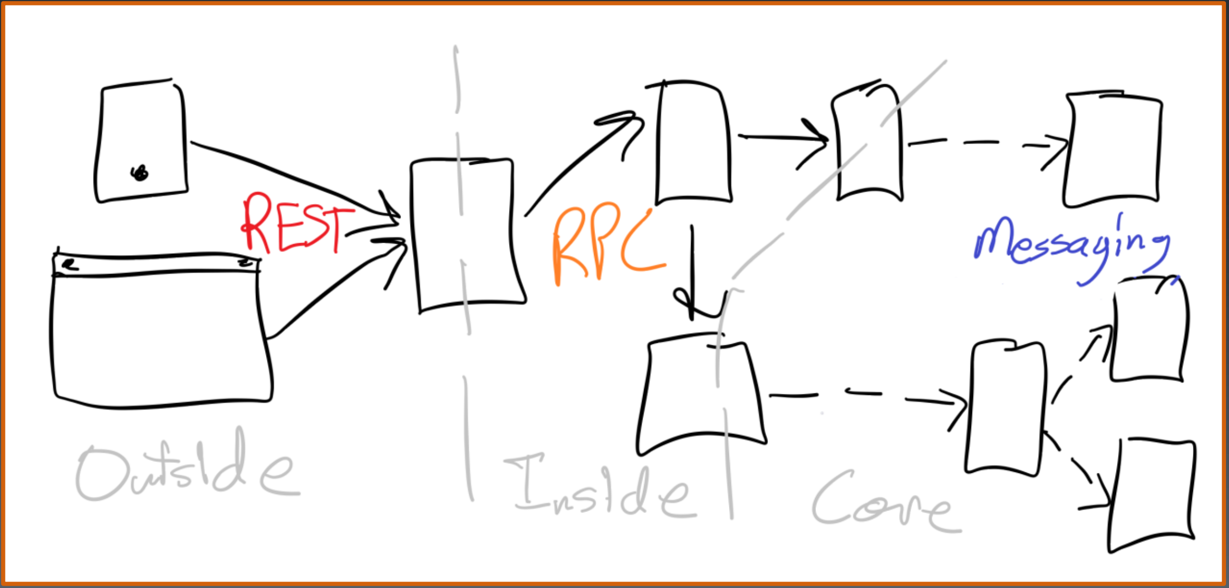
Objectives

- Describe the range on microservice communication schemes
- List the Request/Response style service communications types
- Explore interface evolution
- Define RESTful interface
- Examine REST in detail
- Examine RPC in Detail

Communications Options

- Communications are the lifeblood of microservices
- Schemes
  - Request/Response
    - REST
    - RPC
      - CNCF gRPC/ProtoBuf
      - Apache Thrift
  - Messaging
    - Broadcast
    - Pub/Sub (aka multicast)
      - Apache Kafka
      - CNCF Nats
      - EMQ
    - Anycast
    - Unicast
  - Streaming
    - Characterized by a continuous flow of data from a server to a connected client

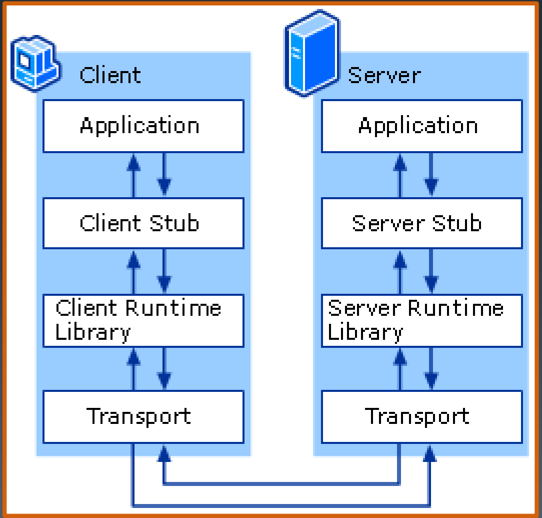
Each scheme has unique characteristics and requirements



Request/Response

- REST
  - Language/platform agnostic
  - Many supporting frameworks
  - Two main data formats
    - XML
    - JSON
- RPC
  - Long standing approach to distributed computing
  - Extends the concept of an in process function call across a network
  - Several modern schemes allow evolution and are suitable for microservice development
    - gPRC and Protocol Buffers
    - Apache Thrift
    - JSON RPC

Resource	POST create	GET read	PUT update	DELETE delete
/dogs	create a new dog	list dogs	**bulk update dogs**	delete all dogs
/dogs/1234	**error**	show Bo	if exists update Bo **if not error**	delete Bo



HTTP

- The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems and is the foundation of data communication for the World Wide Web
- The HTTP standard was coordinated by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C)
  - RFC 2616 (June 1999) defines HTTP/1.1, the version in common use
- HTTP functions as a request-response protocol in the client-server computing model
  - The client submits an HTTP request message to the server
    - A web browser is typically the client
  - The server returns a response message to the client
    - The server provides resources such as HTML files or performs other functions on behalf of the client
    - The response contains completion status information about the request and may also contain requested content in its message body
- Part of HTTPs appeal is that it is simple
  - You can code a decent Java or C++ HTTP server in a day …
  - But you don't need to because there are tens already written to choose from

HTTP Request Methods

- GET

Requests a representation of the specified resource

- HEAD

Asks for the response but without the body

- POST

Requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI

- PUT

Requests that the enclosed entity be stored under the supplied URI

- DELETE

Deletes the specified resource

- TRACE

Echoes back the received request so that a client can see what (if any) changes or additions have been made by intermediate servers

- OPTIONS

Returns the HTTP methods that the server supports for the specified URL

- CONNECT

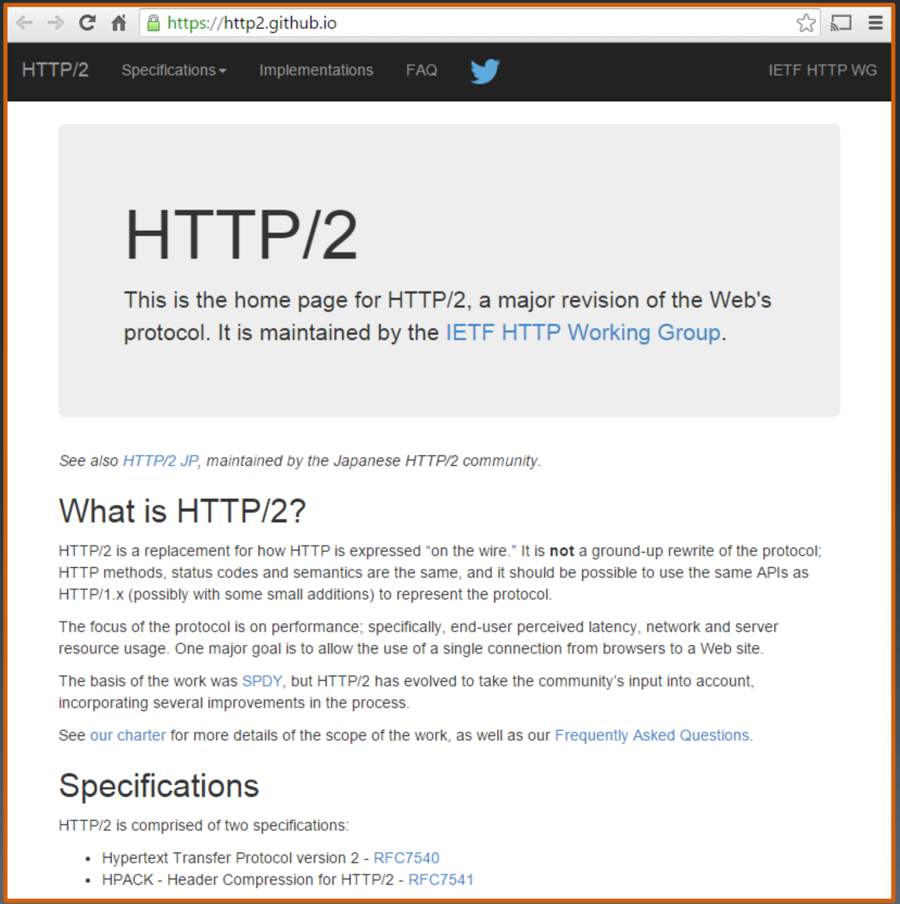
Converts the request connection to a transparent TCP/IP tunnel

- PATCH

Used to apply partial modifications to a resource

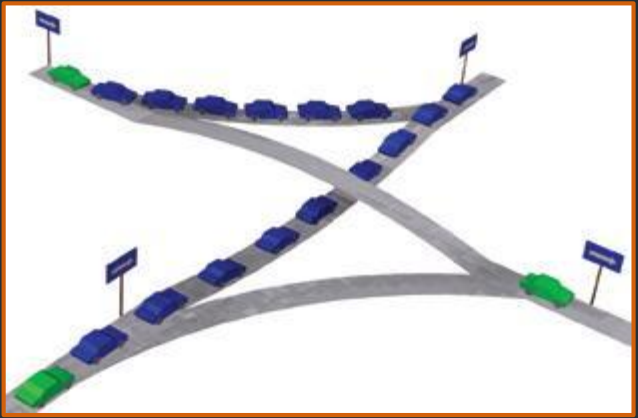
HTTP Evolution

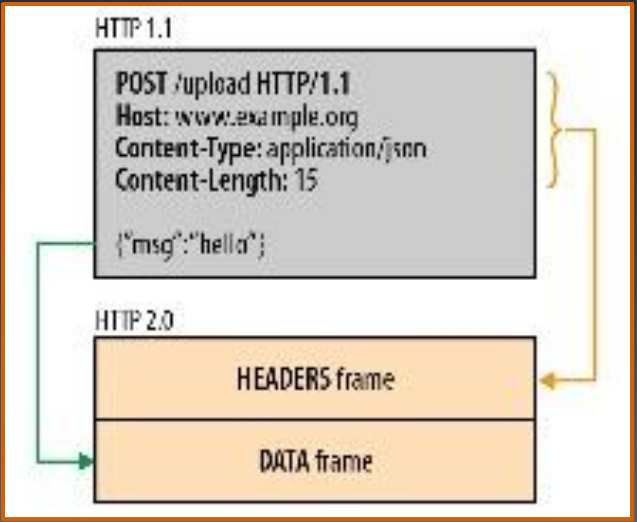
- The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems
- Standards development of HTTP is coordinated by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) through the RFC (Requests for Comments) process
- Timeline
  - 1989: Tim Berners-Lee proposes the "WorldWideWeb" project
  - 1990: HTTP becomes the foundation of data communication for the World Wide Web
  - 1991: HTTP v0.9 is established ad hoc (GET only)
  - 1995: Dave Raggett leads HTTP Working Group (HTTP WG) to expand the protocol with extended operations, extended negotiation, richer meta-information, tied with a security protocol
  - 1996: RFC 1945 establishes the HTTP V1.0 standard
  - 1997: RFC 2068 defined HTTP/1.1
  - 1999: RFC 2616 updates 2068 and is the version in common use
  - 2007: HTTPbis Working Group formed to revise and clarify the HTTP/1.1 specification
  - 2014- 06: HTTPbis WG releases six-part specification obsoleting RFC 2616:
    - RFC 7230, HTTP/1.1: Message Syntax and Routing
    - RFC 7231, HTTP/1.1: Semantics and Content
    - RFC 7232, HTTP/1.1: Conditional Requests
    - RFC 7233, HTTP/1.1: Range Requests
    - RFC 7234, HTTP/1.1: Caching
    - RFC 7235, HTTP/1.1: Authentication
  - 2015-05: HTTP/2 published as RFC 7540



HTTP/2

- Problems with old HTTP
  - No support for concurrent requests
    - HTTP/1.0 allowed only one request to be outstanding at a time on a given TCP connection
    - HTTP/1.1 added request pipelining to allow multiple outstanding requests, but still suffered from head-of-line blocking
    - HTTP/1.0 and HTTP/1.1 clients that need to make many requests in parallel use multiple connections to a server in order to achieve concurrency and reduce latency
  - HTTP header fields are often repetitive and verbose
    - This causes unnecessary network traffic and fills the initial TCP congestion window quickly, often resulting in excessive latency when multiple requests are made on a new TCP connection
- HTTP/2 solutions
  - HTTP/2 supports all of the core features of HTTP/1.1 but aims to be more efficient in several ways
  - HTTP/2 allows interleaving of request and response messages on the same connection and uses an efficient coding for HTTP header fields
    - Better network utilization because fewer network connections are required and connections live longer
  - HTTP/2 allows prioritization of requests, letting more important requests complete more quickly, improving performance
  - HTTP/2 enables more efficient processing of messages through use of binary message framing
    - Supporting server push and flow control
- What about WebSocket?!?
  - … oh yeah that … WebSocket integration with HTTP/2 was overlooked and does not exist today
  - You can upgrade from HTTP 1.1 to WebSocket or to HTTP/2
  - It appears the HTTP/2 standard will be amended to support multiple WebSocket channels concurrently with HTTP requests over a single TCP connection using HTTP/2 framing





REST

- The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation at UC Irvine (http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)
- Representational state transfer (REST) is an architectural style
  - REST is a style not a protocol, principal protocol developed along REST guidelines: HTTP
  - While HTTP supports REST it does not enforce it
    - e.g. many GET requests are implemented in such a way as to change state on the server
- REST ignores the details of component implementation and protocol syntax in order to focus on the roles of these, and their interpretation of significant data elements
- REST has been used to describe desired web architecture, to identify existing problems, to compare alternative solutions, and to ensure that protocol extensions would not violate the core constraints that make the Web successful
  - Much like a design guide applied to a commercial software project
- Fielding used REST to design HTTP 1.1 and Uniform Resource Identifiers (URI)
- The Web is a REST system

The properties of REST are:

- Performance
- Scalability of component interactions
- Simplicity of interfaces
- Modifiability of components to meet changing needs (even while the application is running)
- Visibility of communication between components by service agents
- Portability of component deployment
- Reliability

REST Constraints

- Client–server
  - A uniform interface separates clients from servers
  - Clients are not concerned with data storage
  - Servers are not concerned with the user interface or user state
- Stateless
  - No client context is stored on the server between requests
  - Each request from any client contains all of the information necessary to service the request
  - Session state is held in the client, though session state can be transferred by the server to another service such as a database to maintain a persistent state for a period of time and allow authentication
  - The client sends requests when it is ready to make the transition to a new state
  - While one or more requests are outstanding, the client is considered to be in transition
- Cacheable
  - Clients can cache responses
  - Responses must implicitly or explicitly define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests
  - Well-managed caching partially or completely eliminates some client–server interactions
- Layered system
  - A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary
  - Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches
- Code on demand (optional)
  - Servers can temporarily extend or customize the functionality of a client by the transfer of executable code
- Uniform Interface
  - The uniform interface simplifies and decouples the architecture

RESTful Web Services

- A RESTful web service is a web API implemented using HTTP and REST principles
- It provides a collection of resources, with four defined aspects:
  1. The base URI for the web API, such as http://example.com/resources/
  2. The Internet media type of the data supported by the web API
     - This is often JSON but can be any other valid Internet media type provided that it is a valid hypertext standard
  3. The set of operations supported by the web API using HTTP methods (e.g., GET, PUT, POST, or DELETE).
  4. The API must be hypertext driven
- PUT and DELETE methods are idempotent
- GET is a safe method (or nullipotent), calling it produces no side-effects
- Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs
- REST is an architectural style, unlike SOAP, which is a protocol
- Even though REST is not a standard, a RESTful implementation such as the Web typically uses standards like HTTP, URI, JSON, etc.

RESTful web API HTTP methods

Resource	GET	PUT	POST	DELETE
Collection URI, such as `http://example.com/resources`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	**Delete** the entire collection.
Element URI, such as `http://example.com/resources/item17`	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it doesn't exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.	**Delete** the addressed member of the collection.

GET

- The HTTP GET method is used to retrieve a representation of a resource
  - GET http://www.example.com/customers/12345
  - GET http://www.example.com/customers/12345/orders
  - GET http://www.example.com/bucket/sample
- In the OK path, GET returns a representation (typically in XML or JSON) and an HTTP response code of 200 (OK)
  - In an error case, usually a 404 (NOT FOUND) or 400 (BAD REQUEST)
- According to the HTTP specification GET, OPTIONS and HEAD are safe and should be used only to read data, not to change it
  - Do not expose unsafe operations via these methods, they should never change a resource
- GET, OPTIONS, HEAD, PUT and DELETE are idempotent
  - Making multiple identical requests produces the same representation every time without changing the resource beyond the first method call

HTTP Method	Idempotent	Safe
OPTIONS	yes	yes
GET	yes	yes
HEAD	yes	yes
PUT	yes	no
POST	no	no
DELETE	yes	no
PATCH	no	no

Popular REST Platforms

- Java
  - JAX-RS
    - Jersey
    - MOXy
- Python
  - Django REST Framework
  - Flask
- Scala
  - Play
- Node.js
  - Connect
  - Express
- Ruby
  - Sinatra
  - Rails-API
- PHP
  - F3

```
user@RONAM ~
$ curl http://localhost:8080/HelloWorld/rest/myresource
Got it!
user@RONAM ~
$ curl -v  http://localhost:8080/HelloWorld/rest/myresource
* Adding handle: conn: 0x1d27e80
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x1d27e80) send_pipe: 1, recv_pipe: 0
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* Connected to localhost (::1) port 8080 (#0)
> GET /HelloWorld/rest/myresource HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
* Server Apache-Coyote/1.1 is not blacklisted
< Server: Apache-Coyote/1.1
< Content-Type: text/plain
< Content-Length: 7
< Date: Thu, 17 Jul 2014 11:23:19 GMT
<
Got it!
* Connection #0 to host localhost left intact
```

```java
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

/**
 * Root resource (exposed at "myresource" path)
 */
@Path("myresource")
public class MyResource {

    /**
     * Method handling HTTP GET requests. The returned object will be sent
     * to the client as "text/plain" media type.
     *
     * @return String that will be returned as a text/plain response.
     */
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getIt() {
        return "Got it!";
    }
}
```

Clients and Connections

- Client application components use networks to access the application back end
- In order to scale to support huge numbers of clients many considerations must be taken into account at the back end perimeter
  - DNS routing
  - Load Balancing
  - Caching
  - Security
  - Etc.
- We'll need to take a deeper look at the cloud and its network

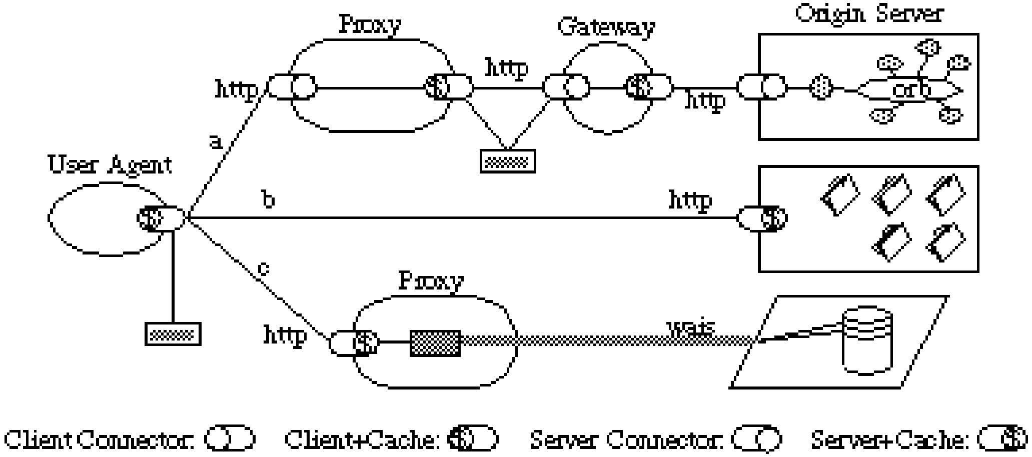


Figure 5-10. Process View of a REST-based Architecture

A user agent is portrayed in the midst of three parallel interactions: a, b, and c. The interactions were not satisfied by the user agent's client connector cache, so each request has been routed to the resource origin according to the properties of each resource identifier and the configuration of the client connector. Request (a) has been sent to a local proxy, which in turn accesses a caching gateway found by DNS lookup, which forwards the request on to be satisfied by an origin server whose internal resources are defined by an encapsulated object request broker architecture. Request (b) is sent directly to an origin server, which is able to satisfy the request from its own cache. Request (c) is sent to a proxy that is capable of directly accessing WAIS, an information service that is separate from the Web architecture, and translating the WAIS response into a format recognized by the generic connector interface. Each component is only aware of the interaction with their own client or server connectors; the overall process topology is an artifact of our view.

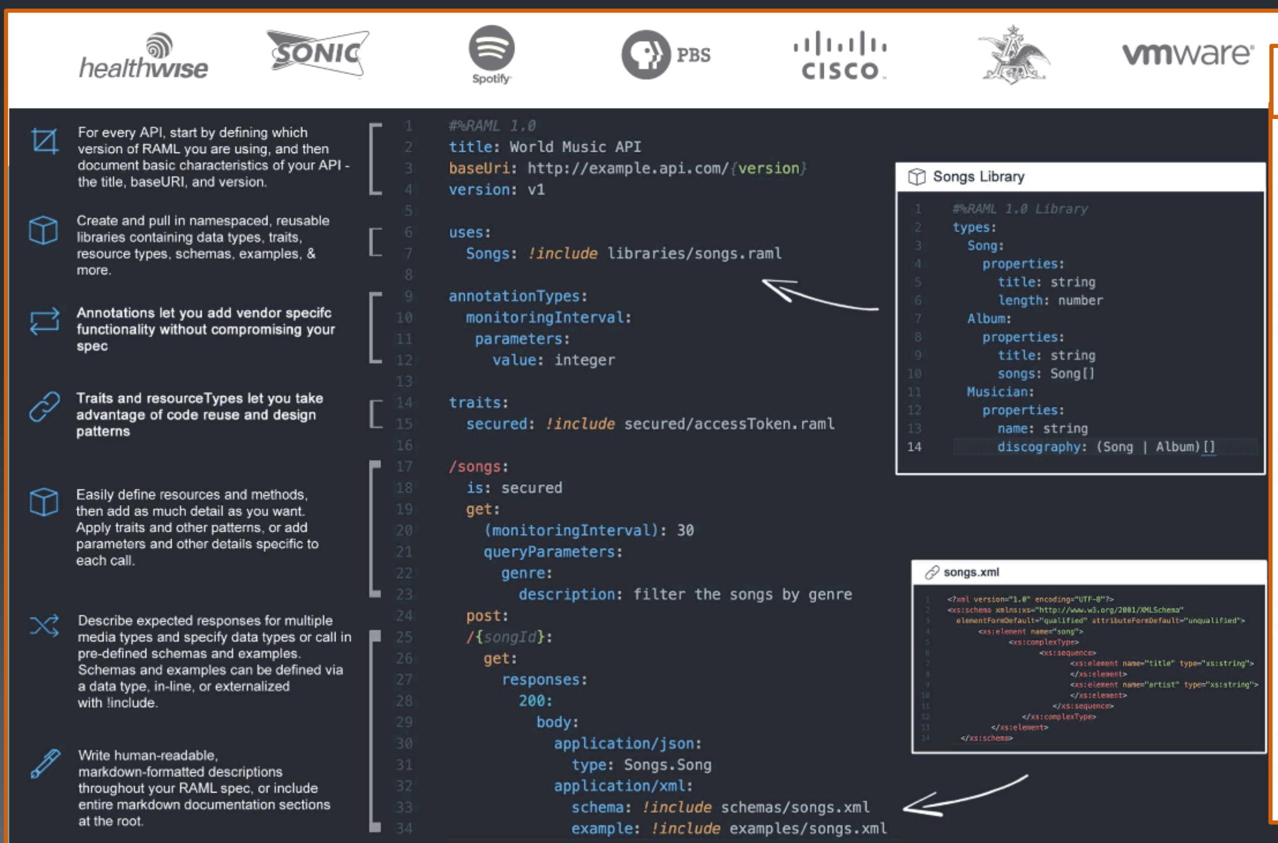
Fielding, 2000

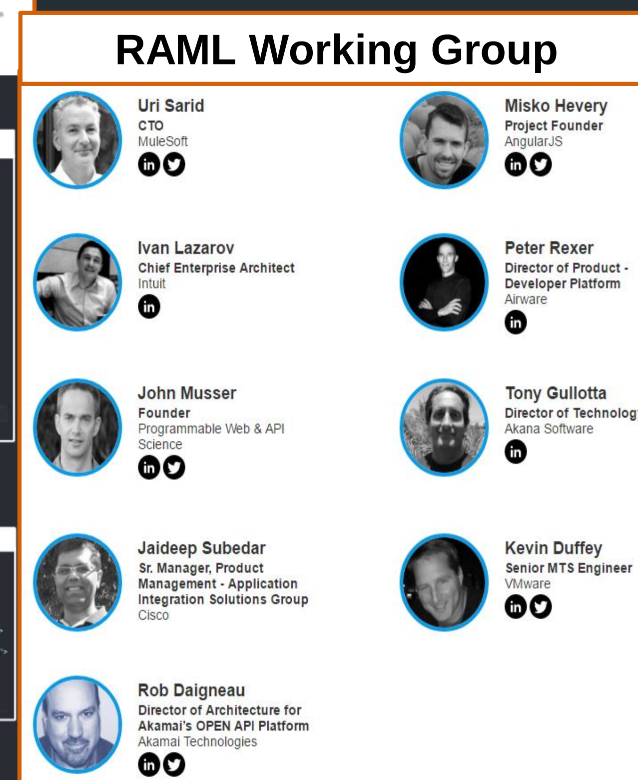
OAI

- Open API Initiative (OAI)
  - Created by an industry consortium interested in standardizing the way REST APIs are described
  - A vendor neutral open governance structure under the Linux Foundation
  - SmartBear Software donated the Swagger Specification
  - Swagger is the basis for the OAI specification
- APIs form the connecting glue between modern applications
- Nearly every application uses APIs to connect with corporate data sources, third party data services or other applications
- Creating an open description format for REST API services that is vendor neutral, portable and open is critical to accelerating the vision of a truly connected world

RAML

- RESTful API Modeling Language (RAML)
- A REST API codification competitor to OAI
- RAML was developed and is supported by a group of technology leaders dedicated to building an open, simple and succinct spec for describing APIs
- The RAML Workgroup provides ongoing contributions to both the RAML spec and a growing ecosystem of tools designed to make API-first design simple and API consumption frictionless
- RAML 1.0 was released on May 16, 2016



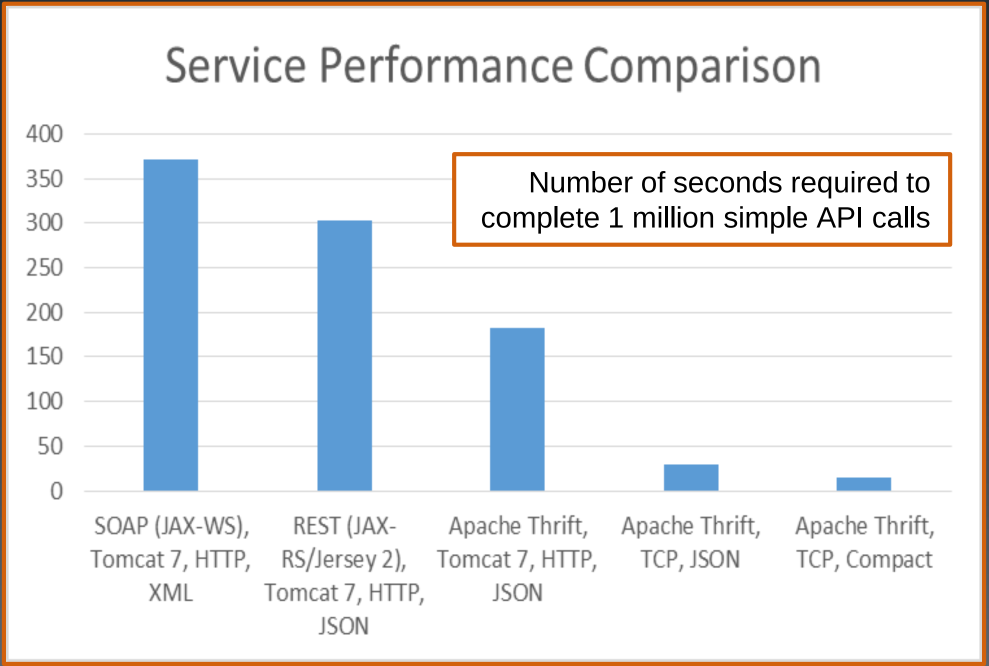


Comparing REST and RPC

- REST leverages the infrastructure of the web
  - GET caching
  - Vast ecosystems of cooperating systems (Proxies, load balancers, gateways, etc.)
  - Easy to understand and consume APIs
  - Evolvable APIs
  - Often the best choice over the Internet
- Modern RPC
  - Raw performance
  - Robust evolvable IDL
  - Good fit for monolith decomposition
  - Often a good choice for backend APIs, Messaging and NoSQL data serialization

Backend API Performance

- REST is predominantly used over the infrastructure of the Web
  - Natural synergies and benefits occur here
- In backend systems web technologies can be cumbersome
  - The is becoming less the case with HTTP/2
- Many Internet giants have invented their own IDL based backend RPC technologies to improve processing performance in the datacenter/cloud
  - Facebook -> Apache Thrift
  - Twitter -> Apache Thrift, Scrooge/Finagle [Thrift for Scala]
  - Google -> Protocol Buffers
- There are benefits to a homogeneous solution but non functional requirements must also be considered when designing end to end corporate solutions



Evolution of RPC

1980 - Bruce Jay Nelson is credited with inventing the term RPC in early ARPANET documents
- The idea of treating network operations as procedure calls

1981 - Xerox Courier possibly the first commercial RPC system

1984 - Sun RPC (now Open Network Computing [ONC+] RPC, RFC 5531)

1991 - CORBA – Common Object Request Broker Architecture
- The CORBA specification defines an ORB through which an application interacts with objects
- Applications typically initialize the ORB and accesses an internal Object Adapter, which maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies
- General Inter-ORB Protocol (GIOP) is the abstract protocol by which object request brokers (ORBs) communicate
- Internet InterORB Protocol (IIOP) is an implementation of the GIOP for use over the Internet, and provides a mapping between GIOP messages and the TCP/IP layer

1993 - DCE RPC – An open (designed by committee) RPC solution integrated with the Distributed Computing Environment
- Packaged with a distributed file system, network information system and other platform elements

1994 - MS RPC (a flavor of DCE RPC and the basis for DCOM)

1994 - Java RMI – a Java API that performs the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java objects and distributed garbage collection
- RMI-IIOP implements the RMI interface over CORBA
- Third party RMI implementations and wrappers are prevalent (e.g. Spring RMI)

1998 - SOAP (Simple Object Access Protocol) specifies a way to perform RPC using XML over HTTP or Simple Mail Transfer Protocol (SMTP) for message negotiation and transmission

2001 - Google Protocol Buffers – developed at Google to glue their servers together and interoperate between their three official languages (C++/Java/Python, JavaScript and others have since been added), used as a serialization scheme for custom RPC systems

2006 - Apache Thrift – developed at Facebook to solve REST performance problems and to glue their servers together across many languages
- The basis for Twitter Finagle, a cornerstone of the Twitter platform

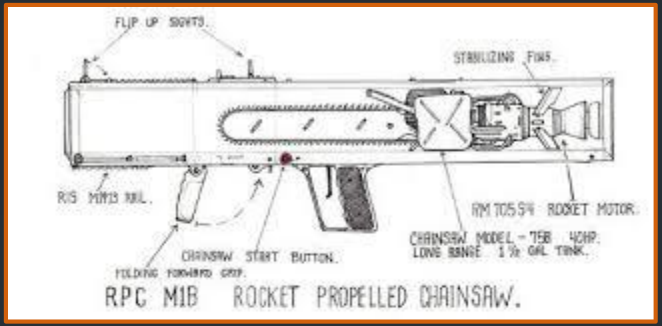
2008 - Apache Avro is a serialization framework designed to package the serialization schema with the data serialized, packaged with Hadoop

2015 - Google gRPC announced as an RPC framework operating over http/2 using protocol buffers for serialization

2017 - Google contributes gRPC to CNCF

Elements of RPC protocols

- Session Management
  - Connection or Connectionless operation
  - If connections are used how are they:
    - Established
    - Maintained
    - Terminated
- Security
  - Is authentication provided
  - Is integrity provided
  - Is confidentiality provided
- Calling Conventions
  - Message structure
  - Interactions of requests and responses
    - Normal Call/Return
    - Returnless calls (one way messages from client to server)
    - Notifications (one way messages from server to client)
  - Synchronous/Asynchronous call support
- Data Serialization
  - How are parameters and return values encoded for transmission



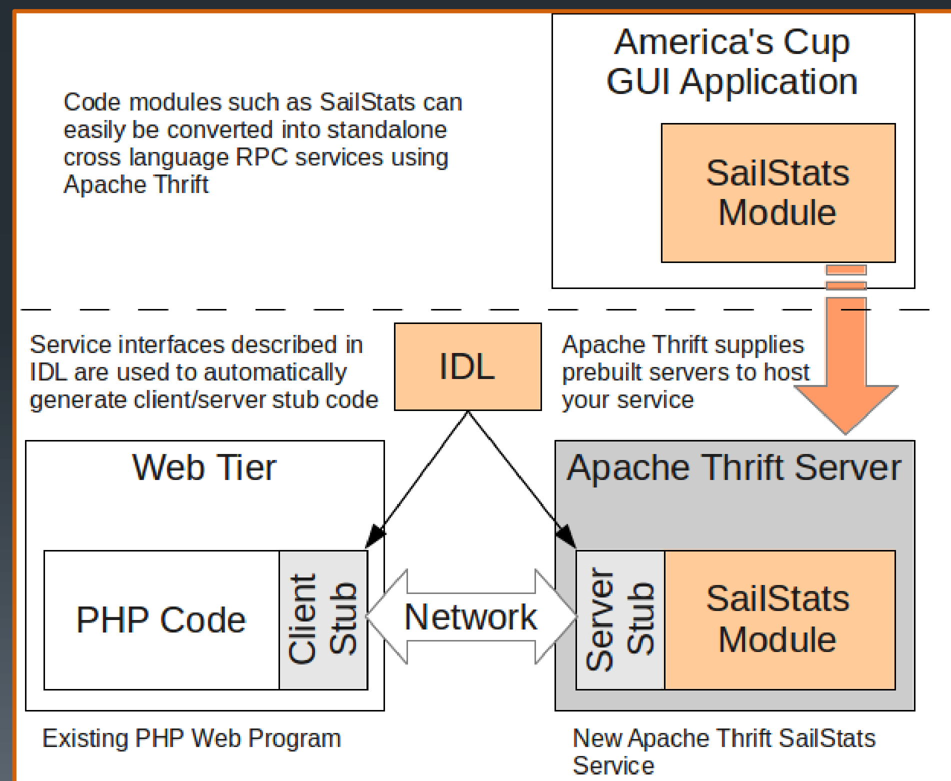
Not that RPC…

Apache Thrift (a case study)

- Apache Thrift is a light weight, fast, cross language RPC framework
  - A plug in transport layer
  - A plug in serialization layer
  - A complete RPC server library
- Commercial systems such as EverNote and open source projects such as Cassandra have adopted Apache Thrift as their principle API provider

C	C++	C#	D
Delphi	Erlang	Go	Haskell
Java	JavaScript	Objective-C	OCaml
Perl	PHP	Python	Ruby
Smalltalk			

Table 1.1 - Languages supported by Apache Thrift



Apache Thrift IDL

- Interface definition languages (IDLs) provide an implementation free view of the critical exchanges taking place between applications and application subsystems
- IDLs tend to be language and platform agnostic
- Apache Thrift IDL supports
  - Constants
  - Data Structures
  - Collections
  - Services, which are collections of functions
  - exceptions
  - And most important, interface evolution

```
service SailStats {
   double GetSailorRating(1: string SailorName),
   double GetTeamRating(1: string TeamName),
   double GetBoatRating(1: i64 BoatSerialNumber),
   list<string> GetSailorsOnTeam(1: string TeamName),
   list<string> GetSailorsRatedBetween(1: double MinRating,
                                       2: double MaxRating),
   string GetTeamCaptain(1: string TeamName),
}
```

Apache Thrift IDL Example

```
namespace * FishTrade

enum Market {
    Unknown       = 0
    Portland      = 1
    Seattle       = 2
    SanFrancisco  = 3
    Vancouver     = 4
    Anchorage     = 5
}

typedef double USD

struct TimeStamp {
    1: i16  year
    2: i16  month
    3: i16  day
    4: i16  hour
    5: i16  minute
    6: i16  second
    7: optional i32 micros
}

union FishSizeUnit {
    1: i32  pounds
    2: i32  kilograms
    3: i16  standard_crates
    4: double  metric_tons
}

struct Trade {
    1: string       fish
    2: USD          price
    3: FishSizeUnit amount
    4: TimeStamp    date_time
    5: Market       market=Market.Unknown//Market where trade occured
}

exception BadFish {
    1: string       fish       //The problem fish
    2: i16          error_code //The service specific error code
}
```

```
exception BadFishes {
    1: map<string, i16>  fish_errors //The problem fish:error pairs
}

service TradeHistory {
    /**
     * Return most recent trade report for fish type
     *
     * @param fish the symbol for the fish traded
     * @return the most recent trade report for the fish
     * @throws BadFish if fish has no trade history or is invalid
     */
    Trade GetLastSale(1: string fish)
        throws (1: BadFish bf)

    /**
     * Return most recent trade report for multiple fish types
     *
     * @param fish the symbols for the fish to return trades for
     * @param fail_fast if set true the first invalid fish symbol is thrown
     *                  as a BadFish exception, if set false all of the bad
     *                  fish symbols are thrown using the BadFishes
     *                  exception. If no bas fish are passed this parameter
     *                  is ignored.
     * @return list of trades cooresponding to the fish supplied, the list
     *         returned need not be in the same order as the input list
     * @throws BadFish first fish discovered to be invalid or without a
     *                 trade history (only occurs if skip_bad_fish=false)
     */
    list<Trade> GetLastSaleList(1: set<string> fish
                                2: bool fail_fast=false)
        throws (1: BadFish bf  2: BadFishes bfs)
}
```

Simple Thrift Service Handler in Java

IDL

```
service Message {
    string motd()
}
```

Listing 9.4 ~/thriftbook/servers/MessageHandler.java

```
import java.util.Arrays;
import java.util.List;
import org.apache.thrift.TException;

public class MessageHandler implements Message.Iface {    #A
  public MessageHandler() {
    msg_index = 0;
  }

  @Override
  public String motd() throws TException {
    System.out.println("Call count: " + ++msg_index);
    return msgs.get(Math.abs(msg_index%3));                #B
  }

  private int msg_index;
  private static List<String> msgs = Arrays.asList("Apache Thrift!!",
                                         "Childhood is a short season",
                                         "'Twas brillig");
}
```

This should be atomic with a multithreaded server (the book uses this example to demonstrate a race)

Server

```
$ ls -l
drwxr-xr-x 2 randy randy 4096 Jul 15 07:33 gen-cpp
drwxr-xr-x 3 randy randy 4096 Jul 15 07:42 gen-py
-rw-r--r-- 1 randy randy  534 Jul 16 00:53 MessageHandler.java
-rw-r--r-- 1 randy randy  470 Jul 15 22:37 simple_client.py
-rw-r--r-- 1 randy randy 1148 Jul 15 06:46 simple_server.cpp
-rw-r--r-- 1 randy randy   35 Jul 15 04:47 simple.thrift
-rw-r--r-- 1 randy randy  590 Jul 16 00:53 ThreadedServer.java
$ thrift -gen java simple.thrift
$ javac -cp /usr/local/lib/libthrift-1.0.0.jar:\
            /usr/local/lib/slf4j-api-1.7.2.jar:\
            /usr/local/lib/slf4j-nop-1.7.2.jar
            ThreadedServer.java MessageHandler.java gen-java/*.java
Note: gen-java/Message.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ java -cp /usr/local/lib/libthrift-1.0.0.jar:\
           /usr/local/lib/slf4j-api-1.7.2.jar:\
           /usr/local/lib/slf4j-nop-1.7.2.jar:\
           gen-java:\
           .
           ThreadedServer
```

Listing 9.5 ~/thriftbook/servers/ThreadedS

```java
import org.apache.thrift.TProcessor;
import org.apache.thrift.server.TServer;
import org.apache.thrift.server.TThreadPoolServer;
import org.apache.thrift.transport.TServerSocket;
import org.apache.thrift.transport.TTransportException;

public class ThreadedServer {
  public static void main(String[] args) throws TTransportException {
    TServerSocket svrTrans = new TServerSocket(8585);                 #C
    TProcessor processor = new Message.Processor<>(new MessageHandler());#D
    TServer server = new TThreadPoolServer(
             new TThreadPoolServer.Args(svrTrans).processor(processor)); #E
    server.serve();                                                    #F
  }
}
```

Client

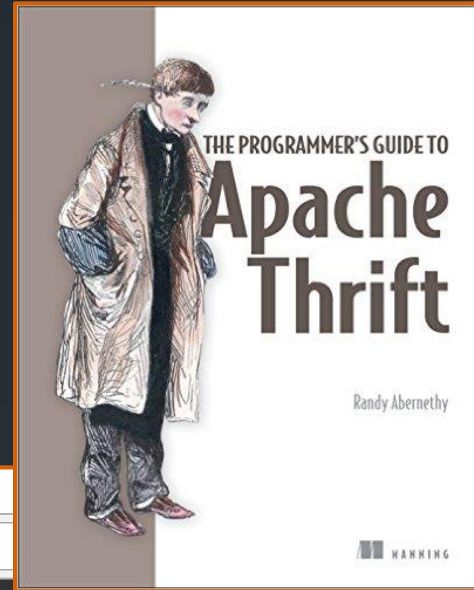
```
$ thrift -gen py simple.thrift
$ python simple_client.py
[Client] received: Childhood is a short season
Enter 'q' to exit, anything else to continue: q
$ python simple_client.py
[Client] received: 'Twas brillig
Enter 'q' to exit, anything else to continue:
[Client] received: Apache Thrift!!
Enter 'q' to exit, anything else to continue: q
$
```

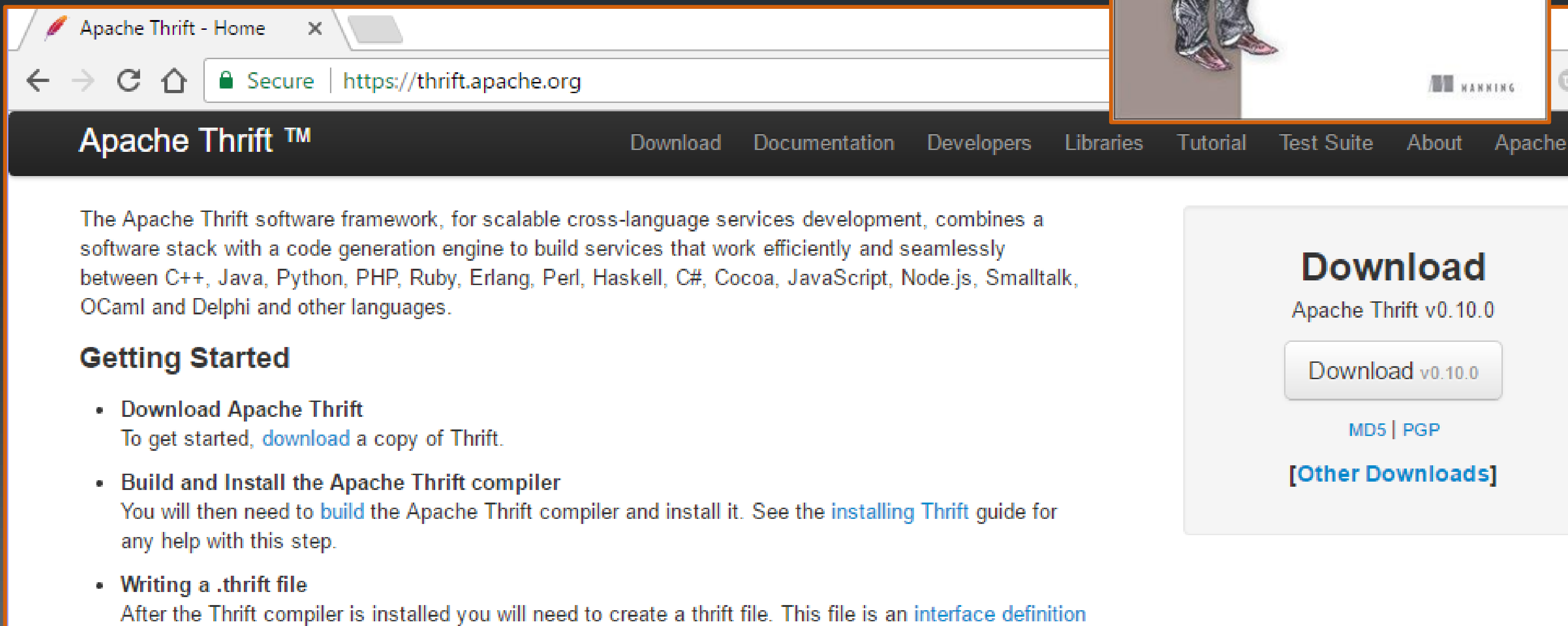
Listing 9.3 ~thriftbook/servers/simp

```
import sys
sys.path.append("gen-py")
from thrift.transport import TSocket
from thrift.protocol import TBinaryProtocol
from simple import Message

trans = TSocket.TSocket("localhost", 8585)
trans.open()
proto = TBinaryProtocol.TBinaryProtocol(trans)
client = Message.Client(proto)

while True:
    print("[Client] received: %s" % client.motd())
    line = raw_input("Enter 'q' to exit, anything else to continue: ")
    if line == 'q':
        break

trans.close()
```

Apache Thrift Info

- Docs on the Web (not super but will get you started)
- Books:
  - The Programmer's Guide to Apache Thrift
    - Randy Abernethy
    - Manning Publications





Summary

- Microservice communication schemes include
  - Request/Response
  - Messaging
  - Streaming
- Request/Response style service communications types include
  - REST
  - RPC
- RESTful interfaces can be described using OAI
- RPC interfaces can be described using IDL

Lab 2

- Creating a RESTful microservice