AARHUS UNIVERSITET

Microservices and DevOps

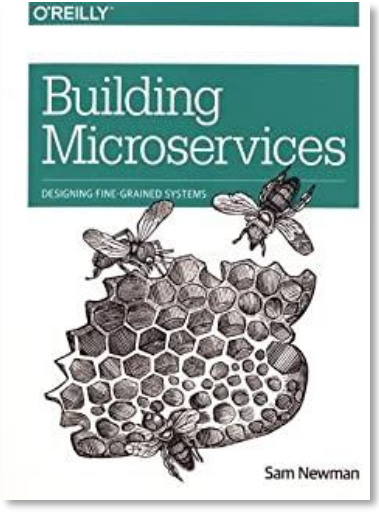
Scalable Microservices

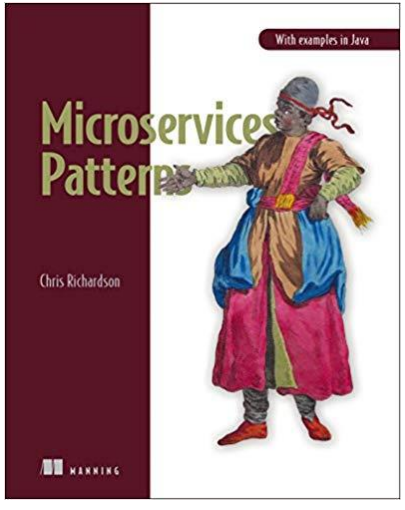
Splitting the Monolith aka.
Application Modernization

Henrik Bærbak Christensen

Introduction

- Making these slides took some effort to structure…
  - Newman talks a bit here and there …
- Using Newman §5 + GOTO stuff, and Richardson §13





AARHUS UNIVERSITET

Context

- OK, so we have this monolith
  - that is busting at the seams, **or**
  - the sensible starting point for our MS architecture
- We want to *start the process of splitting it into sensible microservices.*
  - Entails:
  - *What process do we use to split the monolith*
  - *What boundaries in the monolith shall we split along*
    - *Or do we need to make them first?*
  - *What tactics/patterns to use for each 'extraction'*

Context

- We want to *start the process of splitting it into sensible microservices.*
  - **Splitting Process**
  - **Splitting Boundaries: Seams**
  - **Extraction Tactics**

AARHUS UNIVERSITET

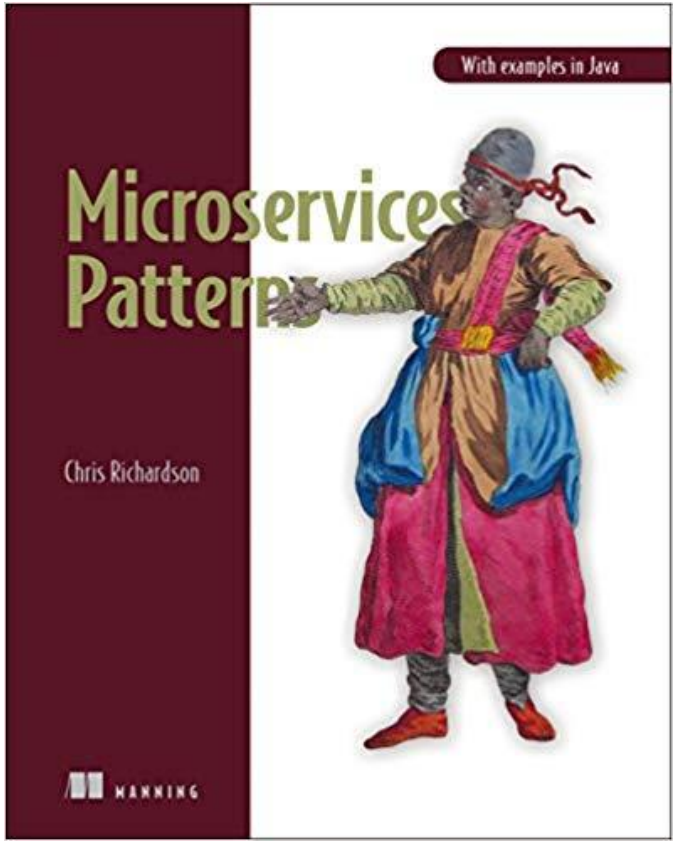
Splitting Process

Based primarily on Richardson

Appl. Modernization

- Source:
  - Chris Richardson

- **Application Modernization**: The process of converting a legacy application to one having a modern architecture and technology stack.



Rule #1

Do not do a big bang rewrite!

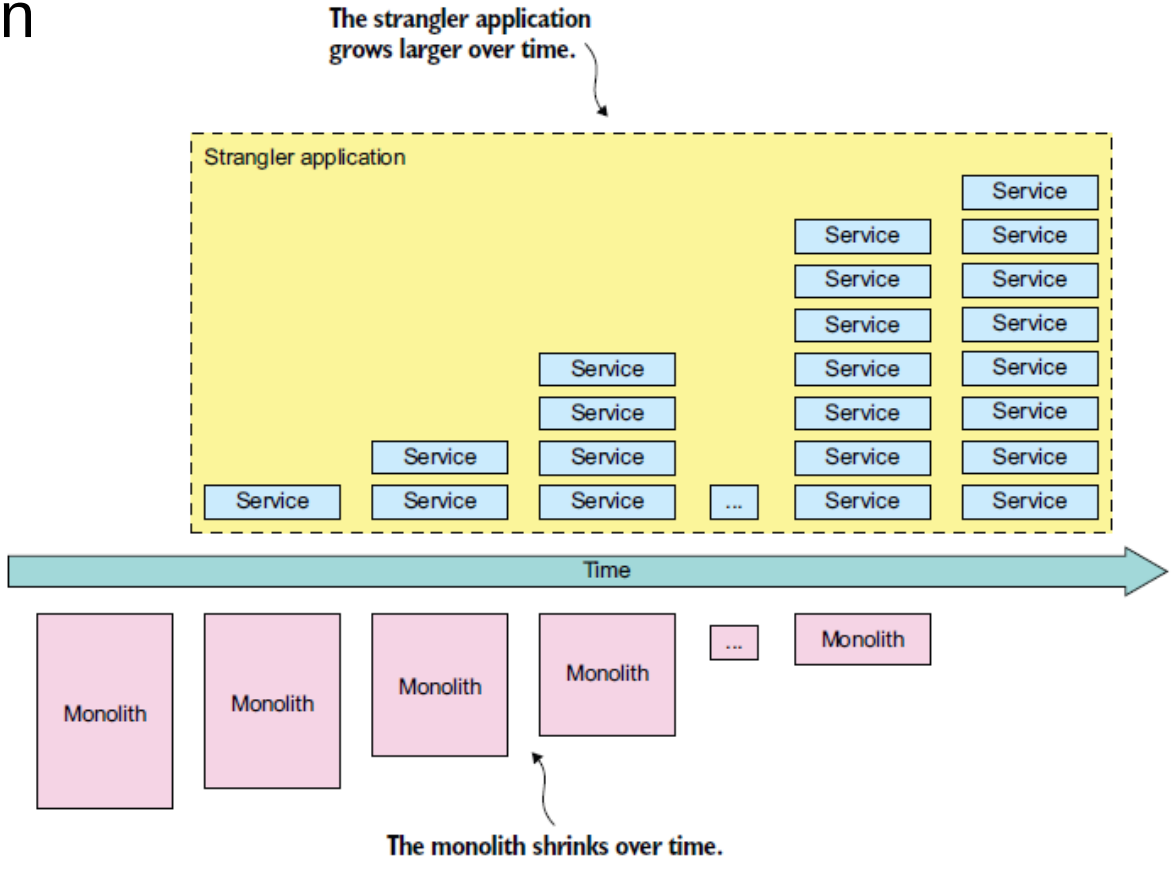
Strangler Pattern

- Strangler Application Pattern:
  - *Modernize an application by developing a new (strangler) application around the legacy application*
- Typically takes months/years

AARHUS UNIVERSITET

Strangling the Monolith

- From Richardson



AARHUS UNIVERSITET

Testing is Central…

- ”The only thing you can’t live without, is a deployment pipeline that performs automated testing.”

[Richardson, p 433]

Software Vise

- Testing is central, because…

Software Vise: Automated tests around our software unit, fixating the behavior, and early detect any behavioral changes.

- … it creates confidence that we do not introduce defects while refactoring the architecture…

AARHUS UNIVERSITET

Strangler Appl

- Benefits
  - Demonstrate value early and often
  - Minimize changes to the monolith
  - Evolve the Tech Deployment Infrastructure as you go
    - Aka: ‘You do not need it all now’…
  - And *you learn as you go!*
    - How to containerize, deploy, release, test, …

Refactoring Strategies

- Proposed strategies for strangling:
  - Implement *new* features as services
  - Separate presentation tier from backend tier
  - Break monolith by extracting business capabilities into services
    - That is: The real strangling

AARHUS UNIVERSITET

New Features as Services

- *Law of Holes: If you find yourself in a hole, stop digging*
  - If you horse is dead, then dismount

- New features of the Monolith that is ‘ball of mud’ should be developed as microservices
  - Get experience with the MS paradigm
  - Stops the monolith growth
  - Does not invalidate current monolith behavior
    - Lessening the risk

Presentation Tier Splitting

- Often the Presentation tier is a feasible boundary to do a split

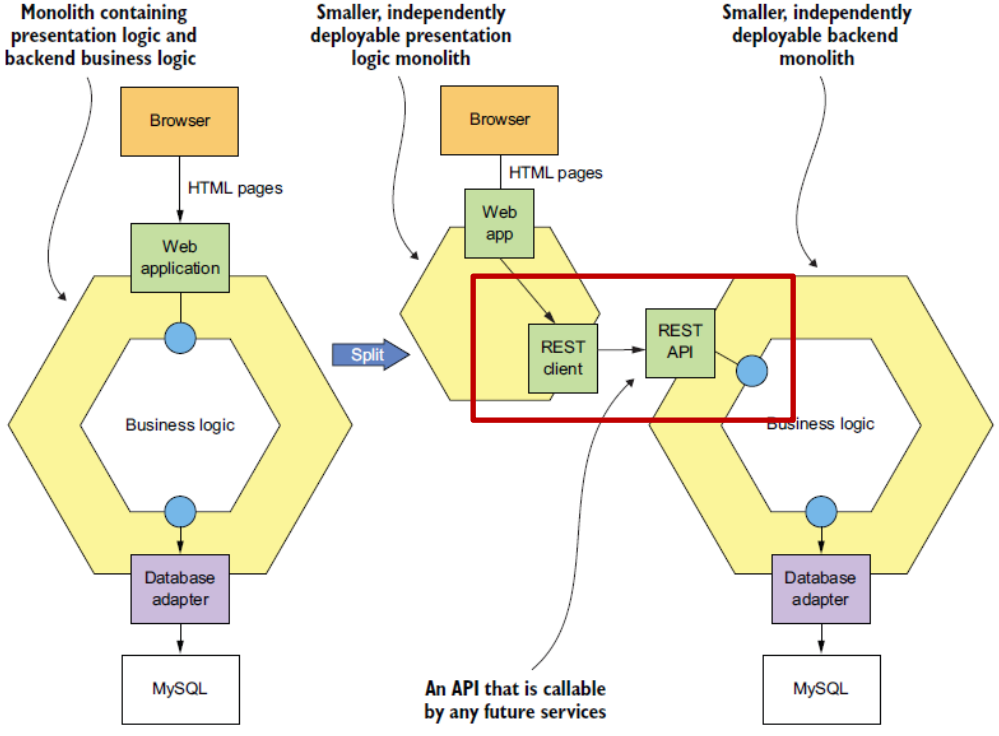


Figure 13.3 Splitting the frontend from the backend enables each to be deployed independently. It also exposes an API for services to invoke.

AARHUS UNIVERSITET

Extract Business Capabilities

- Extract one business capability at a time
  - Applying the strangler pattern
- Entails
  - Splitting the domain model
  - Refactoring the database
- … Which leads to the next headline…

Splitting Boundaries: Seams

AARHUS UNIVERSITET

Seams (?)

- *Seam*: Portion of code that can be changed without impacting the rest of the code (Newman)
- Feather himself:

> **Seam**
> A seam is a place where you can alter behavior in your program without editing in that place.

- Merriam-Webster dictionary

Definition of *seam* (Entry 1 of 2)

1 **a** : the joining of two pieces (as of cloth or leather) by sewing usually near the edge

b : the stitching used in such a joining

2 : the space between adjacent planks or strakes of a ship

AARHUS UNIVERSITET

Seamless Confusion

- Seams is a good analogy, but in my mind, authors each have their own understanding of what it means
  - *Two independent 'things' are joined by a seam, making them form a bigger and more relevant 'whole.'*
  - But Feathers/Newman define the seam to be the 'things' ???
- Anyway – *software architecture is about cutting the whole into 'the right' pieces (creating the seams the right places) and use a suitable technique for joining them.*
  - Joining techniques:
    - API calls (in-process),
    - API calls (out-of-process)
    - Using a zillion different techniques: MQ, shared variable, methods,…

AARHUS UNIVERSITET

Better Terminology

- Component – Connector Viewpoint

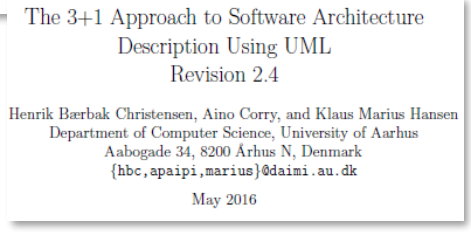
3.2.3 Elements and Relations

The C&C viewpoint has one element type and one relation type:

Component: A functional unit that has a well-defined behavioural responsibility.

Connector: A communication relation between components that defines how control and data is exchanged.

- Component (Newman seam)
  - A microservice, a process
- Connector (what I would call a seam…)
  - The communication: MQ, REST, …


The 3+1 Approach to Software Architecture Description Using UML
Revision 2.4

Henrik Bærbak Christensen, Aino Corry, and Klaus Marius Hansen
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{hbc,apaipi,marius}@daimi.au.dk

May 2016

AARHUS UNIVERSITET

Monolith and Coupling

- Coupling and cohesion have non-code aspects: The deployment aspect
  - It may be that my code change is highly cohesive and decoupled from the rest of the codebase, but…
  - … if I have a monolith, then I have to *redeploy the full monolith due to that single code change*
  - **High coupling in the deployment viewpoint/aspect ☹**

- Microservices enable decoupling in the deployment space *as well* as in the code space

AARHUS UNIVERSITET

BC as Seam

- Newman argue (or postulate?) that bounded contexts work well as seams
- Process
  - Break apart as modules first *kind of ‘monolith first’ pattern*
  - ‘Microservicize’ modules next…
- In which order? Consider
  - Pace of change: Pick the one which changes fast
  - Team structure: Pick the one whose team is out-of-house
  - Security: Pick the one that has stricter security concerns
  - Technology: Pick the one that has specialized tech requirements

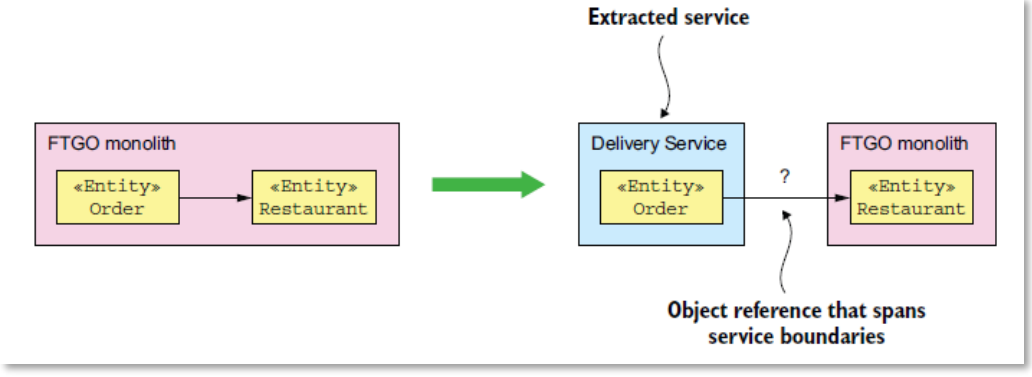
AARHUS UNIVERSITET

Extraction Tactics

Newman focus: The Database

The Issue

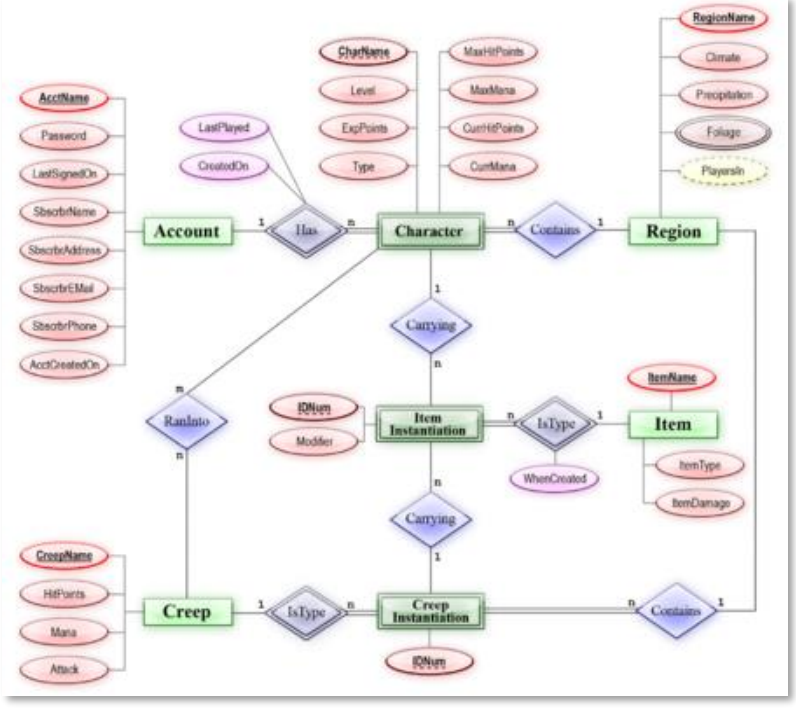
- Breaking a component/module into two microservices means handling those *references* that cross the connector
- Many forms of coupling
  - Object method call
  - Shared data (the database!)
  - …



The Database

- Classic Three-Tier systems all share the same database
  - The breeding ground for a ‘big ball of mud’/Spaghetti structure ☹

- Problem # 1
  - Understanding the table structure
    - Create an ER diagram ☺
  - Can be a major undertaking by itself…
    - Foreign key relations
  - Tools may help…



Foreign Keys

- Ex: Finance
  - Instead of ‘400 copies of sku-185, made 1300$’ we want
  - ‘400 copies of “Matas/hand sanitizer”, made 1300$’
  - Require lookup in ‘catalog’ for key ‘sku-185’ to get item name…
    - Aka ‘join tables’
- However, splitting into two services now…

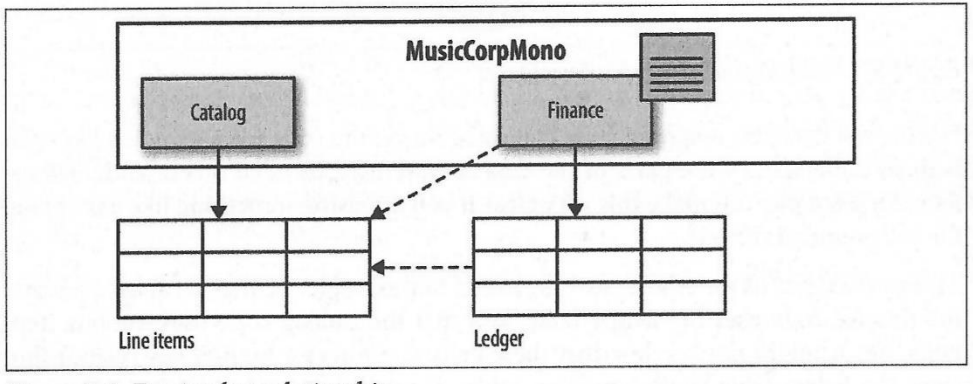


Figure 5-2. Foreign key relationship

Foreign Keys

- Two step breaking (if part of a ‘strangling effort’)
  - Step 1: Replace ‘join’ at database level with API call
    - String itemName = catalog.getNameOf(“sku-185”);
  - Step 2: Replace API call with out-of-service calls
    - Alas, make ‘catalog’ a connector implementation
      - Replace catalogServant with catalogProxy, OR
      - Replace catalogServant with catalogRESTConnector ☺

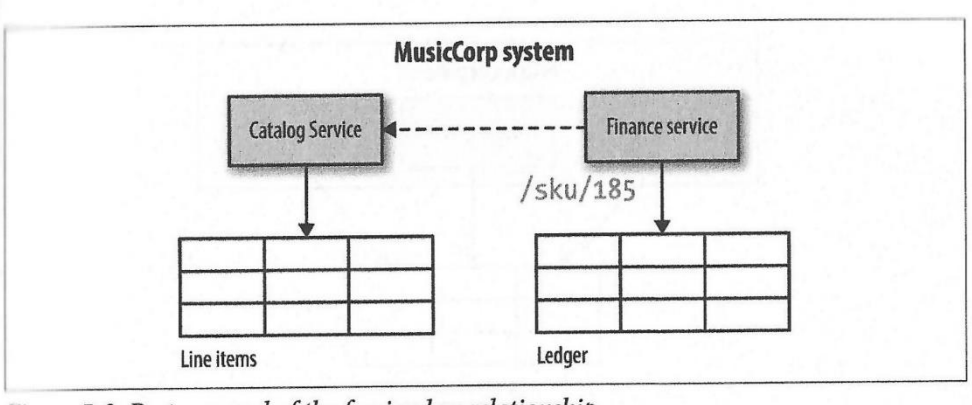


Figure 5-3. Post removal of the foreign key relationship

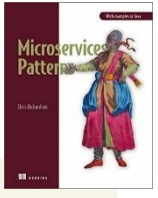
Compare

- Ring a bell?
  - *This is the NoSQL paradigm. No support for joins in the database, replace it with client side (manual) joins*
- The trade-off discussion is the same…
  - One server call is now two server calls
  - Constraint checking (foreign key relations) are now the duty of the clients

AARHUS UNIVERSITET

Transactions

- What about transactions?
  - Commit and rollback / all-or-nothing are strong concepts
- We are getting outside the scope of this course
  - I know rather little of the subject ☺
- General rule
  - Either you *live without transactions*
    - Eventual consistency, correctional events, SAGA
  - Or you *make bigger services*
    - If transaction is vital, and cross boundary of service A and B, then merge them into a single service.



Pattern: Saga
Maintain data consistency across services using a sequence of local transactions that are coordinated using asynchronous messaging. See http://microservices.io/patterns/data/saga.html.

AARHUS UNIVERSITET

Shared Static Data

- Example: Country code table in shared DB
- Solutions:
  - Duplicate the table in all *distributed data management layers*
    - Consistency issue – duplicated data ☹
  - Data as Code – read a property file instead of DB access
    - Consistency issue – but easier than DB changes
      - Easier to push a new property file into a set of code bases
  - Own service – REST call to get data
    - Often overkill, but consistency now in place

Shared Mutable Data

- Two modules both write/read from **shared data**

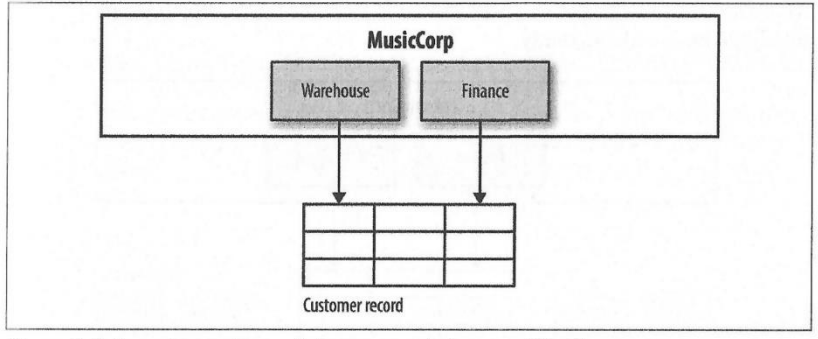


Figure 5-5. Accessing customer data: are we missing something?

- Often because the modeling has missed a BC
  - Solution: Make it ☺

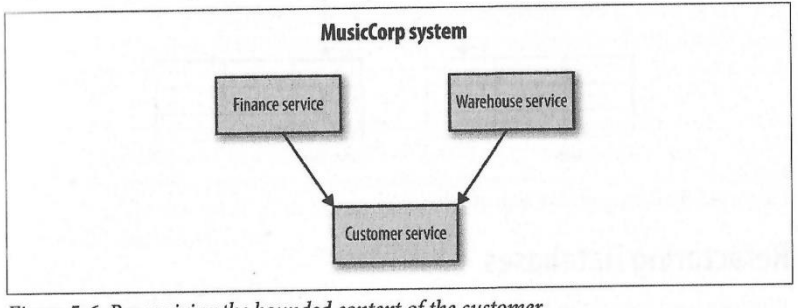


Figure 5-6. Recognizing the bounded context of the customer

Shared Table

- Example: Catalog and Warehouse read/write different columns in a shared table
  - Accidental architectural flaw?

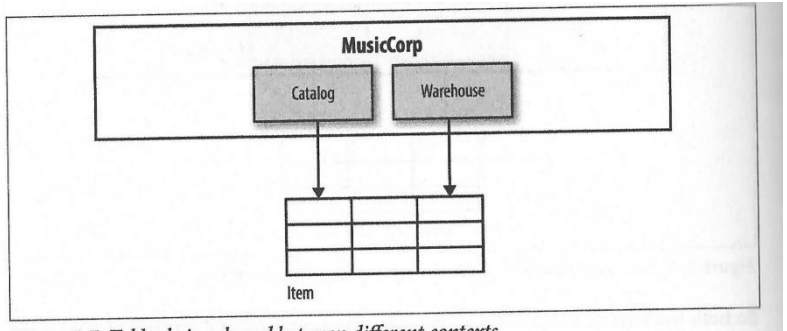


Figure 5-7. Tables being shared between different contexts

- Solution: Break the table
  - Nygard talks more in-detail about that…
    - Shims, trickle-then-batch, etc.

NoSQL?

- How does this relate to NoSQL?
- Well – a lot, basically we have many of the same issues
- The Foreign Key issue is non-existent
  - As we have none of them in a ‘by-the-book’ NoSQL usage
- Potentially we have a worse issue
  - Document paradigm – splitting out embedded documents!
    - { room: “You are in…”, position:”(0,0,0)”, messages: [{…},{…}]}
    - Moving messages to other service is a non-trivial exercise!
- The Shared Data/Table issue is the same

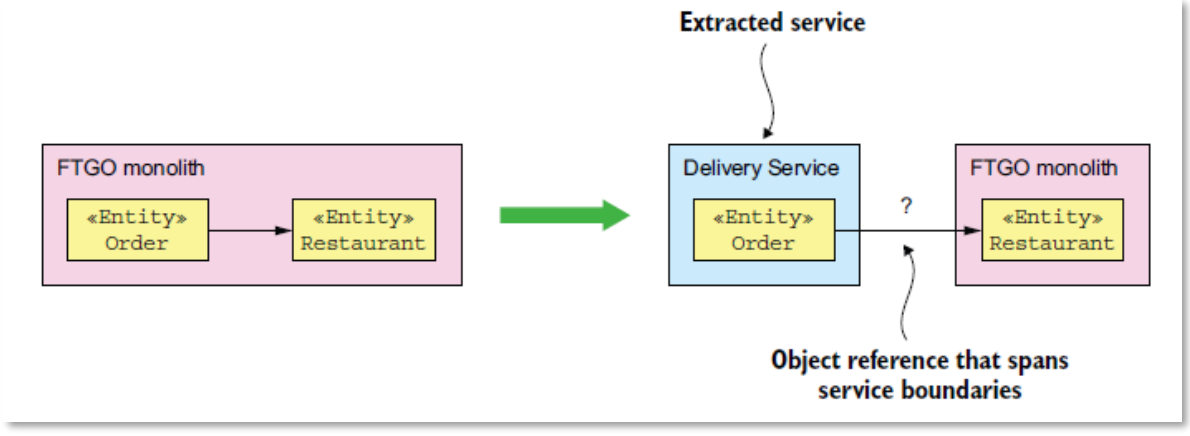
AARHUS UNIVERSITET

Extraction Tactics

But what about API calls?

Newman Forgot?

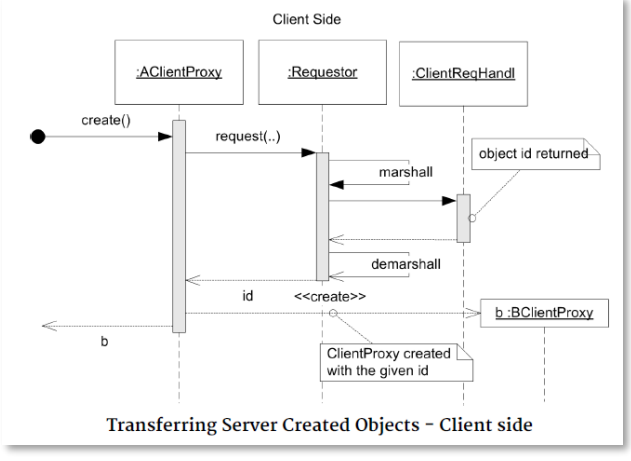
- What about simple API calls using references?
  - myRestaurant.placeOrder(table13Order);



- *In-process object references* makes no sense across process boundaries…

FRDS Broker

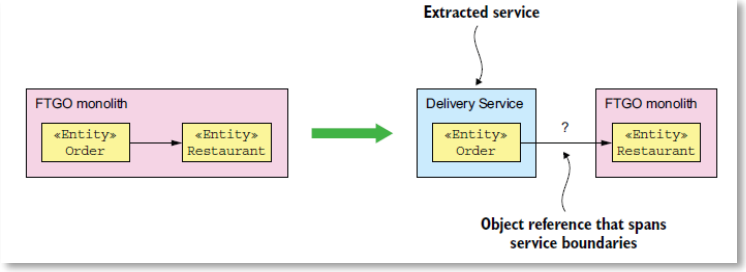
- The very same problem is discussed in my FRDS book regarding the Broker pattern
  - Basic idea: **Replace object reference with a unique ID, which the object owner service can use to access the underlying object, using a name service.**
  - Client side, we create a proxy, that just stores this ID, and then call servant object using this ID
    - Server translate id to servant object

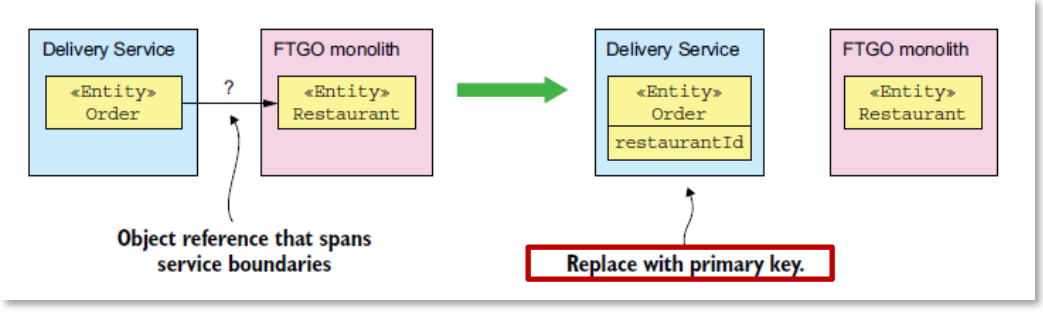


Transferring Server Created Objects – Client side

MicroService Context

- Microservices have to do the same:
  - **Replace object references with unique IDs (“primary keys”)**





- *Aggregates reference each other using primary keys rather than object references*

Sidebar: Aggregates

- *Aggregate:* Collection of domain objects, that are treated as a unit. A domain model is a collection of aggregates.

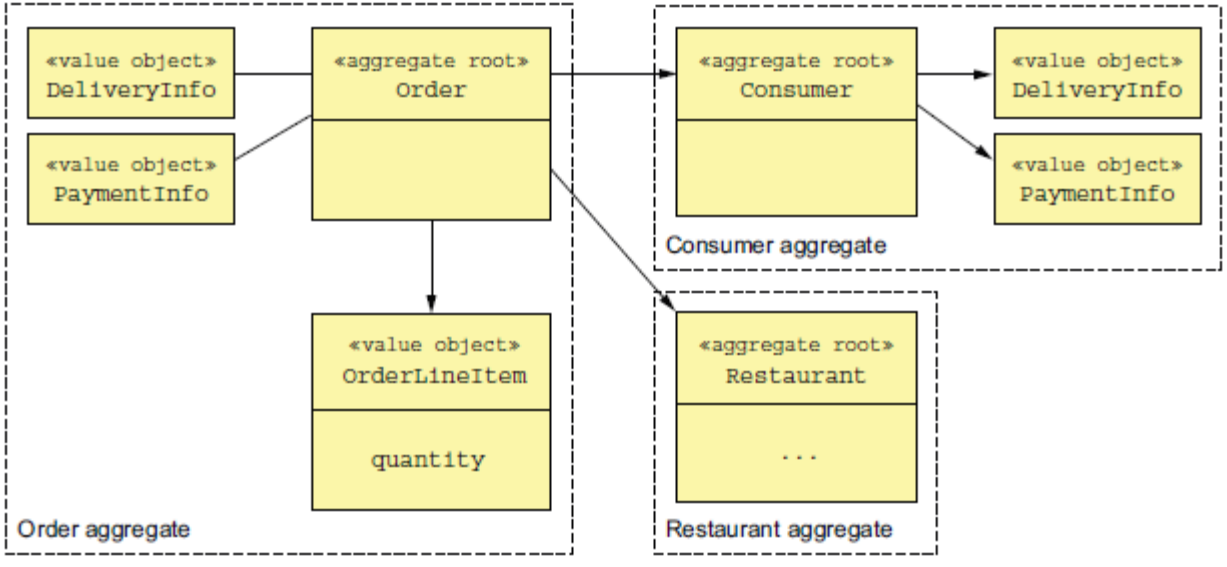


Figure 5.5 Structuring a domain model as a set of aggregates makes the boundaries explicit.

AARHUS UNIVERSITET

REST Context

- In REST
  - A Create is a POST message which must return a Location

```
HTTP/1.1 201 Created
Date: Mon, 07 May 2018 12:16:51 GMT
Content-Type: application/json
Location: http://telemed.baerbak.com/bloodpressure/251248-1234/id73564827

{
  patientId:  "251248-1234",
  systolic:   144.0,
  diastolic:  87.0,
  time: "20180515T094804Z"
}
```

  - So URI's path-segment is an obvious unique ID candidate

AARHUS UNIVERSITET

Branch By Abstraction

If the seams are not clear

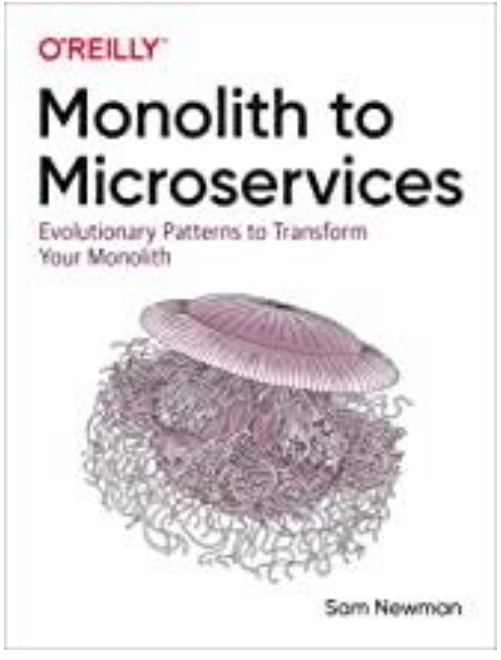
(From a Newman's GOTO Workshop)

AARHUS UNIVERSITET

Newman's New Book

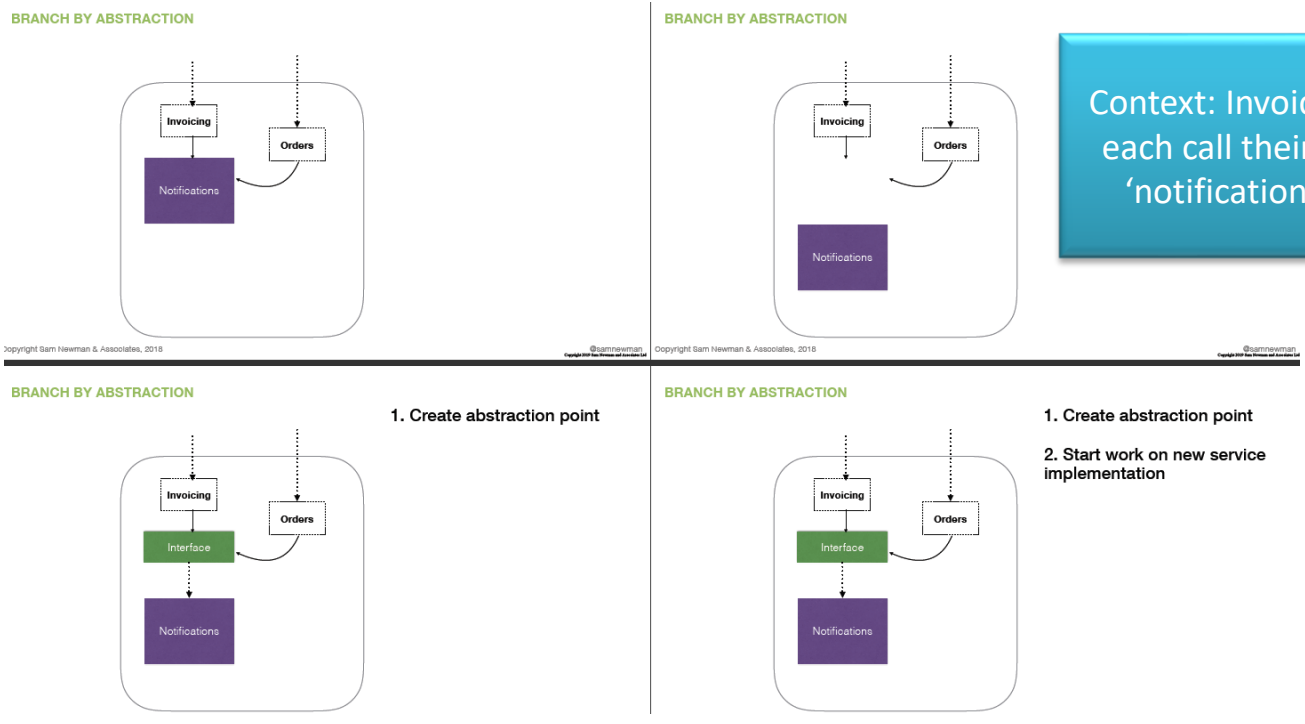
- Slides are from before Newman published his book

- Essential set of patterns, if you are in that situation…



Goto2019 Workshop

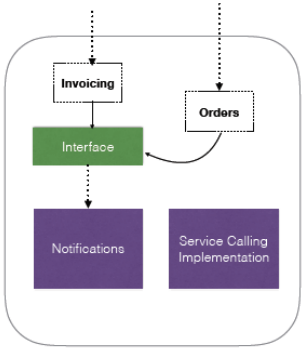
- One very concrete pattern
  - *Monolith Splitting Pattern:* ***Branch By Abstraction***



BRANCH BY ABSTRACTION

1. Create abstraction point

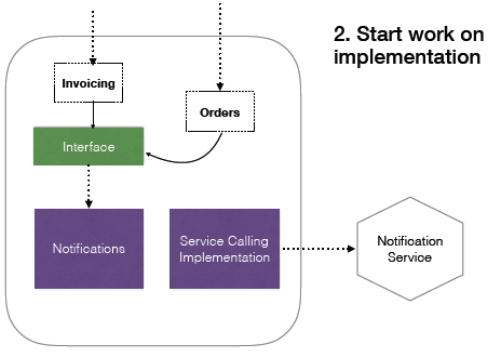
2. Start work on new service implementation



BRANCH BY ABSTRACTION

1. Create abstraction point

2. Start work on new service implementation

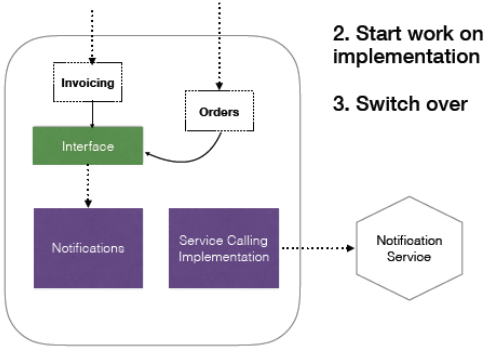


BRANCH BY ABSTRACTION

1. Create abstraction point

2. Start work on new service implementation

3. Switch over

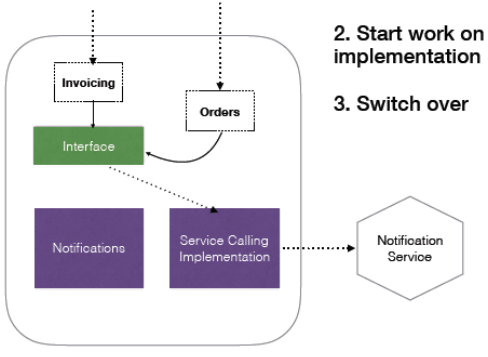


BRANCH BY ABSTRACTION

1. Create abstraction point

2. Start work on new service implementation

3. Switch over



AARHUS UNIVERSITET

BottomLine

- We need to refactor our notification services so you
  - 3) Encapsulate what varies
  - 1) Program to an interface
  - 2) Favor object composition

  Notification

  All that use Notifi…

  - And then let *dependency injection* determine if you use the old legacy code or use the new µ-service

AARHUS UNIVERSITET

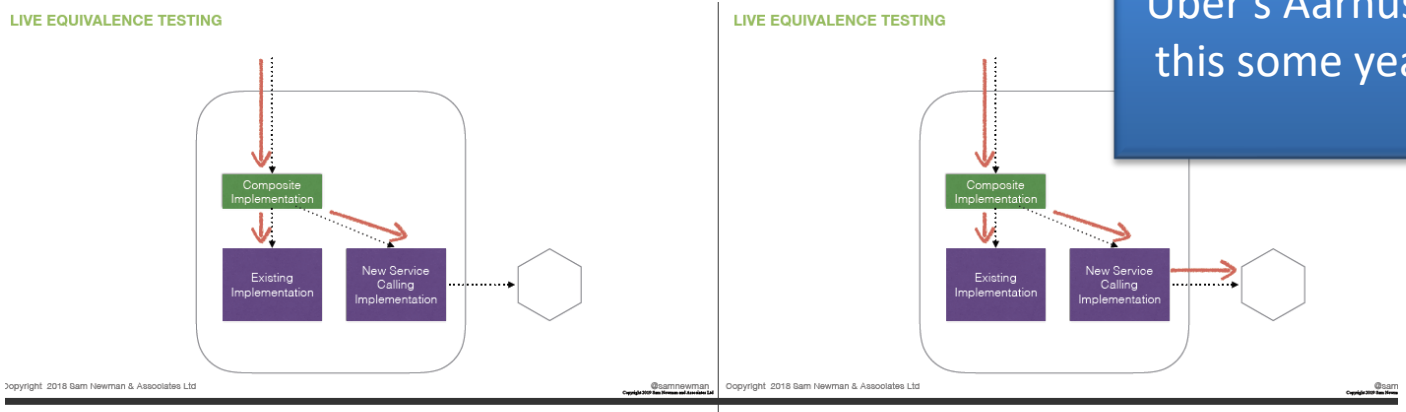
Extraction Tactics

A Testability tactic
for major rework

AARHUS UNIVERSITET

Migration Pattern

- *Live Equivalence Testing*



Uber's Aarhus unit did this some years ago…

Call both implementations & compare

Divert proportion of traffic to test new implementation (canary release)

CANARY RELEASING

https://cloudplatform.googleblog.com/2017/03/how-release-canaries-can-save-your-bacon-CRE-life-lessons.html

AARHUS UNIVERSITET

Reporting

Cross Service Data

AARHUS UNIVERSITET

Reporting

- Reporting: Group together data from across multiple parts of our organization to generate useful output
  - Typically a managerial perspective
    - Purchase patterns, browsing patterns, optimization, sales distribution, …
- Monolith / Three tier case is ‘easy case’
  - Select * from * where({85 lines here})
- But what now?

Reporting

- Retrieval via Service Calls
  - Pull the data by pulling from each service
    - Often require a special ‘reporting API’, to avoid overhead of a zillion GET requests, interfering with normal processing
- Data Pumps
  - Push data to reporting system
    - HTTP calls, or (better) standalone programs with direct DB access
      - (Now we have a shared database, relying on the schema!)
- Event Data Pump
  - Subscribe to ‘state change events’
    - We will return to the ‘event sourcing’ tactics for databases later
- And – Monitoring… Stay tuned…

AARHUS UNIVERSITET

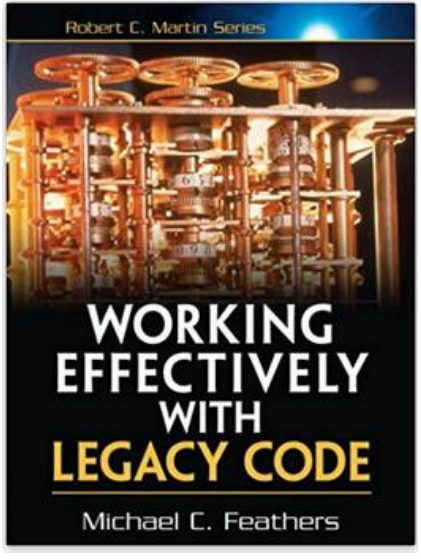
Code Level Legacy Refactoring

Just mentioning …

AARHUS UNIVERSITET

Incremental Testability

- An *incremental* refactoring process to ***introduce*** testability
- Primary Source
  - Martin C. Feathers



Our goal today:
A few key points!

Software Vise

- Approaches
  - **Edit and Pray ☹**
    - Carefully plan changes, ensure you understand the code, make changes, run system and poke around to make you did not break anything…
  - **Cover and Modify ☺**
    - Idea: Make a *safety net* that cloaks our code of interest and ensure no bad changes leak out into the surrounding code.
      - *Covering = Covering with Automated Tests !*
      - Tests as tools to *identify behavioral changes*

Software Vise: Automated tests around our software unit, fixating the behavior, and early detect any behavioral changes.

Summary

- Avoid big-bang rewrites, have strong testing (the ‘vise’), and do it one small step at a time
  - The strangler pattern
  - Lots of extraction tactics
    - For the database
    - For the components and connectors