Cloud Native Architecture Patterns and Katas
GIDS 2018
Matt Stine (@mstine)
http://www.mattstine.com
Introduction
Class Overview

By the end of this workshop, you’ll understand:

- The business drivers influencing companies to leverage Continuous Delivery, DevOps, and Cloud Native architectures.
- The unique characteristics of cloud infrastructure and how architectures can exploit these characteristics.
- How to work with an evolving cloud native architectural pattern language called “Bricks and Mortar.”
Class Overview

And you'll be able to:

- Articulate the high-level narrative of cloud native architecture and why it is important to your business.
- Articulate the paradigm shift involved in cloud native architectural thinking.
- Describe the cloud native architectural patterns, what problems they solve, the pros and cons of various implementation approaches, and the relationships between the patterns.
- Apply cloud native architecture patterns to various practice “katas” to prepare for future use on real projects.
Your Instructor

- 18 years in the Enterprise IT industry
- 6 years as a Cloud Platform and Application Architect
- Frequent speaker on the conference circuit
- Host of the Software Architecture Radio podcast
  http://softwarearchitecturerad.io
Your Instructor

http://www.oreilly.com/programming/free/migrating-cloud-native-application-architectures.csp
Class Components

• Lecture
• Q&A Sessions
• Architecture Kata Sessions
Class Logistics

- We're scheduled to run from 9:30 until 17:00.
- 15 minute breaks at 11:00 and 15:15
- Lunch is served from 12:45 to 13:45
Agenda

• Class Introduction
  9:30 - 9:35

• The Business Drivers for Architectural Change
  9:35 - 9:50

• A High-Level Overview of Continuous Delivery and DevOps
  9:50 - 10:05

• The Unique Characteristics of Cloud Infrastructure
  10:05 - 10:20

• Cloud Native Architecture Concepts
  10:20 - 10:40
Agenda

• Introduction to the Brick and Mortar Pattern Language  
  10:40 - 10:50

• Externalized Configuration Pattern  
  10:50 - 11:00

• Break  
  11:00 - 11:15

• Externalized State Pattern  
  11:15 - 11:25

• Externalized Channels Pattern  
  11:25 - 11:35
Agenda

- Runtime Reconfiguration Pattern
  11:35 - 11:45

- Concurrent Execution Pattern
  11:45 - 11:55

- Brick Telemetry Pattern
  11:55 - 12:05

- Q&A Session: Morning Topics
  12:05 - 12:25

- Service Discovery Pattern
  12:25 - 12:35
Agenda

- Architecture Katas Set Up
  12:35 - 12:45

- Lunch
  12:45 - 13:45

- Brick Kata: Preparation and Discussion
  13:45 - 14:30

- Brick Kata: Peer Review
  14:30 - 14:55

- Edge Gateway Pattern
  14:55 - 15:05
Agenda

- Fault Tolerance Pattern
  15:05 - 15:15

- Break
  15:15 - 15:30

- Event-Driven System Pattern
  15:30 - 15:40

- Contract Management Pattern
  15:40 - 15:50

- Integration Telemetry Pattern
  15:50 - 16:00
Agenda

• Mortar Kata: Preparation and Discussion
  16:00 - 16:45

• Mortar Kata: Peer Review
  16:45 - 17:10
The Business Drivers for Architectural Change
Disruption
Software is eating the world.

- Mark Andressen
Disruptive Characteristics

- Software is primary engagement model
- New and innovative business models
- Fast and frequent deliveries
- Hypothesis-driven development
Agility

The Waterscrumfall
Waterscrumfall
Consequences

- Slow Delivery
- Large Batch Sizes
- Infrequent Feedback
- Increased Waste
Digital Transformation
Disruptive companies are also approaching resiliency differently.
Stop trying to prevent mistakes.
Embrace failure.
Resiliency

From MTBF to MTTR
We need better tools and techniques.
Visibility
Fault Isolation
Fault Tolerance
Resiliency

Scalability
Automated Recovery
A High-Level Overview of DevOps and Continuous Delivery
THE GREAT CONFLICT
My Definition:

DevOps represents the idea of tearing down organizational silos and building shared toolsets, vocabularies, and communication structures in service of a culture focused on a single goal: delivering value rapidly and safely.
THE THREE WAYS
DevOps Handbook

HOW TO CREATE WORLD-CLASS AGILITY, RELIABILITY, & SECURITY IN TECHNOLOGY ORGANIZATIONS

Gene Kim, Jez Humble, Patrick Debois, & John Willis

FOREWORD BY JOHN ALLSPAW
The First Way: Flow
The Second Way: Feedback
The Third Way: Continual Learning and Experimentation
My Definition:

Technically supporting the concept to cash lifecycle by proving every source code commit to be deployable to production in an automated fashion.
Ingredients

• Configuration Management
• Continuous Integration
• Automated Testing
Continuous Delivery

CI Developer Workflow
Continuous Delivery

The Deployment Pipeline
The Unique Characteristics of Cloud Infrastructure
My Definition:

Any computing environment in which computing, networking, and storage resources can be provisioned and released elastically in an on-demand, self-service manner.
Deployment Models

- Public
  - Amazon Web Services
  - Google Cloud Platform
  - Microsoft Azure
- Private
  - VMware vSphere
  - OpenStack
- Community
- Hybrid
Cloud Infrastructure

Service Models

The *aaS Pyramid
API Driven

- Automation
- Audit
- Authorization
- Accounting
Speed

If you need a component, create it!

- Load Balancers
- Databases (SQL/NoSQL)
- Message Queues
- Private Networks
- Storage Volumes
Cloud Infrastructure

Speed

Can eliminate:

• Ticket Systems
• Approval Processes
• Waiting Queues
• Configuration Errors
Speed

As fast as you can design the system architecture that you need, you can usually provision and begin using it.
Elastic

Goodbye Capacity Planning!
Elastic Capacity Planning

- Peer into the crystal ball...
- "What's the most capacity we'll need?"
- Guess incorrectly...
  - Blow available capacity on Black Friday
  - Hundreds of idle CPUs
Elastic

As demand increases, we simply expand capacity by provisioning more resources to service that demand.
Elastic

As demand decreases, we simply contract capacity by returning resources to the pool.
Cloud Native Architecture Concepts
Modularity
Think About the Three Ways

We want a *quantum* of this experience.
Decomposition Strategies
What’s yours?

- Bounded Contexts
- Value Streams
- Single Responsibility Principle
- Failure Domains
- Anti-Corruption Layers
Architecting for DevOps

Strategies are not mutually exclusive!
So What About Modularity?

- Loose Coupling
- High Cohesion
- Encapsulation
- Well-Defined Interface
Gratuitous Nod to Microservices!
If a microservice isn't giving you *Three Ways Value*, you probably don't need it.
Conway’s Law

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
If your architectural and organizational decomposition strategies don’t align well, then CONWAY WILL FIGHT YOU!
Observability
Think About the Three Ways

We need feedback to create a safer system of work and to (in)validate our hypotheses.
See Failure When It Happens
Measure Everything
What is Normal?

- Values
- Rates of Change
- Mean?
- P95/99/99.9?
What is Normal?

<table>
<thead>
<tr>
<th>Site</th>
<th># of requests</th>
<th>page loads that would experience the 99th percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazon.com</td>
<td>190</td>
<td>85.2%</td>
</tr>
<tr>
<td>kohls.com</td>
<td>204</td>
<td>87.1%</td>
</tr>
<tr>
<td>jcrew.com</td>
<td>112</td>
<td>67.6%</td>
</tr>
<tr>
<td>saksfifthavenue.com</td>
<td>109</td>
<td>66.5%</td>
</tr>
<tr>
<td>nytimes.com</td>
<td>173</td>
<td>82.4%</td>
</tr>
<tr>
<td>cnn.com</td>
<td>279</td>
<td>93.9%</td>
</tr>
<tr>
<td>twitter.com</td>
<td>67</td>
<td>58.3%</td>
</tr>
<tr>
<td>pinterest.com</td>
<td>84</td>
<td>57.0%</td>
</tr>
<tr>
<td>facebook.com</td>
<td>178</td>
<td>83.3%</td>
</tr>
<tr>
<td>google.com (yes, that simple noise-free page)</td>
<td>31</td>
<td>26.7%</td>
</tr>
<tr>
<td>google.com search for &quot;http requests per page&quot;</td>
<td>76</td>
<td>53.4%</td>
</tr>
</tbody>
</table>
Architecting for DevOps

**Traceability**
- Git SHA
- Package Coordinates
- API Version

**Quantitative Measurements**
- Business Metrics
- Tech Metrics

**Service Telemetry**

**Health**
- Me
- My Dependencies

**Qualitative Observations**
- Logging
- Exception Reporting
Architecting for DevOps

**Circuit Breakers**
- Request Rate
- Response Latency
- Open/Closed?

**Messaging**
- MoM Metrics
- Data Flow Metrics

**Integration Telemetry**

**Distributed Tracing**
- Trace Latency
- Span Latency
- Metatagging

**Correlated Logging**
- Event Tracing
- Retry Analysis
This is an Architectural Responsibility

- Architecture can make observability harder!
- Overhead Concerns
- Tools don't know your business.
NOTE: Architecting for DevOps aids in Continuous Delivery!
Architecting for Continuous Delivery

Adding some layers...
Think About Full Lifecycle Architecture
Architecture is abstract until it is operationalized.

Neal Ford
Architectures that aren’t operationalized exist only on whiteboards!
• Deployability
• Testability

We'll examine these qualities by asking questions of our architectures.
Deployability
Have you automated **ALL** of your deployment tasks?
Can you transform a brand new deployment environment into your running architecture without manual work?
Can you vary configuration across environments without rebuilding code?
Do you deploy like this EVERYWHERE?
Can you do this without your users noticing?
Testability
Have you automated **ALL** testing tasks that you possibly can?
Do you have to deploy all the things to test anything?
If testing is an experiment, can you control everything except your experimental variable?
Can you run the same tests against any environment (including production)?
Can you verify that you continue to meet your contractual obligations?
Cloud Capabilities

- API-driven
- Speed
- Elasticity
- Geography
- Specialized Services
Exploiting the capabilities of Cloud can enhance our ability to practice DevOps and Continuous Delivery!
• Disposability
• Replaceability

We'll examine these qualities by asking questions of our architectures.
Disposable

*adjective*

1. designed for or capable of being thrown away after being used or used up:
   
   *disposable plastic spoons; a disposable cigarette lighter.*

2. free for use; available:

   *Every disposable vehicle was sent.*

http://www.dictionary.com/browse/disposable
Replace

verb

1. to assume the former role, position, or function of; substitute for (a person or thing):
   *Electricity has replaced gas in lighting.*

2. to provide a substitute or equivalent in the place of:
   *to replace a broken dish.*

http://www.dictionary.com/browse/replace
Consequence

*noun*

1. an act or instance of following something as an effect, result, or outcome.

2. importance or significance:
   
a matter of no consequence.

http://www.dictionary.com/browse/consequence
Disposability

Can I destroy a service instance at any time without consequence?
Disposability

Can I repave the entire architecture at any time without consequence?
Disposability

Can I respond to changes in demand by adding or removing instances of a service without consequence?
Replaceability

Can I replace a sick service instance with a brand new copy without consequence?
Replaceability

Can I route traffic to any available service instance without consequence?
Replaceability

If I lose an AZ or Region, can I route traffic to another without consequence?
Replaceability

Can I swap between multiple implementations of the same service contract without consequence?
Replaceability

Can I swap between multiple running versions of a service without consequence?
These are Architectural Responsibilities

- Architecture can make disposability impossible.
- Architecture can make replaceability impossible.
- Architecture must take charge of removing the consequences of disposing and replacing service instances.
Architectural Decision Making Can:

- Enhance or Detract from Our Ability to Practice DevOps
- Enhance or Detract from Our Ability to Practice Continuous Delivery
- Exploit or Waste the Characteristics of Cloud Infrastructure
We could have called this "DevOps Native" or "Continuous Delivery Native" Architecture!
DevOps Native Architecture

Summary
Summary

Balancing:

Agility and Resilience
Summary

Supported By:

DevOps and Continuous Delivery
Summary

On a Foundation of:

Cloud and Architecture
Introduction to the Brick and Mortar Pattern Language
Patterns should...be described uniformly. This helps us to compare one pattern with another...

Pattern-Oriented Software Architecture, Volume 1: A System of Patterns
We describe design patterns using a consistent format...making design patterns easier to learn, compare, and use.

Design Patterns: Elements of Reusable Object-Oriented Software
Brick and Mortar Pattern Template

- **Context**
  
  *The basic situation in which we find ourselves working.*

- **Problem**
  
  *Presents the problem as a system forces which must be balanced.*

- **Solution**
  
  *Describes the components that make up the general solution, how they relate to one another, and their runtime interactions.*
Overview

Brick and Mortar Language Structure

• Brick Patterns
  Patterns for constructing individual (micro)services.

• Mortar Patterns
  Patterns for composing bricks into complete distributed systems.
Overview

Brick Patterns

• Externalization Patterns
  *Structural patterns for creating deployable, disposable, and replaceable bricks.*
  
  ▪ Externalized Configuration
  ▪ Externalized State
  ▪ Externalized Channels

• Runtime Patterns
  *Behavioral patterns for creating deployable, replaceable, and observable bricks.*
  
  ▪ Runtime Reconfiguration
  ▪ Concurrent Execution
  ▪ Brick Telemetry
Overview

Mortar Patterns

• Distributed Systems Patterns
  Composition patterns addressing common distributed systems challenges.
  - Service Discovery
  - Edge Gateway
  - Fault Tolerance

• Integration Patterns
  Composition patterns addressing integration and observability challenges.
  - Event-Driven System
  - Contract Management
  - Integration Telemetry
Overview

Brick and Mortar Language Relationships
Externalized Configuration
Context

An application's configuration will vary independently from its code throughout its lifecycle.
Problem

Traditional techniques for managing configuration tightly couple these two orthogonal concepts.
Forces

- Different environments will have different configuration settings:
  - resource handles to the database (e.g. a JDBC URL)
  - credentials to external services (e.g. Amazon S3)
  - per-deploy values such as the canonical hostname (e.g. blog-test.example.com vs. blog-prod.example.com)
  - features that are toggled on or off
Forces

- Configuration is often bundled within deployment artifacts (e.g. Java properties files).
- Build processes often modify configuration based on arguments.
- The Deployment Pipeline should only build each deployment artifact once, and deploy the same artifact to multiple environments.
Components

- The Environment
  *Sources of configuration name/value pairs*

- Environment Adapter
  *Component that understands a particular source of configuration and extracts name/value pairs from it*

- Configuration Repository
  *Component that manages the aggregate configuration of the brick.*

- Brick Component
  *Business component that is interested in configuration*
Solution Structure

Components and Relationships
The Environment Adapter

An Extensible Component

• Environment Variables
• Start Command Arguments
• Database
• Configuration Service
Solution Dynamics

Runtime Sequence of Events

Bootstrap → Environment Adapter
Load Config

Environment Adapter → Configuration Repository
Load from Environment

Configuration Repository → Business Component
Store Config

Business Component
Instantiate Component

Business Component
Retrieve Config
BREAK TIME
We will start again at 11:15!
Externalized State
Disposability and Replaceability require the elimination of "snowflake deployments" from the architecture.
Problem

*Traditional state management techniques prevent us from achieving "phoenix deployments."*
Forces

- Early web architectures emphasized server-side state management:
  - Fat Clients to Thin Clients
  - Vertically Scaled Cache Management
  - Stateful Scaffolding on Stateless Protocol (HTTP)
Context and Problem

Forces

- Cloud Infrastructure:
  - Resource Limited Horizontal Scale
  - Limited Load Balancer Support for Sessions
  - Limited (No) Support for Persistent Local Disk
Solution Structure

Single-Page Application Variant
Solution Structure

Externalized Session Variant
Solution Dynamics

Single-Page Application Variant
Solution Dynamics

Externalized Session Variant
Externalized Channels
Context

We want to be able to opportunistically evolve and recompose our systems as business drivers change.
Problem

Interleaving details of intercomponent communication with business logic makes it difficult to leverage functions/data streams in unplanned ways.
Forces

- Business processes and components are often reusable in other contexts.
- Business processes and components are independent of:
  - Wire Protocol
  - Serialization Method
  - Fault Tolerance Aspects
Forces

• Data streams can be "tapped" and redirected to other processes.
• Integration architectures are often "design once, change never."
• Business logic is often implemented in an integration architecture specific framework scaffold.
Solution Structure

Binder Variant (Spring Cloud Stream)

- **Transport [Component]**: Consumes/produces messages from native transport layer, serializes/deserializes into objects.
- **Input Channel [Component]**: Generic message producing channel.
- **Sink [Component]**: Consumes messages from Input Channels, sends them outside the messaging architecture.
- **Input Channel [Component]**: Generic message producing channel.
- **Processor [Component]**: Consumes messages from Input Channels, applies a transformation, sends transformed messages to Output Channels.
- **Output Channel [Component]**: Generic message consuming channel.
- **Output Channel [Component]**: Generic message consuming channel.
- **Source [Component]**: Consumes messages from outside the messaging architecture, sends them to an Output Channel.
- **Data Store [External System]**
- **IoT Device [External System]**

**Message Broker [External System]**

**Blue Components [Component]**: Represent a complete brick, and could be deployed independently as a container.
Solution Structure

Transport [Component]
Consumes/produces messages from native transport layer, serializes/deserializes into objects.

Inbox [Component]
Generic message producing channel.

Outbox [Component]
Generic message consuming channel.

Pattern Router [Component]
Routes messages to components based on matched patterns in message structure.

Business Object [Component]
Produces/consumes messages to/from Pattern Router.

Pattern Matching Variant (Seneca.js)

Message Broker [External System]

Blue Components [Component]
Represent a complete brick, and could be deployed independently as a container.

Messaging Application Container
Solution Dynamics

Binder Variant (Spring Cloud Stream)
Solution Dynamics

Pattern Matching Variant (Seneca.js)
Runtime Reconfiguration
Context

*We want to make configuration changes at runtime that are quick and don't otherwise disturb the state of the running system.*
Problem

Even with *Externalized Configuration*, configuration changes are often coupled to deployment events.
Forces

- Configuration often happens only as a boot time event.
- Environment Variables only injected as part of a deployment event.
- Debugging often implies "turning up the volume" on observability. Sometimes we need a "pre-warmed" system to listen to.
Forces

- Decoupling feature releases from deployment through feature toggles requires runtime configuration changes.
- Rotating DB credentials should be faster than (and not require) the deployment pipeline.
Config Server + Management Bus
Solution Dynamics

Config Server + Management Bus
Concurrent Execution
Context

*Users want business value (features), not deployments. Experiments often require concurrent execution of multiple software versions.*
Traditional techniques for managing deployments require downtime events and don't support concurrent execution of multiple software versions.
Forces

- Feature releases are coupled to deployment events.
- Production deployments will often target a single infrastructure environment.
- Downtime events expose users to "how the sausage is made."
Forces

- Failed deployments exacerbate downtime events.
- Hypothesis (in)validation (e.g. A/B testing) requires selective UX.
- Selective UX in a monolithic application is complex/error prone.
Solution Structure

Router + Optional Components
Solution Dynamics

Router + Service Registry
Brick Telemetry
Context

Realizing the DevOps Way of Feedback requires that we have visibility into both the business value and technical behavior generated by our services.
Problem

Common approaches to service visibility fall short of the architectural qualities that we need.
Forces

- Visibility is often accomplished via post facto application of agent-based monitoring tools.
- Agent-based monitoring tools don't understand business value.
- Determining an application's health often requires complex logic.
- Traceability of an application is difficult (or impossible) to accomplish with OTS solutions.
Solution Structure

Traceability Components

- **Traceability Endpoint [Component]**: Aggregates and reports on traceability data obtained from Traceability Adapters.
- **Git Traceability Adapter [Component]**: Reports Git commit data obtained from properties file.
- **Git Properties File [Data]**: Contains current Git commit and branch information for this Brick.
- **Maven Traceability Adapter [Component]**: Reports Maven coordinate data obtained from properties file.
- **Maven Properties File [Data]**: Contains current Maven group, artifact, and version information for this Brick.
- **API Version Traceability Adapter [Component]**: Reports currently enabled API version.
- **Feature Toggle Traceability Adapter [Component]**: Reports features currently toggled on/off.
- **Feature Toggle Framework [Component]**: Manages feature toggling for this Brick.
Solution Structure

Health Components

- **Health Endpoint [Component]**
  - Aggregates and reports on health data obtained from Health Indicators.

- **DB Health Indicator [Component]**
  - Reports health for a connected database.

- **Business Component A Health Indicator [Component]**
  - Reports on the health of a business component.

- **Disk Space Health Indicator [Component]**
  - Reports on available disk space.

- **Business Component B Health Indicator [Component]**
  - Reports on the health of a business component.

- **Database**
- **Local Disk**

**Brick**
Solution Structure

Metrics Components

Metrics Endpoint [Component]
Provides a point-in-time dump of current metrics via HTTP.

Metrics Registry [Component]
Manages the state and configuration of all metrics currently being recorded.

Metrics Reporter [Component]
Periodically dumps metrics data to an external data store.

Business Component A [Component]
Records timing data for an operation.

Business Component B [Component]
Records a histogram of sizes for an operation.

Business Component C [Component]
Reports the current value for a measurement.

Metrics Store [External System]

Query and Analyze Metrics

SRE
Solution Dynamics

Health
Socratic Q&A: Morning Topics
Service Discovery
Context

Decomposition of architecture into services leads to increasingly more distributed systems.
Problem

As systems become distributed, and as service instance lifecycles become more dynamic and independent, location of and communication with dependencies becomes more challenging.
Forces

- Cloud platforms often assign auto-generated, internal hostnames or private IP's to service instances.
- As services are scaled and unhealthy instances are replaced, the addresses of a service's instances are constantly changing.
- Binding a service to anything other than logical names for its dependencies leads to friction in the architectural lifecycle.
Forces

- Applying Concurrent Execution is made more difficult (or impossible) when binding services to fixed addresses for their dependencies.
- We may want to remove a service instance from the available pool but keep it running to troubleshoot a problem.
Solution Structure

Service Registry (Container)
Maintains mappings of logical service names to tables of addresses, ports, and other metadata.

Register service/refresh registration
Cache routing table locally

Service Registry Client (Component)
Registers service and periodically refreshes. Caches routing table locally.

Client-Side Load Balancer (Component)
Routes traffic to other Bricks using locally cached routing table.

Brick (Container)

Components Elided

Route traffic to other services

Horizontally Scaled Pool

Client-Side Load Balancing Variant
Solution Structure

Service Registry
[Container]
Maintains mappings of logical service names to tables of addresses, ports, and other metadata.

Register Brick/
refresh registration

Cache routing
table locally

Brick
[Container]
Obtain
Brick
Health
Make
remote
service
call
Horizontally
Scaled
Pool

Sidecar
[Container]
Proxy
remote service
call

Brick
[Container]
Obtain
Brick
Health

Sidecar
[Container]

Cache routing
table locally

Sidecar Proxy Variant
Client-Side Load Balancing Variant

Service Registry Client

Service Registry

Client-Side Load Balancer

Remote Brick

Register Brick in Service Registry

Cache Routing Table in Client

Update Load Balancer Pool from Local Routing Table Cache

Refresh Registration in Service Registry

Refresh Local Routing Table Cache

Register Brick in Service Registry

Make Remote Call
Solution Dynamics

Sidecar Proxy Variant
Architecture Katas Set Up
Architectural Katas

Based on http://archkatas.herokuapp.com

- You'll divide into groups and grab supplies.
- You'll take the architectural problem and create a practice architectural solution in ~1 hour.
- You can ask me any question you have about the project.
- Any technology stack is fair game.
- You may safely make assumptions about technologies you don't know well.
- You may not assume you have hiring/firing authority over the development team.
Architectural Katas
Making them Cloud Native

- As a company we're agreeing to guide ourselves by DevOps principles and practice Continuous Delivery.
- We have no infrastructure; we'll use one or more public cloud providers to deliver our software.
- Think deeply about your decomposition strategy and what advantages it will bring you.
- Use the CNA Patterns you know so far in order to enable your architectures to these ends.
Where's Fluffymon?

A service describing missing pets, pet rewards (brokered/managed by the service), and location data points (GPS) of pet sightings using augmented reality to overlay last-seen pet locations.

- **Users:** dozens of missing pet owners, hundreds of 'spotters' (initially), broader depending on rollout success
- **Requirements:**
  - users interested in finding pets register on the site
  - anyone can see a list of pets missing near to their location
  - pet finders can post 'pet found' messages (with mandatory photo proof) and collect rewards on confirmation from pet owners
  - users can comment on pet missing entries, offering data points (sighted, area checked with no results, etc)
  - mobile device accessibility
Where’s Fluffymoon?

A service describing missing pets, pet rewards (brokered/managed by the service), and location data points (GPS) of pet sightings using augmented reality to overlay last-seen pet locations.

- Additional Context:
  - one of a host of AR services being launched by parent company
  - local scalability (per-city), but possibly scaling out to other cities
  - company wants to create a larger social community around pets
  - potential ad revenue from partners like pet stores have the potential to make millions

CREDIT: http://nealford.com/katas/list.html
LUNCH

We will start the kata assignment promptly at 13:45!
Brick Pattern Assignment

In this kata, we will focus on the Brick Patterns. You will assume that you are the architect in charge of the image management part of the solution, and you will provide a Brick architecture for the Image Management service. Your solution should consider the cloud native characteristics that will help your team build and operate that service.
GO!

We will start peer review at 14:30!
Peer Review

• Each group will have 5 minutes to present.
• You'll be reviewed on your solution and your answers to questions.
• Review will be: Thumbs Up/Meh/Down!
• We'll carry on these activities until we run out of time.
Edge Gateway
Context

Decomposed architectures must always be recomposed. This recomposition often happens within the user interface layer of an application.
Recomposing an architecture within the User Interface layer presents significant complexities that can lead to decreased agility and degraded user experience.
Forces

- Systems often must support multiple user experience options (web/mobile/AVR).
- Recomposing architectures as the UI layer can require exposing the architecture to the public network.
- API needs for a mobile device are often quite different from a web UI.
Forces

- Exposing a network graph to mobile devices can increase latency, increase data usage, and degrade battery life.
- UI platforms may not support the integration architecture used for all services.
- Native apps often have longer upgrade cycles. Recomposing the architecture there can lead to friction in the architectural lifecycle.
Solution Structure

Mobile Phone Client [Container]
Web Browser Client [Container]
Voice Interaction Client [Container]

API Gateway [Container]
Routes, composes, and translates requests to Bricks.

Load Product Details
Call Catalog Service

Load Product Details
Call Reviews Service

Load Product Details
Call Recommendations Service

Catalog Brick [Container]
Reviews Brick [Container]
Recommendations Brick [Container]

API Gateway Variant
Solution Structure

Backend for Frontend (BFF) Variant

Mobile Phone Client [Container]

Load Product Details

Mobile BFF [Container]
Routes, composes, and translates requests to Bricks.

Call Catalog Service

Catalog Brick [Container]

Call Reviews Service

Reviews Brick [Container]

Call Recommendations Service

Recommendations Brick [Container]

Web Browser Client [Container]

Load Product Details

Web BFF [Container]
Routes, composes, and translates requests to Bricks.

Call Catalog Service

Catalog Brick [Container]

Call Reviews Service

Reviews Brick [Container]

Call Recommendations Service

Recommendations Brick [Container]

Voice Interaction Client [Container]

Load Product Details

Voice Interaction BFF [Container]
Routes, composes, and translates requests to Bricks.
Solution Dynamics

Edge Gateway (Both Variants)
Fault Tolerance
Context

In order to accomplish its assigned tasks, each brick will need to communicate with other bricks, and with external systems, to which we'll collectively refer as dependencies.
When a brick's dependencies become unhealthy, unreachable, or slower than normal to respond, that brick's own performance is degraded, and such degradation can potentially cascade across the entire architecture.
Context and Problem

Forces

- The network is not reliable.
- Latency is non-zero and unpredictable.
- Service availability is a product of its dependencies' availabilities.
Forces

- Failures can be transient.
- Failures can cascade.
- An incorrect or stale response is often preferable to no response.
Variants

- **Timeouts**
  Set time limits for operations that may never complete.

- **Retries**
  Retry failed operations in case of intermittent failure conditions.

- **Bulkheads**
  Isolate failure modes in different bricks or other isolation zones.

- **Circuit Breakers**
  Protect a brick from unhealthy dependencies and provide graceful degradation.
Solution Structure

**Business Component**
- Component
- Invokes remote dependency via Circuit Breaker.

**Circuit Breaker**
- Component
- Proxies usage of CSLB, detects unhealthy remote bricks, brokers fallback behavior.

**Client-Side Load Balancer**
- Component
- Routes traffic to other Bricks using locally cached routing table.

**Fallback Behavior**
- Component
- Provides graceful degradation of behavior in the event of unhealthy remote dependencies.

**Brick**
- Container
  - Route traffic to remote Brick.

Brick Container (Components Shown)

Circuit Breaker Variant
Solution Dynamics

Circuit Breaker Variant

Client-Side Load Balancer
[Component]
Routes traffic to other Bricks using locally cached routing table.

Remote Brick
[Container]

Business Component
[Component]
Invokes remote dependency via Circuit Breaker.

Circuit Breaker

Closed
Successful Request/Response
Trip Circuit After Failure Threshold Exceeded
Delegate to Fallback Component
Reset After Successful Request

Open
Successful Request/Response
Short-Circuited Request
Delegate to Fallback Component

Half Open
Tripped After Failed Request
Delegate to Fallback Component
Periodic Reset Attempt

Fallback Behavior
[Component]
Provides graceful degradation of behavior in the event of unhealthy remote dependencies.
BREAK TIME

We will start again at 15:30!
Event-Driven System
Context

Distributed systems often begin with a focus on synchronous, web-based interaction.
Problem

Synchronous web interactions and traditional approaches to state management often lead to intractable data management problems, especially in complex domains.
Forces

- Proper encapsulation of a distributed node includes ownership of its state. This leads to decomposed data stores, each governed by a single service.
- Many modern data stores sacrifice ACID transactions in order to support other desirable properties.
- Normalizing data to a single service can lead to network call proliferation for even simple operations.
Forces

- Business processes that desire transactionality but that include multiple distributed nodes can be challenging to implement.
- Queries that were previously implemented via SQL joins across multiple contexts now require multiple service calls and programmatic joins.
Solution Structure

API [Component]
Provides the client-facing API by interacting with the Command and Query Processors.

Command Processor [Component]
Executes commands and publishes the results as events.

Event Processor [Component]
Subscribes to events, processes them, and updates the materialized view.

Query Processor [Component]
Executes queries against the materialized view.

Event Store [Container]
Maintains an append-only, time-ordered log of all events published by the system.

Database [Container]
Maintains a queryable, materialized view of the current state of the system based on processed events.

Other Bounded Context [External System]
Subscribe to Events

Brick [Container]
Publish Events

Execute read/write operations.

Event Sourcing/CQRS Variant
Event Sourcing/CQRS Variant
Contract Management
Context

Decomposed architectures with decentralized governance still must maintain interface compatibility in order to communicate effectively.
Problem

As service teams move autonomously and evolve interfaces independently, incompatibilities can be introduced, often leading to recentralization of governance via complex versioning strategies.
Forces

• We want to support the autonomous evolution of service interfaces.
• Services must prove that they meet their existing contracts before they can deploy a new version of their interface.
• Not all upstream services are interested in a given service's new interface features and would prefer to keep the existing version.
Forces

• Maintaining multiple API versions within a single service is complex.
• Deprecating an existing API version is often political.
• Deploying "all the things" in order to perform integration testing across service boundaries is expensive.
Solution Structure

Consumer Brick
[Container]
Brick that consumes the contract published by the Provider.

Consumes the Mock Provider

Tests the Consumer via its API

Contract Test
[Container]
Test that specifies the contract expected from the provider.

Executes Provider Mock based on specified contract

Provider Mock
[Container]
Mock Brick created by the Contract Test, which behaves according to the expected contract.

Provider Brick
[Container]
Brick that should comply with the contract specified by the Consumer Contract Test.

Replays contract specification against Provider

Specification Player
[Container]
Replays contract specification against Provider and measures compliance.

Contract Specification
[Document]

Consumes contract specification document

Produces contract specification document

Consumer-Driven Contract Variant
Solution Dynamics

Consumer-Driven Contract Variant
Integration Telemetry
Context

Realizing the DevOps Way of Feedback requires that we have visibility into the runtime behavior emerging from our brick composition.
Problem

Distributed systems require additional types of visibility, and common approaches to visibility fall short.
Forces

- Distributed systems have emergent behavior.
- The edges in the graph are as (or more) important than the nodes.
- The health of two connected nodes does not tell you the health of the edge.
Forces

- Decentralized governance makes it difficult to know the complete dependency graph for request handling.
- It can be difficult to isolate and correlate the components of an end-to-end request flow.
Circuit Breaker Dashboard Variant
Solution Structure

Distributed Tracing Variant
Solution Dynamics

Circuit Breaker Dashboard Variant
Solution Dynamics

Distributed Tracing Variant
Mortar Pattern Assignment

In this kata, we will focus on the Mortar Patterns, but Brick Patterns are also fair game. We will now zoom out to the entire Where's Fluffymon? solution. You will assume that you are the architect in charge of the entire solution, and you will provide a complete system architecture. Your solution should consider the cloud native characteristics that will help all teams to build and operate the system.
GO!

We will start peer review at 16:45!
Peer Review

- Each group will have 5 minutes to present.
- You'll be reviewed on your solution and your answers to questions.
- Review will be: Thumbs Up/Meh/Down!
- We'll carry on these activities until we run out of time.
Thank You!

Cloud Native Architecture Patterns and Katas

Matt Stine (@mstine)

http://www.mattstine.com
GREAT INDIAN DEVELOPER SUMMIT 2019

Conference: April 23-26, Bangalore

Register early and get the best discounts!

www.developersummit.com  @greatindiandev  bit.ly/gidslinkedin