Pivotal Cloud Foundry®

Microservices

Pivotal

Topics

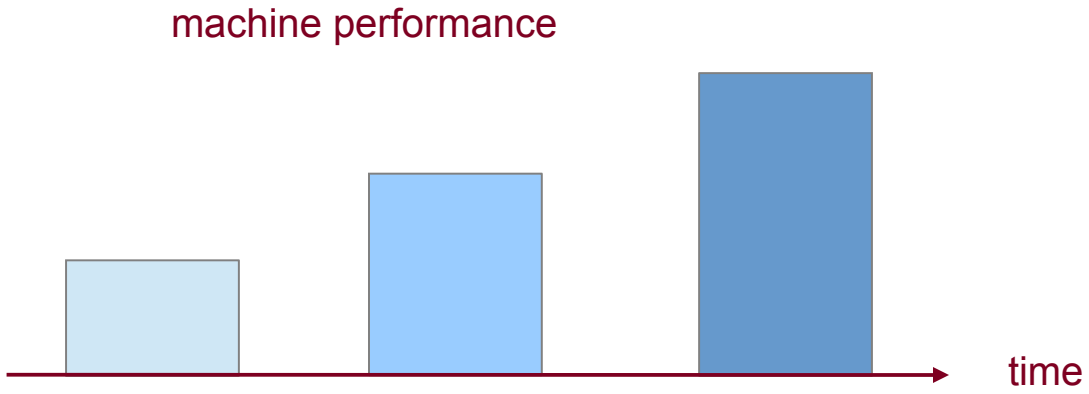
- **Simplified History**
- What are Microservices?
- Spring Cloud Services
- Migrating to Microservices

Good overview:

http://martinfowler.com/articles/microservices.html

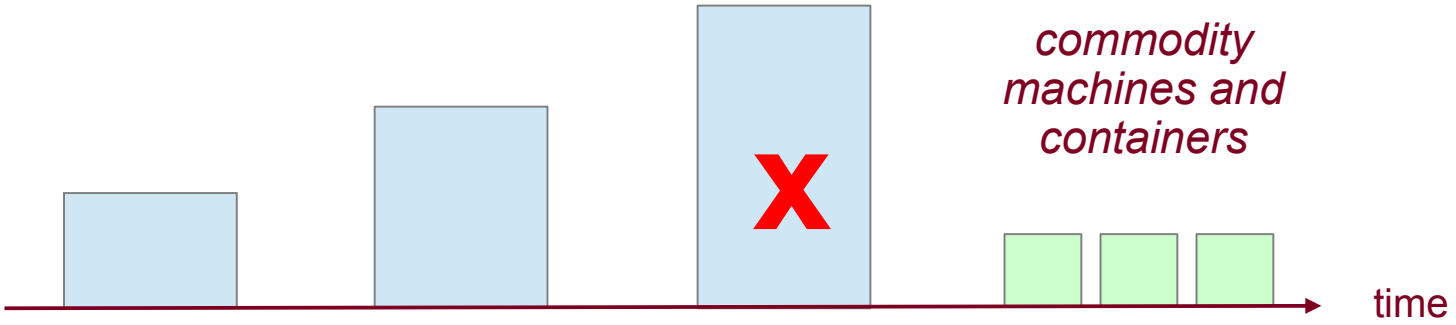
Simplified Enterprise *Compute* History – I

- As time went on, higher power machines were needed (vertical scaling)
- The machines were named (like “pets”) and *individually* configured



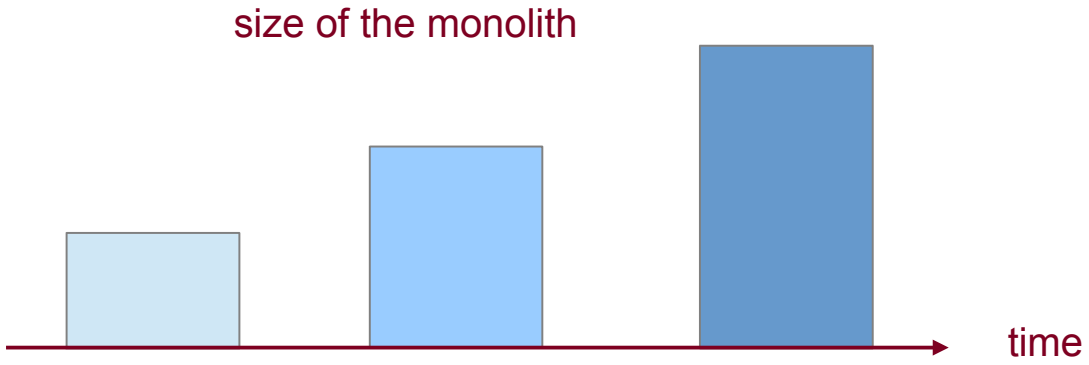
Simplified Enterprise Compute History – II

- Eventually vertical scaling fails, the model breaks down
  - Vertical scaling too expensive (or not possible)
  - Exponential demand could not be met
- Ephemeral virtual machines and containers are cloud native
  - Horizontal, elastic infrastructure
  - Everything is automated
  - Not treated uniquely – like “livestock”



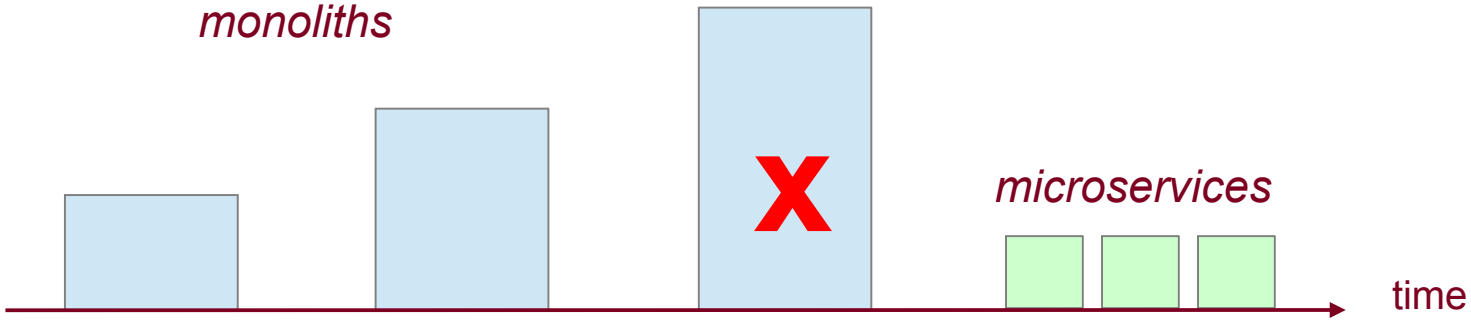
Simplified Enterprise *Application* History – I

- Enterprise applications were built as full stack "*monoliths*"
- In general, monolith size grew as compute capabilities grew



Simplified Enterprise *Application* History – II

- Monolith complexity and the move to a cloud infrastructure requires a fundamentally different application architecture
  - Inherently *distributed* and *elastic*
- Microservices following *cloud native* design principles are an approach to developing on cloud infrastructure

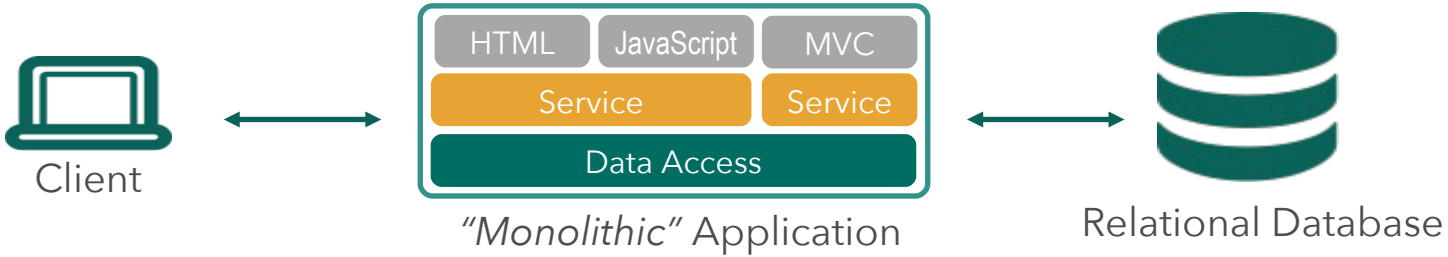


Topics

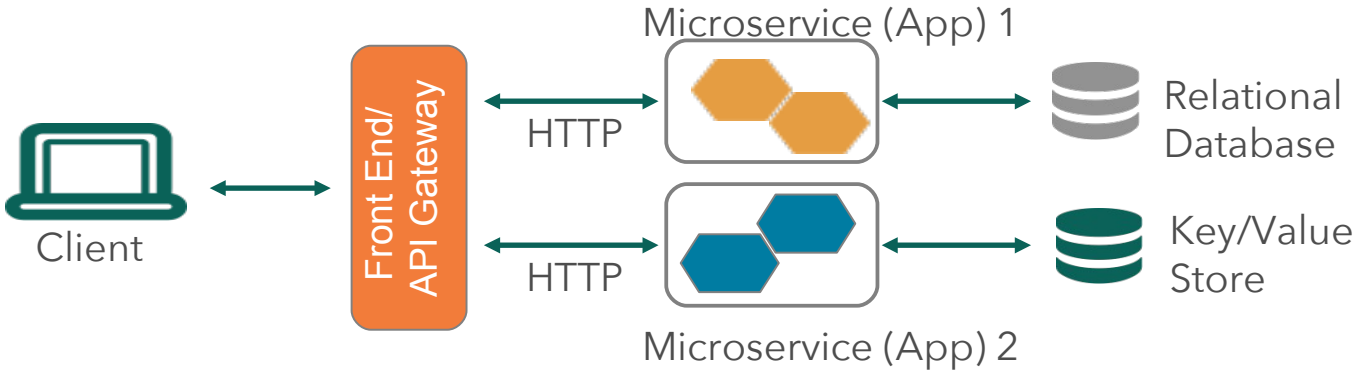
- Simplified History
- **What are Microservices?**
- Spring Cloud Services
- Migrating to Microservices

Monolithic vs. Microservice Architectures (Simplified)

- Classic *three-tier* application



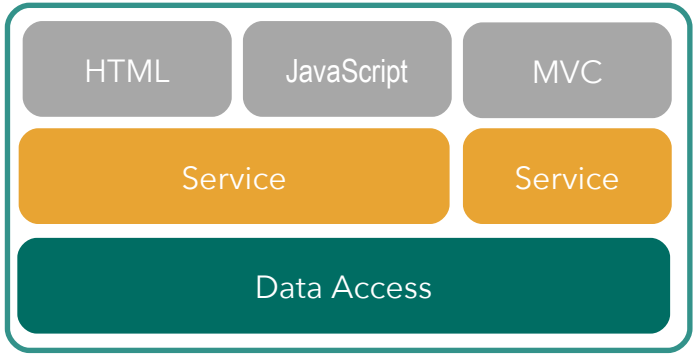
- Microservices architecture



Note- Many monoliths have always had characteristics of microservices

Monoliths- Simplified

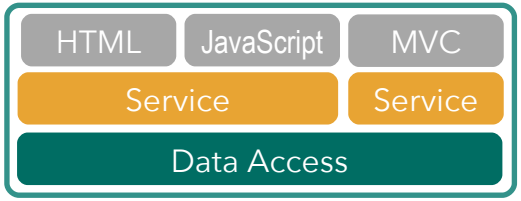
- Multi-purpose OS and application servers runs the “big app”
- Sparse, coordinated releases
- Separate ops, DBAs, dev teams
- Coupled dependencies



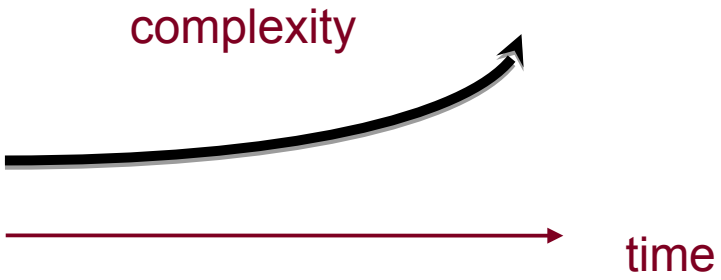
“Monolith” Application

Monoliths- Problems

- Complexity reduces agility
  - “Hit a wall”
  - Affects the ability to compete
- Difficult to understand and contribute
- Coupling creates unintended consequences and delays
- May not work well with cloud infrastructure
  - Not 12-factor conformant
  - Doesn’t scale well



“Monolith” Application

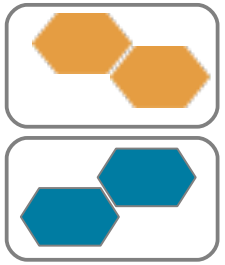


Example- Amazon Switching to Microservices

- Sometime around 2002, Jeff Bezos issued a mandate
  - All teams will henceforth expose their data and functionality through service interfaces
  - Teams must communicate with each other through these interfaces
    - There will be no other form of inter-process communication allowed
  - It doesn't matter what technology you use
  - …
- This reinforced their microservices strategy

http://apievangelist.com/2012/01/12/the-secret-to-amazons-success-internal-apis/
http://thenewstack.io/led-amazon-microservices-architecture/

Microservice Features



- API interaction only
  - Loosely coupled
  - Often RESTful APIs
- Bounded contexts / domain-driven design
  - Single view of data
- Independently deployable apps
- Polyglot persistence
  - Each service uses the most suitable storage system
  - Relational DB, key-value store, document store …
- Multi-language (if desired)
- Independently scalable

Microservice Teams

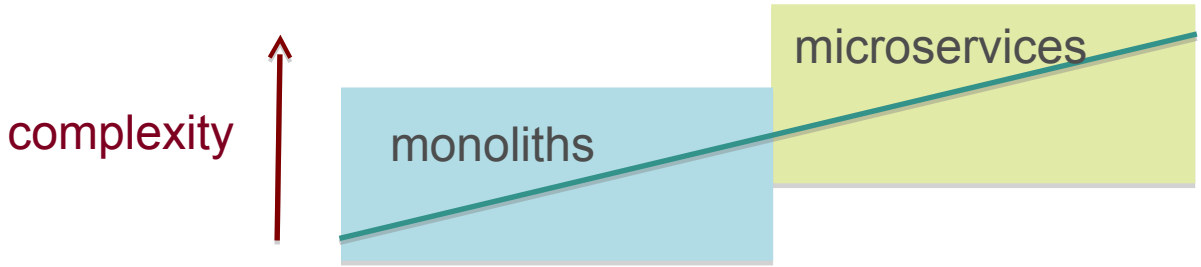
- Small teams communicating through API contracts
  - “Two pizza” teams
- Develop, test and deploy each service *independently*
- Often test-driven
  - Team A provides tests for team B to run on their microservice- “this is what we need from your service”
  - The tests must pass- that and the API is the contract between teams

http://blog.idonethis.com/two-pizza-team/

Pivotal

12 Factor Monolith

- A monolith can be a 12 factor app!
- There is no requirement to move to microservices
  - Simpler projects probably don't need it
  - Big projects with a lot of team members probably do
  - Probably should use the principles of microservices either way
    - For example, an API-only interface between components



Tradeoffs

- **Monolith**
  - Easier to build at first
  - But *ultimately* more complex to enhance and maintain
- **Microservices**
  - Harder to build at first
  - *Ultimately* simpler to extend, enhance and maintain
  - Scaling out (more processes) easier
  - Many more moving parts to manage

Why a Platform?

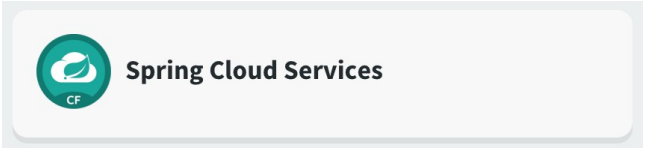
CLOUD FOUNDRY

- Deploying distributed systems is complicated
  - Security, resilience, redundancy, load-balancing
- But there are known patterns to solving distributed problems
- A platform provides the necessary tools
  - Natural fit for deploying a microservice-based system
  - Application instances are the unit of deployment
  - Can be started, stopped and restarted independently on-demand
  - Provide dynamic load-balancing, scaling and routing

Topics

- Simplified History
- What are Microservices?
- **Spring Cloud Services**
- Migrating to Microservices

Spring Cloud Services



- The Cloud Foundry platform is designed for cloud-native apps, whether they are monoliths or microservices-based
- Spring Cloud Services provides added functionality for applications with many microservices
- Implemented as services in the Marketplace
- Based on Netflix OSS and Spring Cloud

http://docs.pivotal.io/spring-cloud-services/

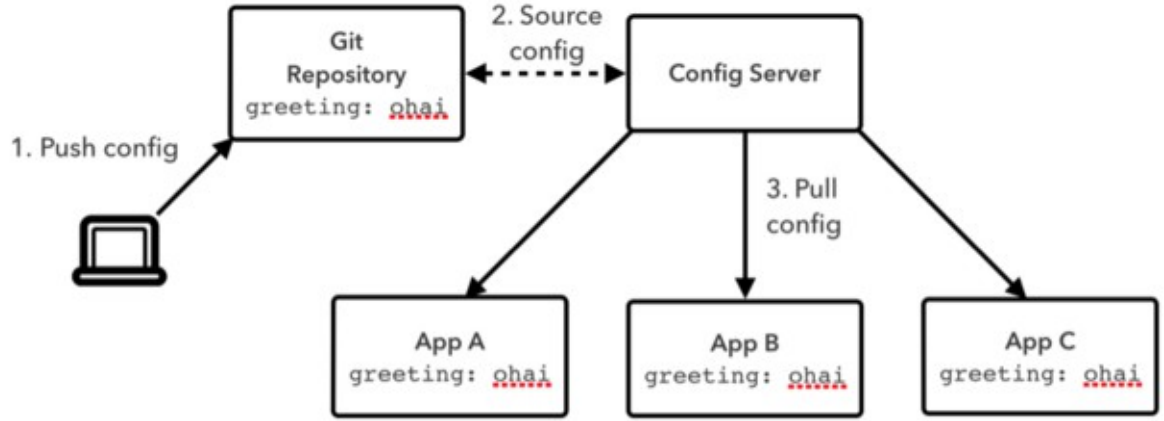
NETFLIX | OSS

Pivotal

Config Server

NETFLIX | OSS

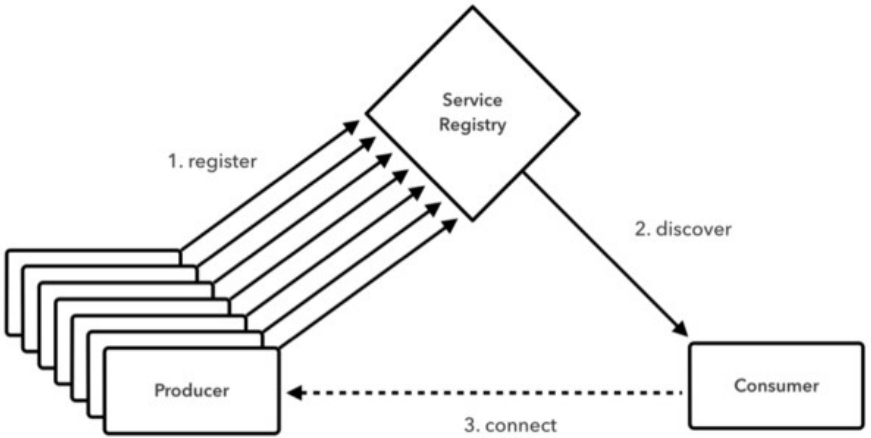
- Cloud Foundry provides configuration via environment variables, as desired with 12-factor apps
- The Config Server is an externalized application configuration service, extending the capabilities of the platform



Service Registry

NETFLIX | OSS

- Services handled through the managed and user-provided services work well in many cases
- For applications with many microservices, this can become difficult to manage
- The service registry is an implementation of the service discovery pattern

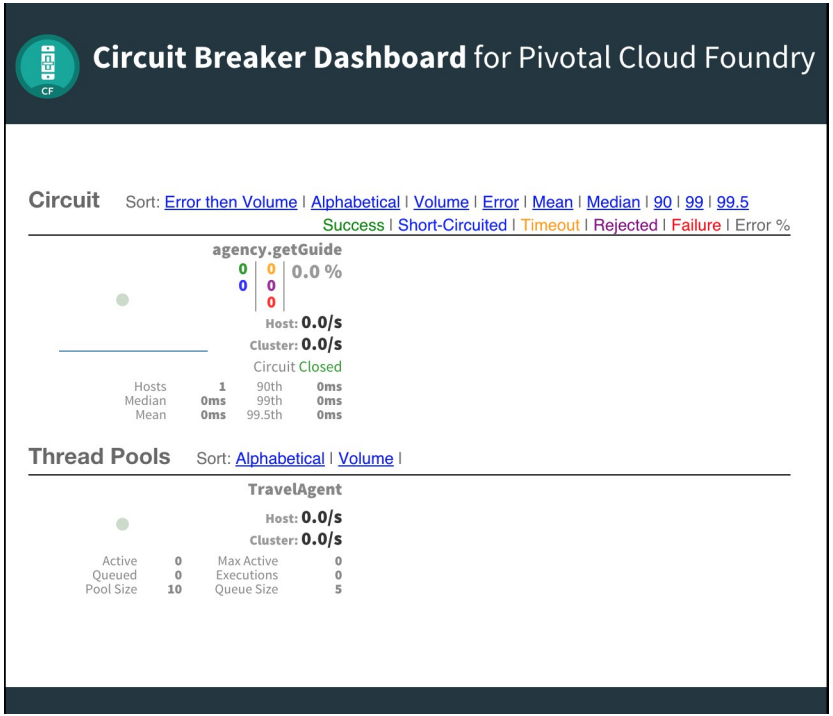


Pivotal

Circuit Breaker Dashboard

NETFLIX | OSS

- Any application should handle failures gracefully
- For applications with many microservices, failures are much more likely to occur
- Circuit breakers are a standard pattern in applications for handling failures
- Apps that implement circuit breakers can bind to the Circuit Breaker Dashboard service
- The dashboard consumes a streaming endpoint exposed by the app and renders the state of its circuit breakers

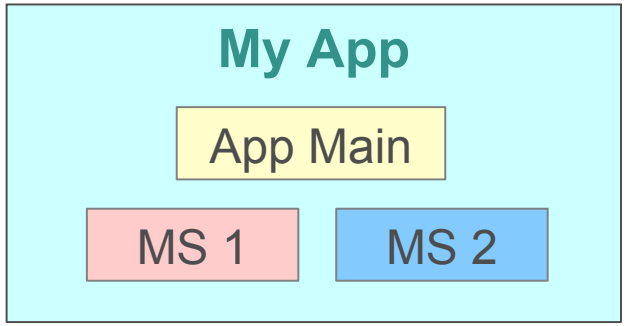


Topics

- Simplified History
- What are Microservices?
- Spring Cloud Services
- **Migrating to Microservices**

Route to Microservices: New App

- Start with a new app (“greenfield”)
  - Keep it simple, at first
  - Apply 12-factor patterns
    - http://12factor.net
  - *Cloud-ready at every stage*
  - Focus on APIs as contracts



- Decompose into microservice(s)
  - Enables separately manageable and deployable units
  - Each can use own storage solution (*polyglot persistence*)

Route To Microservices: Existing App

- Focus on parts of the app that require agility or have operational issues
- Develop *new* functionality as microservices *around* existing single-process application
  - Use Facades/Adapters/Translators to integrate them
- *"Strangle the monolith"*
  - Refactor *existing* monolith functionality into new microservices
  - Long-term evolution:
    - Monolith withers to nothing
    - Or is reduced to a solid, *reliable* core that is not worth refactoring (because we *know* it works)

Summary

- A 12-factor app can be a monolith
- To optimize agility for large applications, use microservices
- Cloud Foundry supports monolithic and microservices-based applications, as long as they are 12-factor
- Spring Cloud Services excels at handling applications with many microservices