Software Architecture Fundamentals Workshop

Part 2: A Deeper Dive



Mark Richards **Independent Consultant** Hands-on Enterprise / Integration Architect Published Author / Conference Speaker

http://www.wmrichards.com http://www.linkedin.com/pub/mark-richards/0/121/5b9



O'REILLY SOFTWARE ARCHITECTURE SERIES Software Architecture Fundamentals:

Part 1:

the Basics

Understanding Neal Ford. Mark Richards VIDEO



O'REILLY" SOFTWARE ARCHITECTURE SERIES Software Architecture Fundamentals Part 3 Soft Skills: Problem Solving, Decision Making, Refactoring, Productivity, & Communication Neal Ford. Mark Richards VIDEO



O'REILLY"
SOFTWARE ARCHITECTURE SERIE
Software Architecture

ThoughtWorks[®]

NEAL FORD

Director / Software Architect / Meme Wrangler

Fundamentals Service-Based Architectures Structure, Engineering Practices, and Migration Neal Ford, Mark Richards

VIDEO





••• <> . . nealford co A Ø Jnealford.com Writing Presentations & Abstracts Biography Architectural Katas inspired by Ted Neward's original Architectural Katas "How do we get great designers? Great designers design, of course." Fred Brooks "So how are we supposed to get great architects, if they only get the chance to architect fewer than a half-dozen times in their career? Ted Neward About Architectural Katas are intended as a small-group (3-5 people) exercise, usually as part of a larger group (4-10 groups are ideal), each of whom is doing a different kata. A Moderator keeps track of time, assigns Katas (or allows this website to choose one randomiy), and acts as the facilitator for the servise. The Architectural Katas started as a presentation workshop by Ted Neward. They've taken on a life of their own. Learn more -Rules Doing an Architectural Kata requires you to obey a few rules in order to get the maximum out of the activity. Read Rules * Lead Want to run the Architectural Katas yourself? There's only a few things you need to know before you do. Ted Neward, the originator of Architectural Katas, has information on his site about leading Katas exercises. Read on the original site = List Katas » Random Kata » Follow Neal on Twitter at @neal4d Neal works at ThoughtWorks, a very interesting place. Neal speaks frequently on the No Fluff, Just Stuff conference circuit. Meme Agora RSS feed. nealford.com/katas/

Software architecture reflects the mapping between capabilities and constraints.











"our business is constantly changing to meet new demands of the marketplace"





"due to new regulatory requirements, it is imperative that we complete endof-day processing in time"





"we need faster time to market to remain competitive"





"our plan is to engage heavily in mergers and acquisitions in the next three years"





"we have a very tight timeframe and budget for this project"





architecture pitfall



armchair architecture

whiteboard sketches are handed off as final architecture standards without proving out the design



armchair architecture

occurs when you have noncoding architects

occurs when architects are not involved in the full project lifecycle

occurs when architects don't know what they are doing



architecting for change



architecting for change

business is in a constant state of change

increased competition



regulatory changes

mergers

growth

acquisitions

architecting for change

technology is in a constant state of change



frameworks

techniques for change





reduce dependencies





create domain-specific architectures

create product-agnostic architectures

reduce dependencies



less da preintertiserale geveloke de (inedle que patiene dy les)

reduce dependencies



messaging service bus adapters architecture patterns

leverage standards

industry standards







<?xml?>



Financial products Markup Language

leverage standards

corporate standards







ORACLE



leverage standards

standards may not always be your first choice, but they significantly help in reducing the effort for change

larger resource pool

better integration with other systems

product-agnostic architecture isolate products to avoid vendor lock-in



domain-specific architecture

generic architectures are difficult to change because they are too broad and take into account scenarios that aren't used

limit the scope of the architecture by taking into account drivers, requirements, business direction, and industry trends

domain-specific architecture



Architecture Anti-pattern



...but further on leads you into a maze filled with monsters

vendor king

product-dependent architectures leading to a loss of control of architecture and development costs



vendor king



vendor king





(More) Architecture Patterns



space-based architecture



microservices architecture



service-based architectures









let's talk about scalability for a moment...





processing unit



middleware



middleware






space-based architecture

middleware

manages dynamic processing unit deployment





space-based architecture

it's all about variable scalability...

good for applications that have variable load or inconsistent peak times



not a good fit for traditional large-scale relational database systems

relatively complex and expensive pattern to implement



business services

BS BS BS

BS BS

BS

abstract enterprise-level coarse-grained services owned and defined by business users

no implementation - only name, input, and output data represented as wsdl, bpel, xml, etc.

ExecuteTrade

PlaceOrder

ProcessClaim

concrete enterprise-level coarse-grained services owned by shared services teams

custom or vendor implementations that are one-to-one or one-to-many relationship with business services



concrete application-level fine-grained services owned by application teams

bound to a specific application context



application services AS

concrete enterprise-level fine-grained services owned by infrastructure or shared services teams

implements non-business functionality to support both enterprise and business services

WriteAudit

CheckUserAccess

LogError





message enhancement message transformation protocol transformation

mediation and routing process choreography service orchestration





separately deployed components



service component



bounded context



service orchestration



service orchestration



CQRS



http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/





CORS Command Query Responsibility Separation

CQRS natural fits

task-based user interface

meshes well with event sourcing

eventual consistency

eventual consistency

O
O
Eventually Consistent - Revisited - All Things Distributed

Eventually Consistent - Revisited - All Things Distributed

With the second se

C Reader

All Things Distributed

Werner Vogels' weblog on building scalable and robust distributed systems.

Eventually Consistent - Revisited

By Werner Vogels on 22 December 2008 04:15 PM | Permalink | Comments (14)

I wrote a <u>first version of this posting</u> on consistency models about a year ago, but I was never happy with it as it was written in haste and the topic is important enough to receive a more thorough treatment. <u>ACM</u> <u>Queue</u> asked me to revise it for use in their magazine and I took the opportunity to improve the article. This is that new version.

Eventually Consistent - Building reliable distributed systems at a worldwide scale demands tradeoffs between consistency and availability.

At the foundation of Amazon's cloud computing are infrastructure services such as Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and need to be accounted for up front in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customers of these services will be confronted with the consequences of using replication techniques inside the services.

One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when the underlying distributed system provides an eventual consistency model for data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. In this article I present some of the relevant background that has informed our



Contact Info Werner Vogels CTO - Amazon.com

werner@allthingsdistributed.com

Other places

Follow werner on <u>twitter</u> if you want to know what he is current reading or thinking about. At <u>werner.ly</u> he posts material that doesn't belong on this blog or on twitter.

Syndication

Subscribe to this weblog's atom feed or rss feed "Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability."

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

CQRS natural fits

task-based user interface

meshes well with event sourcing

eventual consistency

consistency or availability (but never both)

complex or granular domains



LMAX http://martinfowler.com/articles/lmax.html

JVM-based retail financial trading platform

centers on Business Logic Processor handling 6,000,000 orders/sec on 1 thread

surrounded by Disruptors, network of lock-less queues

overall structure





business logic processor

in-memory

event sourcing via input disruptor

snapshots (full restart—JVM + snapshots — less than 1 min)

multiple instances running

each event processed by multiple processors but only one result used

input/output disruptors



disruptors

custom concurrency component

multi-cast graph of queues where producers enqueue objects and consumers dequeue in parallel

ring buffer with sequence counters





"mechanical sympathy"

started with transactions

switched to Actor-based concurrency

hypothesized & measured results

CPU caching is key **single** writer principle

"One ring to rule them all..."

no architecture fits every circumstance

evaluate good, bad, and ugly

watch for primrose paths that turn ugly

embrace pragmatism

what's your day job?

building wicked cool architectures is an stimulating, rewarding intellectual challenge...

...building CRUD applications isn't

coolness sometimes equals accidental complexity

architecture pitfall



spider web architecture

creating large numbers of web services that are never used just because you can


spider web architecture



spider web architecture

just because you can create a web service at the click of a button doesn't mean you should!

let the requirements and business needs drive what services should be exposed





Meeting Hacks



makers vs. managers

Δ

ŋ

 $\Theta \gg$

Ċ

MAKER'S SCHEDULE, MANAGER'S SCHEDULE

Want to start a startup? Get funded by Y Combinator.

July 2009

=

 (\mathbf{i})

One reason programmers dislike meetings so much is that they on a different type of schedule from other people. Meetings cos. them more.

paulgraham.com

There are two types of schedule, which I'll call the manager's schedule and the maker's schedule. The manager's schedule is to bosses. It's embodied in the traditional appointment book, with each day cut into one hour intervals. You can block off several hours for a single task if you need to, but by default you change what you're doing every hour.

When you use time that way, it's merely a practical problem to meet with someone. Find an open slot in your schedule, book them, and you're done.

Most powerful people are on the manager's schedule. It's the schedule of command. But there's another way of using time that's common among people who make things, like programm and writers. They generally prefer to use time in units of half a day at least. You can't write or program well in units of an hour Since most powerful people operate on the manager's schedule, they're in a position to make everyone resonate at their frequency if they want to. But the smarter ones restrain themselves, if they know that some of the people working for them need long chunks of time to work in.

www.paulgraham.com/makersschedule.html





take one (or many) for the team



know the dramatis personæ

don't ask important questions you don't already know the answer to

meeting imagery



don't improvise white board drawings in important meetings





is this more important than the work you are pulling people away from?









- —keep it short
- -keep it relevant
- -everyone
 - -too many?
 - -geographically dispersed?
- -police inappropriate levels of interaction

dev-huddle

Architecture Refactoring Techniques



architecture refactoring



determine the

architectural root

cause of each issue





determine what architecture changes are needed



create a business case justifying the changes





present your case and plan to the business for approval and funding



create a timeline containing estimates and resources

develop a high-level architecture refactoring plan



application builds fail almost every time something is committed to the repository



application components are too tightly coupled and dependent on one another



application components do not have the right level of granularity and isolation from a roles and responsibility standpoint



every time the application is deployed with new functionality, something else usually breaks



application components are too tightly coupled and dependent on one another



application components do not have the right level of granularity and isolation from a roles and responsibility standpoint



application deployments take a long time, sometimes lasting up to 25 minutes



the application is a monolithic and growing quickly based on new features and functionality; it is getting too large for a single application.



users are increasingly reporting performance issues with the application



the application is becoming too large; it is consuming about 95% of the available jvm resources during normal load (memory, cpu, threads)

root cause summary



the application is too tightly coupled and is growing beyond what the current architecture can support

determine architecture changes





the monolithic application is too tightly coupled and is growing beyond what the current architecture can support split the application into multiple deployable units, thereby decoupling components and allowing for more growth potential

justifying your case technical justification



split the monolithic application into multiple deployable units, thereby decoupling components and allowing for more growth potential



components will be more decoupled, thereby eliminating frequent build issues



each part will use fewer jvm resources, thereby increasing performance and allow for more growth



deployment is limited to a separate application unit, thereby reducing deployment time and increasing robustness

justifying your case business justification



split the monolithic application into multiple deployable units, thereby decoupling components and allowing for more growth potential



new functionality can be delivered faster, thereby improving overall time to market



overall application quality will be improved, thereby reducing bugs and the associated costs of fixing them



development and deployment costs associated with developing new functionality will be significantly reduced

refactoring techniques



work in small iterations

identify the technical and business value expected at each iteration

use a playbook approach to outline the architecture transformations

decide on migration vs. adaptation (or a combination of both)

refactoring techniques playbook approach

each iteration should clearly illustrate the changes to the architecture each step along the way



refactoring techniques playbook approach



current state

iteration 1

iteration 2

iteration 3

identify the purpose behind each iteration

identify the technical and business value for each iteration

try to minimize "staging iterations"

keep iterations as small as possible while still providing enough technical and business value

refactoring techniques

migration vs. adaptation



migration

the replacement of old components with new ones through migration over time

refactoring techniques migration vs. adaptation



migration

easier to rollback changes

less overall risk

requires switching logic in calling components

refactoring techniques

migration vs. adaptation



adaptation

refactoring of existing components into new functionality

refactoring techniques migration vs. adaptation





harder to rollback changes

no changes to calling components



presenting your case



presenting your case



"how did things get this bad in the first place?"



always present your plan to your immediate manager before going to the business

presenting your case



don't scare people -"present with urgency, not with panic"

presenting your case



although you may know all the answers, don't be argumentative - take this as an opportunity to educate the business instead

Architecture Anti-pattern



...but further on leads you into a maze filled with monsters

Imposter Syndrome

a term coined in the 1970s by psychologists and researchers to informally describe people who are unable to internalize their accomplishments



architecture boundaries

architectural boundaries



there is an art to defining the box that development teams can work in to implement the architecture
architectural boundaries

tight boundaries



architectural boundaries

loose boundaries



architectural boundaries

appropriate boundaries



architect personalities



control freak architect

architect personalities



armchair architect

architect personalities



effective architect

the architect defines the architecture and design principles used to guide technology decisions



development team: we decided to incorporate the guava library for the camel case conversion requirement.



"take it out. I only want you to use the core java api for this application. period."

development team: we decided to incorporate the guava library for the camel case conversion requirement.



"guava - that's a cool library name. carry on..."

development team: we decided to incorporate the guava library for the camel case conversion requirement.



if that's the only feature you are leveraging, you should just use the java api. if there are other features you can justify, then we can talk about it.



what design principle would you create to manage this type of boundary?

look for overlaps in existing functionality

always seek justification for adding a new library

provide guidance by making it clear what type of libraries need discussion and approval and which ones don't



development team: we need better performance - can we access the database directly from the presentation layer?



development team: we need better performance - can we access the database directly from the presentation layer?

presentation layer	component component
business layer	component component
persistence layer	component component
database layer	

development team: we need better performance - can we access the database directly from the presentation layer?



"no."

development team: we need better performance - can we access the database directly from the presentation layer?



"it doesn't matter to me. if you think it would help performance, then go for it."

development team: we need better performance - can we access the database directly from the presentation layer?



"those layers are closed so that we can better control change through layer isolation, so no. have you been able to identify what might be causing the performance issues?"



what design principle would you create to manage this type of boundary?





what design principle would you create to manage this type of boundary?

clearly document and diagram the architecture

justify your reasons for the architecture decisions

make sure you effectively communicate your decisions

architecture scope

development team: we added some really cool capabilities that might be needed sometime in the future...



"did i tell you to add those capabilities? didn't think so. take them out."

architecture scope

development team: we added some really cool capabilities that might be needed sometime in the future...



"great forward thinking guys! someday the users might need that capability, and now we have it ready..."

architecture scope

development team: we added some really cool capabilities that might be needed sometime in the future...



"let's verify those features with the analysts to see if we need them. if not then we'll take them out. we just need to make sure we don't do anything to prevent that capability from being added in the future."

controlling the boundaries architecture scope



what design principle would you create to manage this type of boundary?

adding additional features over and above the requirements adds additional development, testing, and maintenance time and costs

this practice can lead to the infinity architecture anti-pattern

document non-required features, verify it with the user community, and *make sure you don't do anything to restrict that functionality in the future*

Architecture Anti-pattern



...but further on leads you into a maze filled with monsters



Yesterday's best practice is tomorrow's anti-pattern.

<u>A Case against the GO TO Statement.</u>

by Edsger W.Dijkstra Technological University Eindhoven, The Netherlands



Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to effectuate the desired effect, it is this process that in its dynamic behaviour has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF











...but further on leads you into a maze filled with monsters

martinfowler.com/bliki/AntiPattern.html

Architecture is abstract until operationalized.

nealford.com/memeagora/2015/03/30/architecture_is_abstract_until_operationalized.html



Architecture is abstract until operationalized.





nealford.com/memeagora/2015/03/30/architecture_is_abstract_until_operationalized.html







@neal4d

ThoughtWorks*

NEAL FORD

Director / Software Architect / Meme Wrangler

O'REILLY®

SOFTWARE ARCHITECTURE SERIES



Clojure Inside Out Stuart Halloway & Neal Ford VIDEO

VAR



Functional Thinking: Functional programming using Java, Clojure & Scala

Neal Ford

VIDEO

O'REILLY [®] SOFTWARE ARCHITECTURE SERIES	O'REILLY* SOFTWARE ARCHITECTURE SERIES	O'REILLY [®] SOFTWARE ARCHITECTURE SERIES	O'REILLY [®] SOFTWARE ARCHITECTURE SERIES	O'REILLY* SOFTWARE ARCHITECTURE SERIES
Software Architecture Fundamentals: Part 1: Understanding the Basics	Software Architecture Fundamentals: Part 2: Taking a Deeper Dive Neal Ford, Mark Richards	Software Architecture Fundamentals Part 3 Soft Skills: Problem Solving, Decision Making, Refactoring, Productivity, & Communications Neal Ford, Mark Richards	Software Architecture Fundamentals Part 4 Soft Skills: Leadership, Negotiation, Meetings. Working with People, & Building a Tech Radar Neal Ford, Mark Richards	Engineering Practices for Continuous Delivery Neal Ford
VIDEO	VIDEO	VIDEO	VIDEO	VIDEO