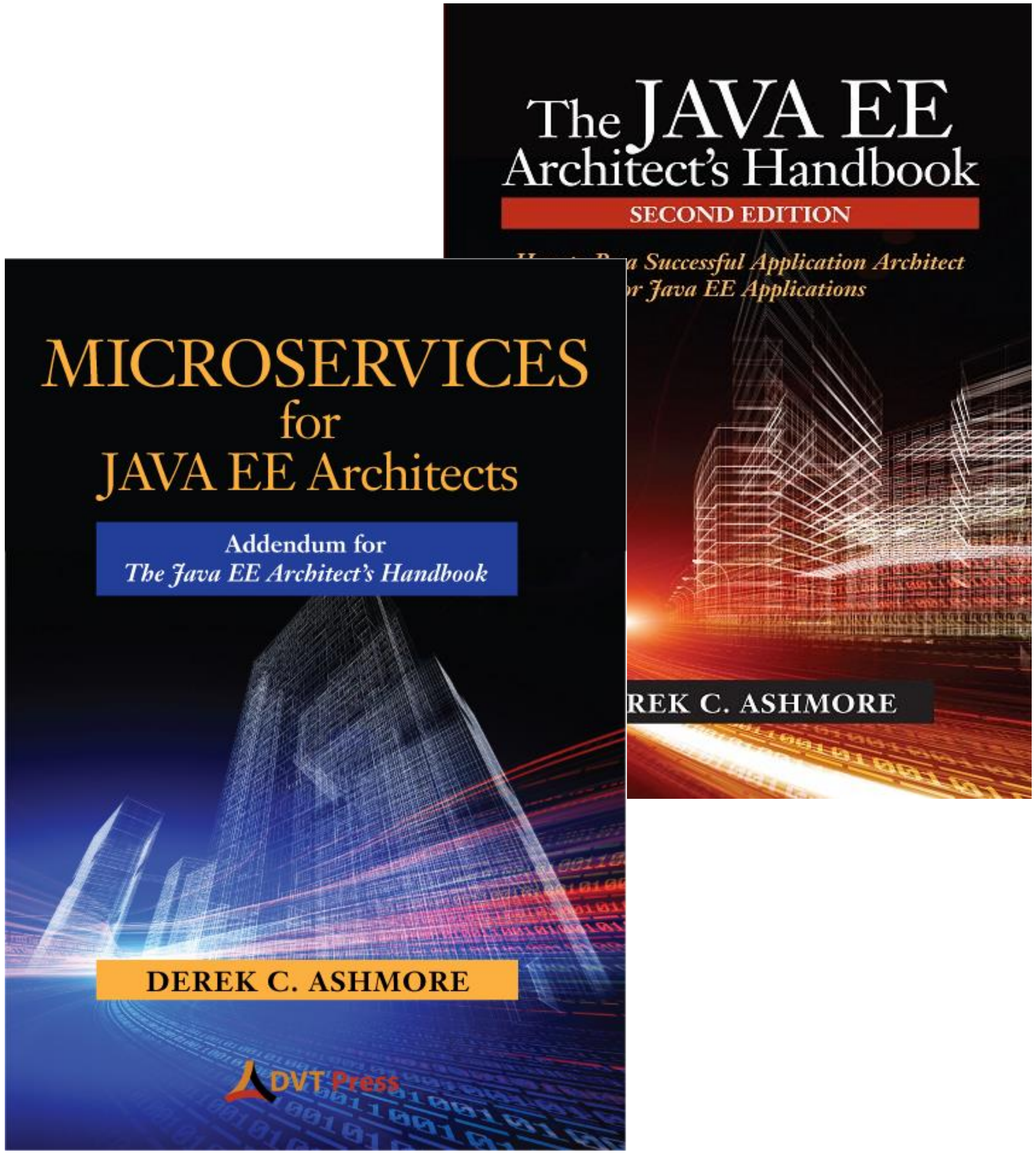
Writing Microservices in Java

Given by Derek C. Ashmore
October 28, 2015

STA GROUP

Who am I?

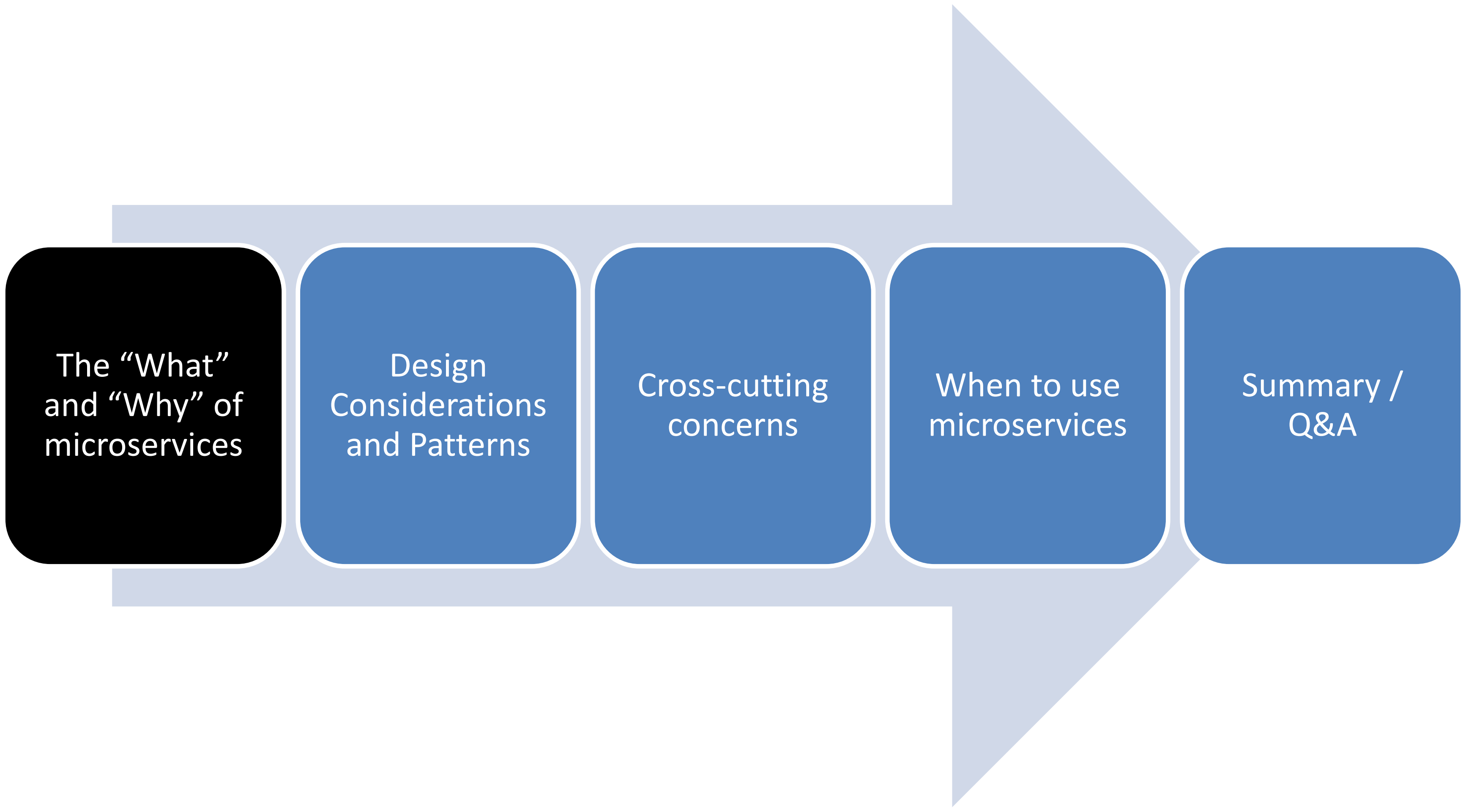
- Professional Geek since 1987
- Java/J2EE/Java EE since 1999
- Roles include:
    - Developer
    - Architect
    - Project Manager
    - DBA
    - System Admin


The JAVA EE Architect's Handbook
SECOND EDITION
a Successful Application Architect
r Java EE Applications
REK C. ASHMORE
MICROSERVICES for JAVA EE Architects
Addendum for The Java EE Architect's Handbook
DEREK C. ASHMORE
DVT Press

Discussion Resources

- This slide deck
  - http://www.slideshare.net/derekashmore
- Sample code on my Github
  - https://github.com/Derek-Ashmore/
- Sample Java Microservice (Moneta)
  - https://github.com/Derek-Ashmore/moneta
- Slide deck has hyper-links!
  - Don't bother writing down URLs

Agenda

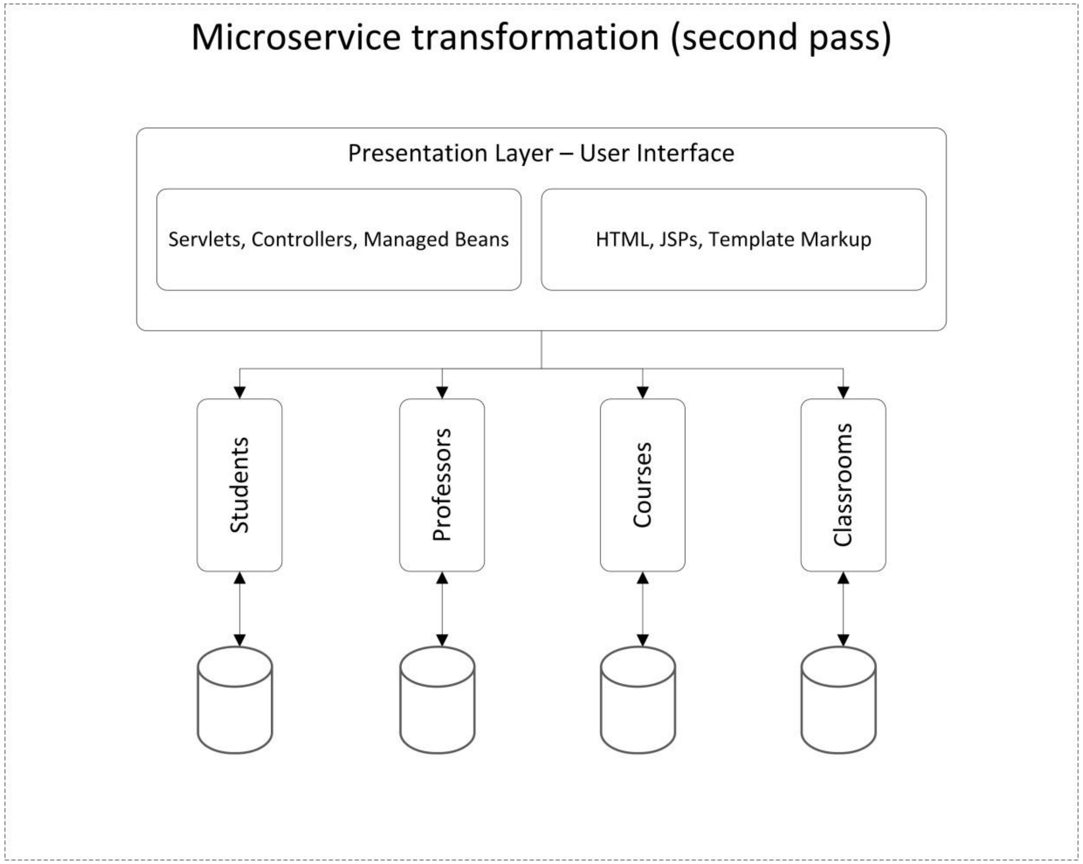

The "What" and "Why" of microservices
Design Considerations and Patterns
Cross-cutting concerns
When to use microservices
Summary / Q&A

What are Microservices?

- No concrete definition
- Common microservice traits
  - Single functional purpose
    - Most/all changes only impact one service
    - Not dependent on execution context
      - “loosely coupled”
  - Independent process/jvm
  - Standard Interface (typically Web Service/REST)
  - Analogy: Stereo system, Linux utilities

Refactoring into Microservices

- Databases physically separated
- What to do with common data needs?
  - Service call **<u>or</u>**
  - Data copy

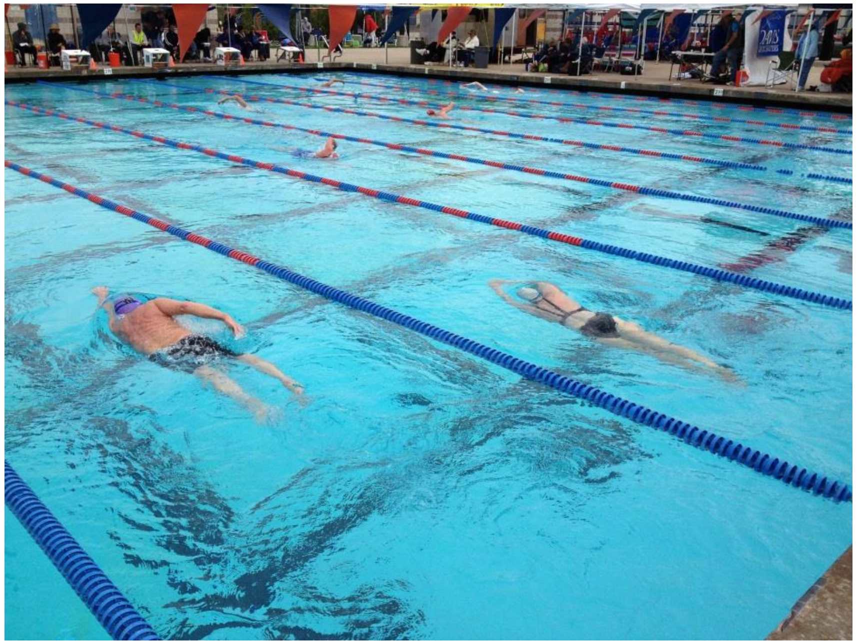

Microservice transformation (second pass)
Presentation Layer – User Interface
Servlets, Controllers, Managed Beans
HTML, JSPs, Template Markup
Students
Professors
Courses
Classrooms

No Lock-in

- Platform agnostic
- Fewer dependency conflicts
- Still have cross-cutting concerns
  - "Toll" for first app
- Still have support concerns
  - Others need to be able to support your work

Easier Management / Higher Throughput

- Easier to manage large numbers of developers
  - Concentrate on intelligently drawing service boundaries
  - Manage/enforce service contracts
- Each service team works independently
- Team independence leads to higher development throughput



Agenda

- The "What" and "Why" of microservices
- Design Considerations and Patterns
- Cross-cutting concerns
- When to use microservices
- Summary / Q&A

Design considerations

- Service Boundaries (gerrymandering)
- Service call Failure / Unavailability
- Data Integrity
- Performance

Service Boundaries

- Core Services
  - Services responsible for maintaining a specific business area data
  - Usually named after Nouns
    - Service is a system of record for a <blank>
      - Student, Course, Classroom, etc.
- Process Services
  - Services responsible for performing single complex tasks
  - Usually represents an Action or Process
    - Service is responsible for processing <blank>
      - Student applications, Debt collection, etc.
  - These services rely on core services
- Partitioning is an art
  - Too few → same drawbacks as traditional architecture
  - Too many → excessive network hops

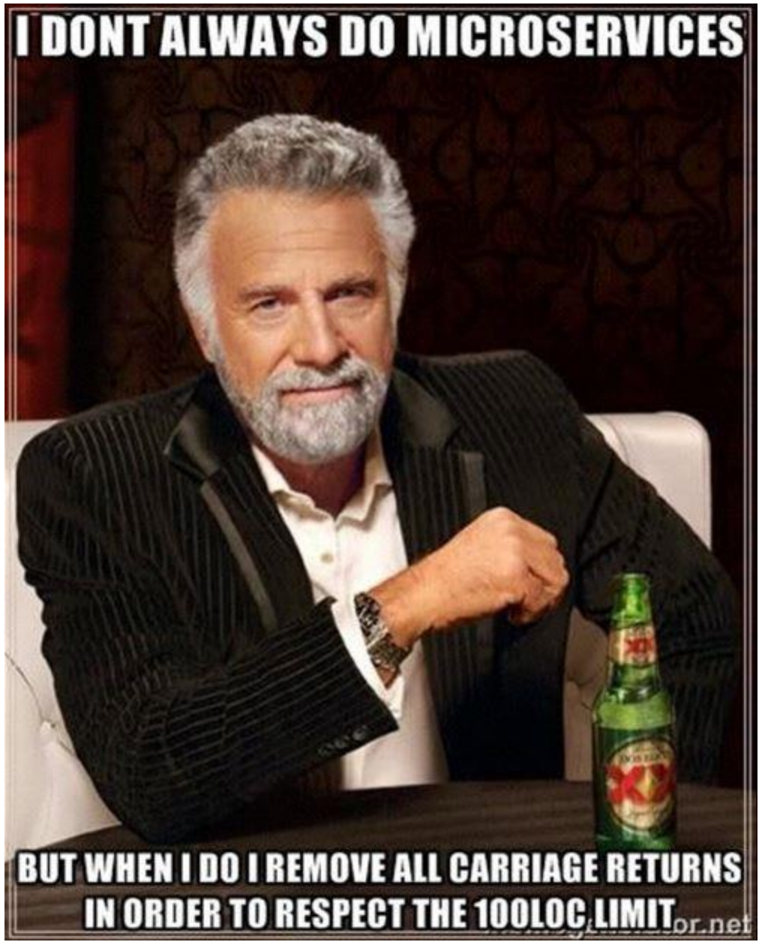
Boundary Sanity Check

- Verbalize a mission statement in ***<u>one</u>*** sentence in business terms
  - Examples
    - This service is the system of record for Student information
    - This service registers students for classes
    - This service suspends students
    - This service records student payments
    - This service produces official transcripts

Context Independence Check

- Does your service have multiple consumers?
  - Could it?
- Could your service execute as easily in batch as online?
  - If ‘No’, then you’re making context assumptions
- Warning Signs
  - Spending time analyzing service call flow
    - Your services likely make context assumptions
  - Agonizing over which service should do a given activity
    - Maybe you need a new service

Microservices are not about size

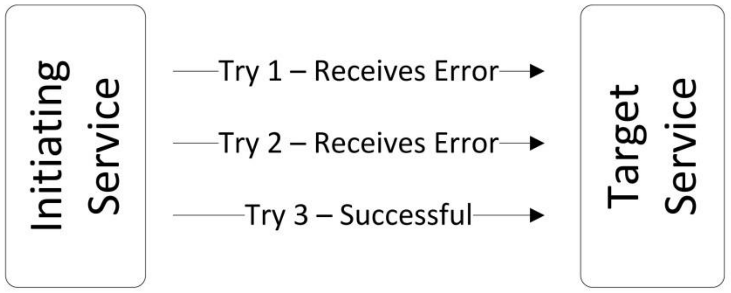

I DONT ALWAYS DO MICROSERVICES
BUT WHEN I DO I REMOVE ALL CARRIAGE RETURNS
IN ORDER TO RESPECT THE 100LOC LIMIT

….. Microservices are about having a single business purpose!

Designing for Failure

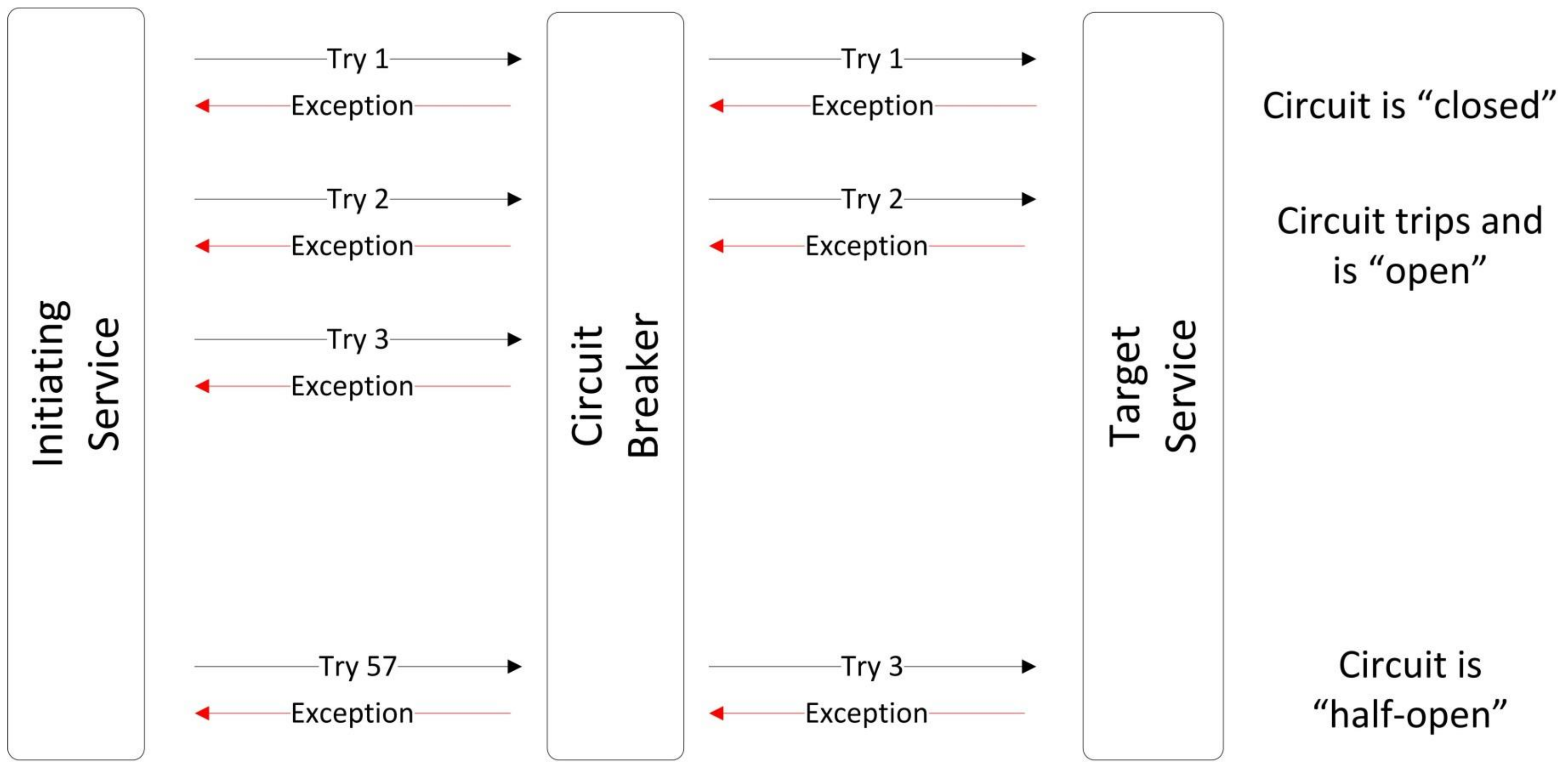
- Dependent services could be down
  - Minimize human intervention
  - Fail sooner rather than later
  - Horizontal scaling / clustering is your first line of defense
  - Coding patterns can help as a backup
- Common Patterns:
  - Retry
  - Circuit Breaker
  - Dispatch via Messaging
  - Service Call Mediator

Retry Pattern



- Best for asynchronous tasks
- Limit the number of tries
- Use sleep interval between tries
- Only addresses temporary outages
- Sample Retry Pattern implementation here.
- Tooling Support:
  - Apache CXF supports Retry
  - Spring Batch RetryTemplate
  - Apache HttpClient (Example here)

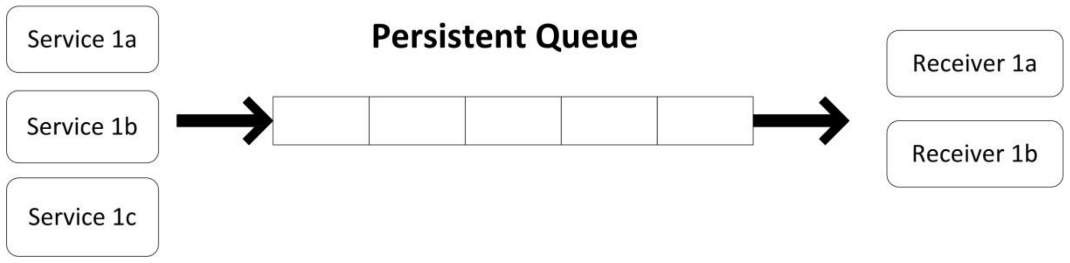
Circuit Breaker



Circuit Breaker (continued)

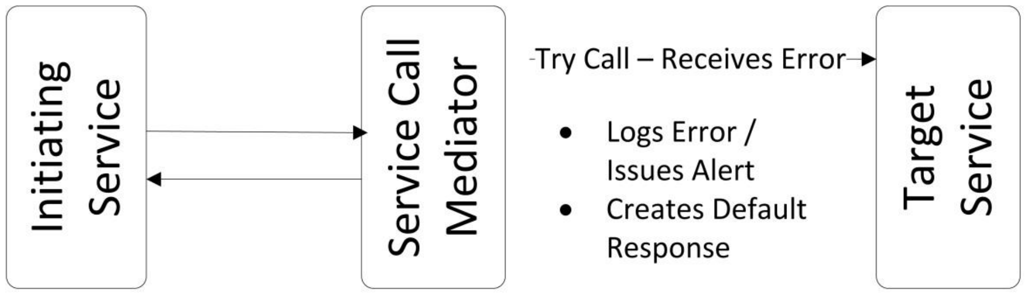
- Objective: Error out sooner
  - Conserves resources
  - Automatically "recovers" after a time period
- Modeled after home circuit
- Works on thresholds
  - Number of errors required to trip circuit
  - Amount of time required to attempt retry
- Has Hysterix support
- Best embedded in interface clients / delegates
- More information here.
- Sample Circuit implementation here.

Dispatch via Messaging



- Place work instruction on persistent queue
- If receivers are down, work stacks in queue
- Work throttled by number of receivers
- Queue can be JMS or AMQP
- Tooling Support:
  - JMS Api (easy API – many use natively)
  - Spring JMSTemplate or RabbitTemplate (producer)

Service Call Mediator

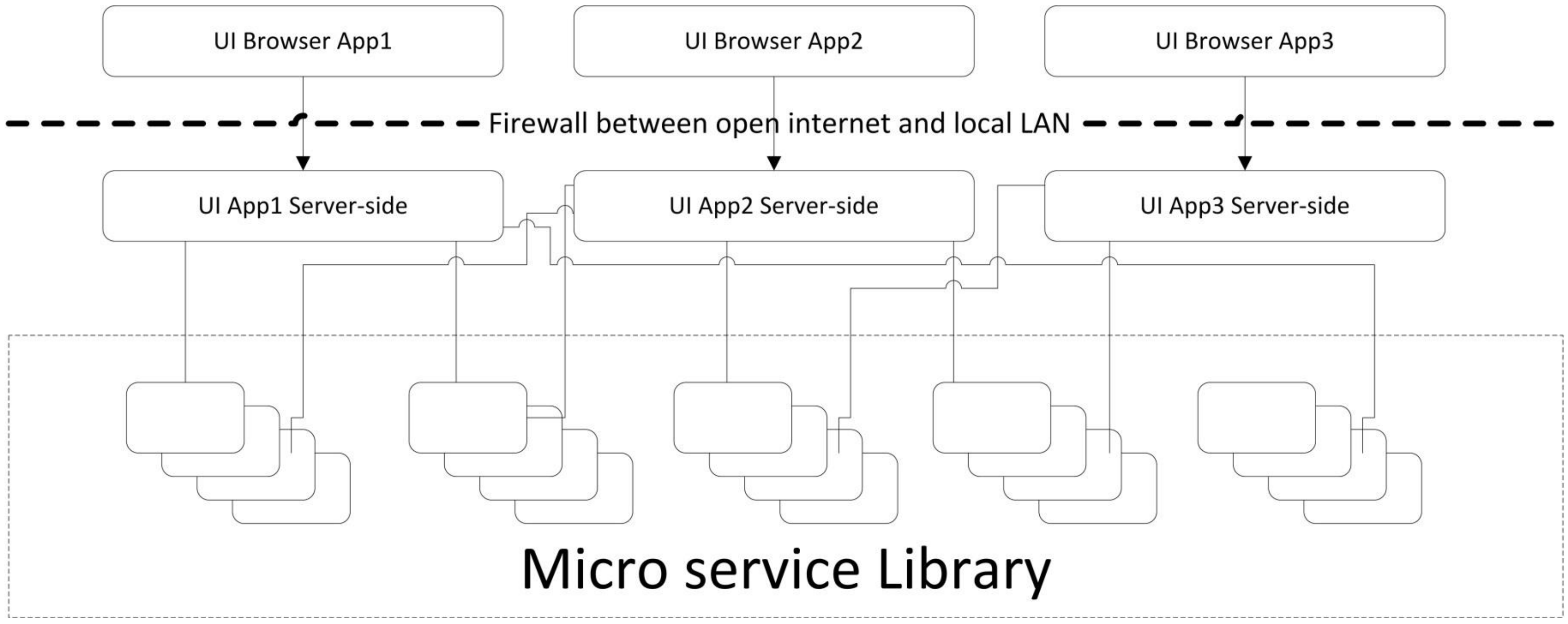

Initiating Service
Service Call Mediator
Try Call – Receives Error
• Logs Error / Issues Alert
• Creates Default Response
Target Service

- Provide "Partial" functionality when dependent services are down
- Providing partial functionality better user experience than complete outage
  - Airline Wifi provider providing service even if payment processing is down
- Sample implementation here

Designing for Performance

- More network traffic
  - Make services course-grained
  - User Interfaces need a general manager
  - Horizontal or Vertical Scaling helps
- Common Patterns:
  - Back-ends for Front-ends (a.k.a. API Gateway)
  - Dispatch via Messaging
  - Expiring Cache

Back-ends for Front-ends

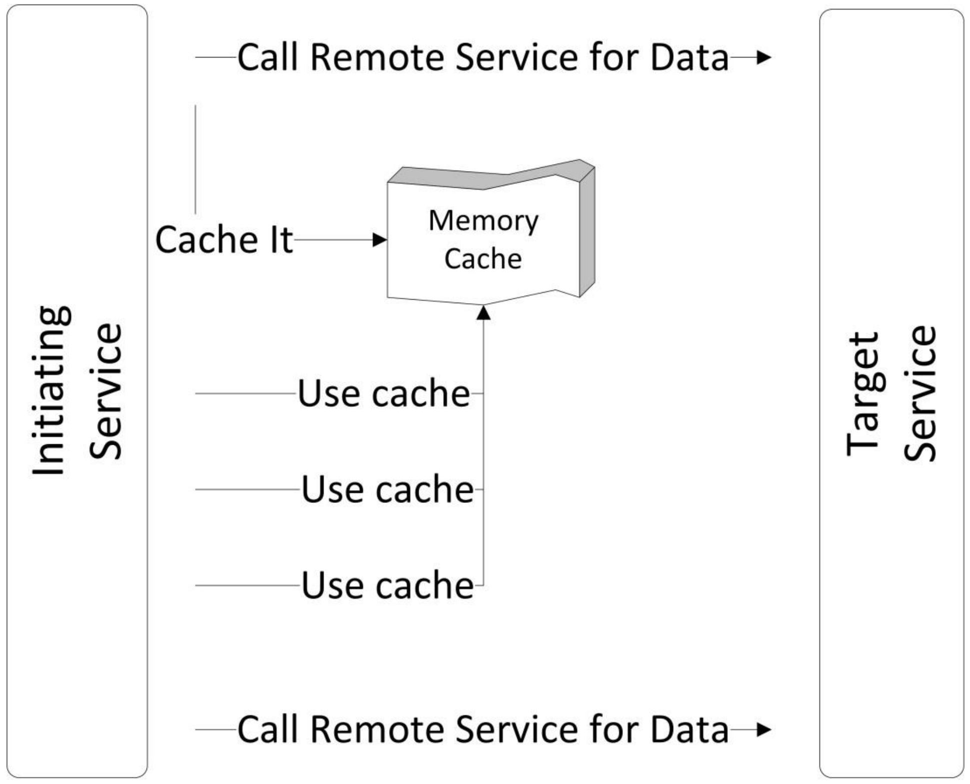


Back-ends for Front-ends

(continued)

- Consolidates service calls for the browser
  - Enhances performance
    - Open web often not as performant as local LAN
- Also known as “API Gateway”
- Implications
  - Don’t expose microservices directly to the browser

Expiring Cache



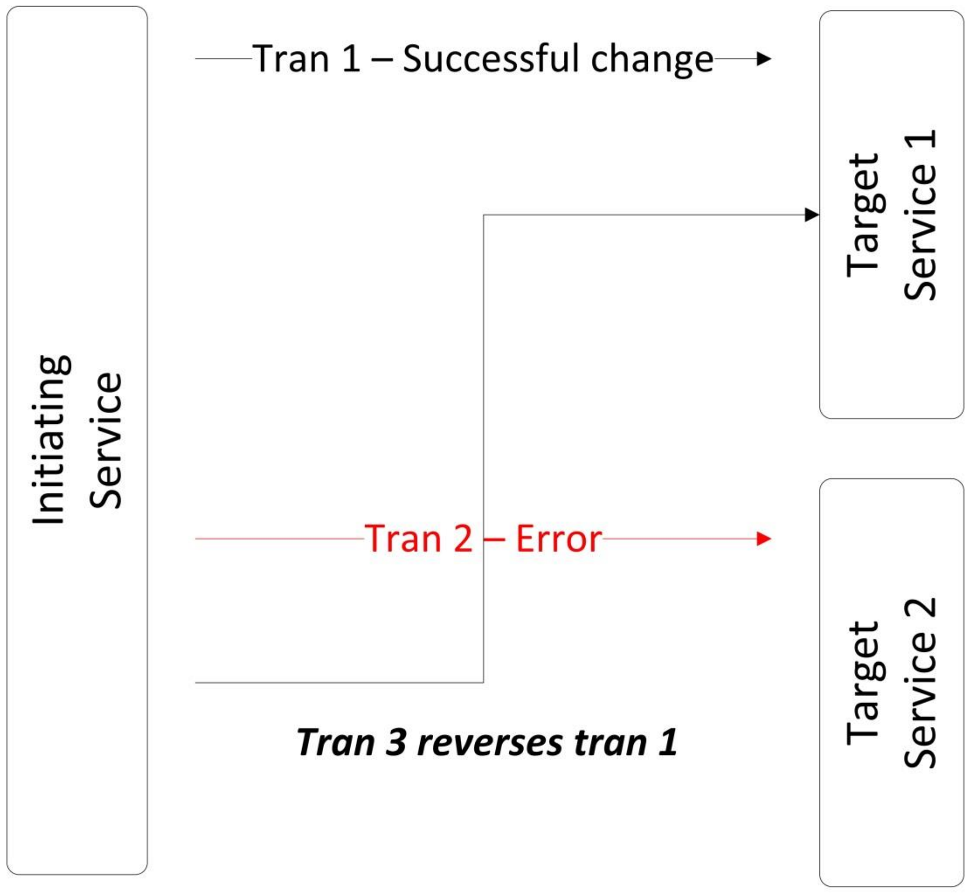
Expiring Cache (continued)

- Look up data once and cache it
  - Evict data from the cache after a defined time period
  - Sometimes known as “Cache Aside”
  - Reduces network calls for data
  - Trades memory for speed
  - More information here
- When to use
  - Only use with static data
  - Different clustered nodes “could” have different data for a short time
- Tooling Support:
  - I recommend Google Guava
  - EHCache, Gemfire, and other tools available

Designing for Integrity

- Services are context independent
  - Have no knowledge of how/when they are executed
- One service == One Transaction
  - Two-phase commits/rollbacks are a much larger problem
- Common Patterns:
  - Custom Rollback
    - Write your own reversing transaction

Custom Rollback



Custom Rollback (continued)

- Reverses a transaction previously posted
- Only use this for multi-service transactions
  - Keeping the transaction within one service is preferred
- This pattern is completely custom
  - No special product support available
- More information here

Common code between services?

- Yes, but….
  - Version it; services make decision as to when to upgrade
  - Changes to common code **<u>can’t</u>** require the deployment of multiple services
    - That ‘common code’ needs to be its own separate service
    - Tends *not* to have business logic as that can change and impact multiple services

Agenda

- The “What” and “Why” of microservices
- Design Considerations and Patterns
- Cross-cutting concerns
- When to use microservices
- Summary / Q&A

Cross-cutting Concerns

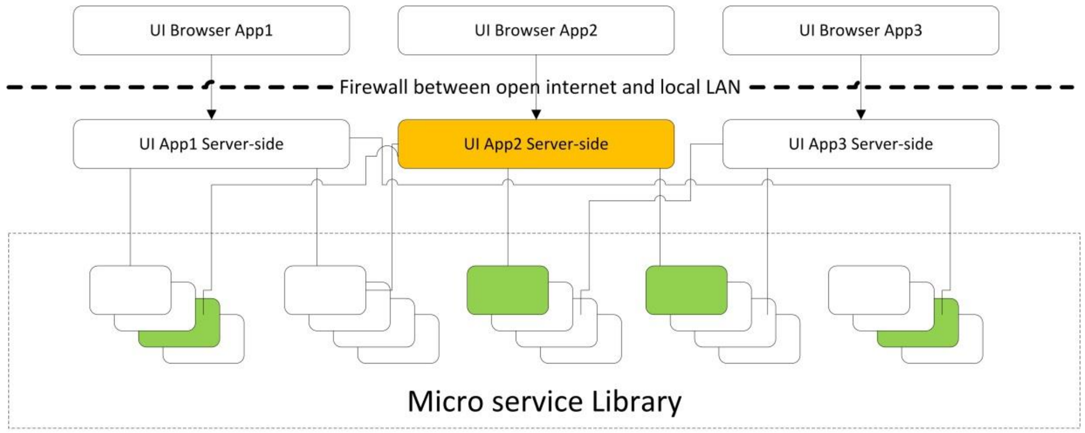
- Deployment
- Transaction tracking
- Security
- Contract Testing
- Same as traditional applications
  - Health checks
  - Logging consolidation
  - Performance measurement

Deployment

- Microservices are deployed as a process
  - For Java, embedded containers are easy
  - Spring Boot
  - Dropwizard
- Docker – standardizes the process deployment and environment
- Sample here.

Correlation IDs

- Provides context for service calls or user actions
- Track using HTTP Header
- Log it on all messages / error reports
- Include it on all service calls or message dispatches
- Code sample here
- Spring Boot support has been requested

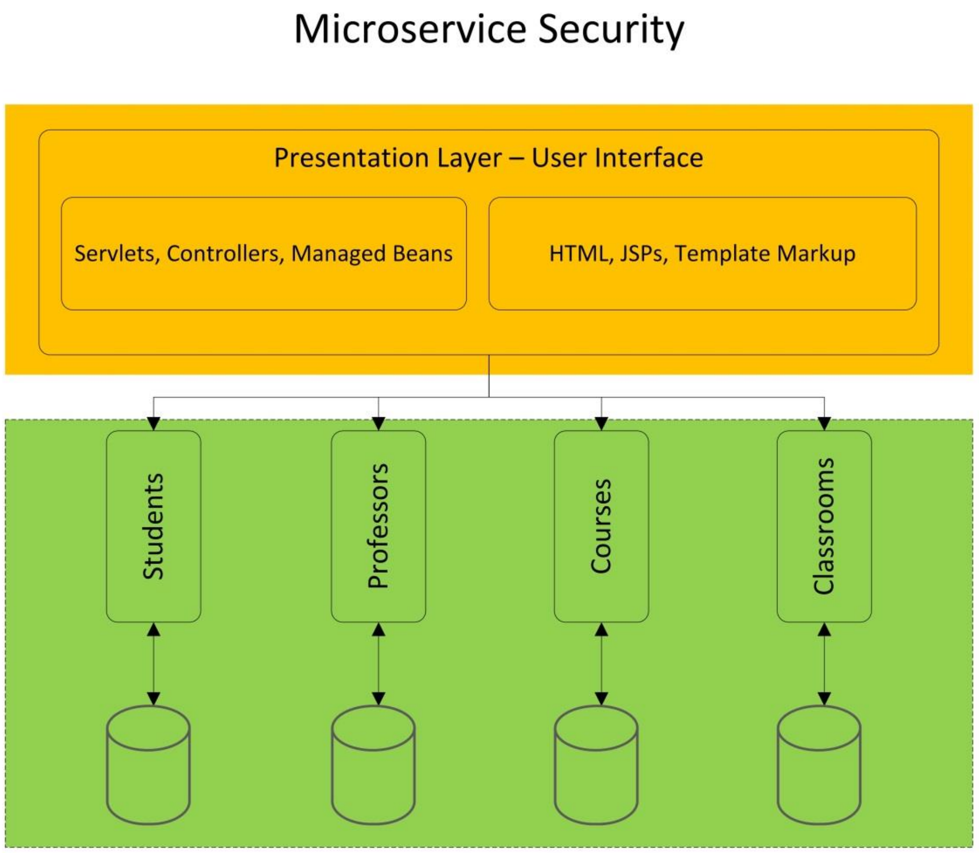


Correlation IDs help you track a group of related micro-service transactions!

Security

User-Level Security

Network-Level and/or Service-Level Security


Microservice Security
Presentation Layer – User Interface
Servlets, Controllers, Managed Beans
HTML, JSPs, Template Markup
Students
Professors
Courses
Classrooms

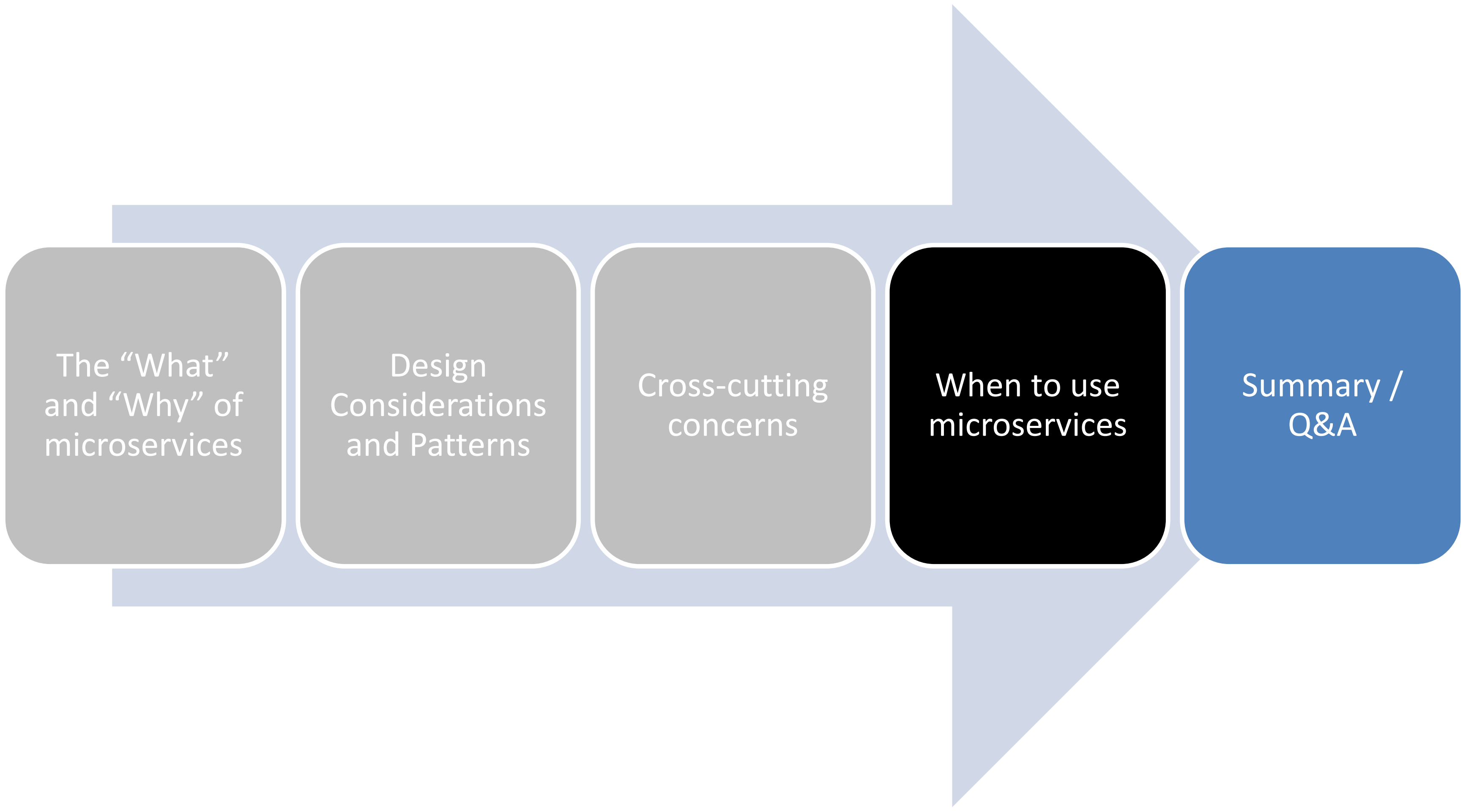
Security (continued)

- Keep User-level security to the UI
- Microservice security in layers
  - Layer 1 – Network routing enforcement
    - Limit access only to within the firewall
    - Limit access to specific hosts or subnets
  - Layer 2 – Use Service Accounts
    - Similar to database access

Contract Testing

- Critical for MS architectures
  - Contract changes can break other services
  - Bulkhead for rogue developers
  - Makes individual services more disposable
- Consumer-based testing
- Tooling support
  - Apache HttpClient
  - SoapUI
  - ActiveMQ for JMS (embedded broker)

Agenda


The "What" and "Why" of microservices
Design Considerations and Patterns
Cross-cutting concerns
When to use microservices
Summary / Q&A

When to consider MS

- Starting out with MS isn't recommended unless
  - Parts of the application will have extremely high volume
    - Need to scale a portion of the application differently
    - ***Note: MS isn't all or nothing!***
- Warning signs for app that's too large
  - Unintended consequences after release
  - High technical debt / design rot
  - Release testing cycles abnormally large
  - Need to coordinate large numbers of developers for a single code base
    - Large number == takes more than two pizzas to feed

Common Mistakes

- Inappropriate Service Boundries
  - Services that are not truly loosely coupled
    - One change → Multiple services deployed
  - Services that make 'assumptions' about execution context
    - Deployments cause unintended consequences
- Not recording all requests/responses
  - Support developers need to localize problems
  - Include request/response data in exceptions
    - Contexted Exceptions in Commons Lang

Common Mistakes (continued)

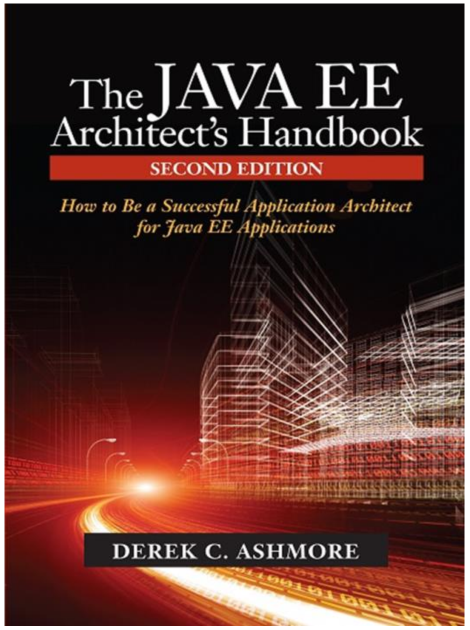
- Not checking arguments up front
  - Derivative exceptions take longer to debug/fix
  - `NullPointerException` == Argument not checked!
- No Change in Governance
  - Easier / quicker path to production
  - Automated Deployments/Backouts
    - Less manual intervention
    - Less manual testing (quick reaction vs prevention)
  - Continuous Delivery / DevOps / Microservices go hand-in-hand

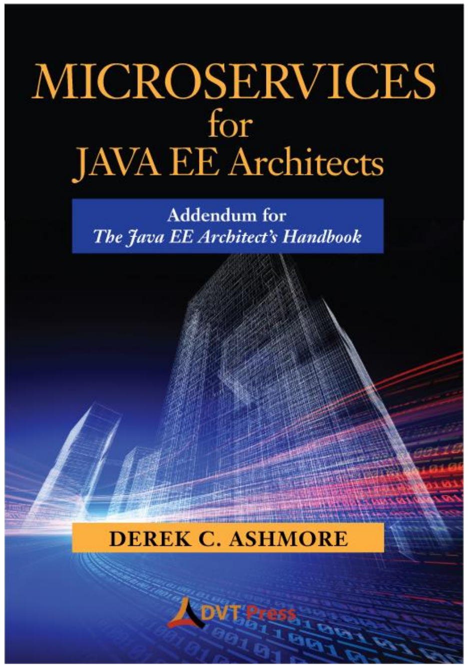
Further Reading

- Microservices reading list
  - http://www.mattstine.com/microservices
- Microsoft's Cloud Design Patterns
  - https://msdn.microsoft.com/en-us/library/dn600223.aspx
- Moneta Java microservice example
  - https://github.com/Derek-Ashmore/moneta
- This slide deck
  - http://www.slideshare.net/derekashmore

Questions?

- Derek Ashmore:
  - Blog: www.derekashmore.com
  - LinkedIn: www.linkedin.com/in/derekashmore
  - Twitter: https://twitter.com/Derek_Ashmore
  - GitHub: https://github.com/Derek-Ashmore
  - Book: http://dvtpress.com/


The JAVA EE Architect's Handbook
SECOND EDITION
How to Be a Successful Application Architect for Java EE Applications
DEREK C. ASHMORE


MICROSERVICES for JAVA EE Architects
Addendum for The Java EE Architect's Handbook
DEREK C. ASHMORE
DVT Press

STA GROUP