The Monolith who went on a Cruise!

Presented By: Vikas Hazrati
Knoldus Inc
About me

Vikas Hazrati - Group CTO with Knoldus Inc
19+ years on the JVM stack, 8+ years with Scala

Knoldus Story
Started in 2010, spread across 4 geographies with 120+ engineers
Focused on Reactive ecosystem and functional programming

Partnerships
- Lightbend
- Databricks
- Datastax
- Confluent
Agenda

The story of transformation

- Vision for Digital transformation
  Current state and the wants

- Story of transformation
  How did we approach the huge monolith

- Tools and Techniques
  the toolkit

- Learnings and Recommendations
  wisdom, may be
Vision

To provide guests with an experience that leverages their previous cruises, cruise plannings, and web behavior history to provide an online experience that recognizes who they are, and drives relevant content and offers based on their interests and the desire to drive pre-cruise, current cruise and post-cruise revenues for all categories business and personal presentation.

Digital transformation
Put customer at the center

Personalization
Deliver content and offers based on who they are

Revenues
Capitalise at all levels of pre, current and post cruise activities
Current set of challenges
What is impeding us right now?

- Bulky system, difficult to change and innovate on
- Static content on ships, printed material that nobody reads - lack of engagement
- Details made available to guests on day to day basis. No way to disseminate knowledge beforehand
- Multi language support lacking for international guests
- Pre and post cruise not optimised for upsell
- Getting behind each day on new technology advancements, stifling innovation
- Zero personalization and microsegmentation
Future State
Personalise across the cruise journey

01 Pre cruise analysis
Recommend package based on history and predictive modeling

02 Booking
Guest information to provide enhanced experience

03 Pre boarding
Provide recommendations on activities and promotions to prebook

04 On Cruise
Learn and react on the basis of activities on cruise

05 Post Cruise
Start next cruise planning, offers
Omni-channel

Personalization at all levels

Follow the guest
From curation to analysis to presentation
Create a 360 view of the guest and his needs

Personalise

All interaction channels
Email, Web, Mobile, Print, Kiosks, etc

Target all mediums
Content, commerce, notification, recommendations

Omni channel would be designed to do the following
A perfect case for Digital transformation
A Prereq for Digital Transformation

- Time to Market
- Technology Relevance
- Faster Innovation
- Service enablement
What was clear

Curation
Understand guest, profile, populate build models

Analysis
Track events, draw predictions, create campaigns

Present
Rules at channel level, give recommendations and offers, feed into machine learning models
How - Reactive Microservices

Integrate with the existing 3rd party systems

Identify the functionalities which would deliver fast results

Deliver the new platform

Start the conversion
This sounds great
Let’s start but where do we start

Where?
It is a huge domain, consists of subdomains working together

Why?
To reap the benefits of new technology and futuristic architecture

How?
Converting monolith to microservices piece by piece

What?
One subdomain at a time

When?
Yesterday!

Who?
The business team and the development team
The actual story, now
Things are going to get dark

pain and sweat ahead
Monolith Decomposition
easier said than done

- Monolith is currently in **production**, taking **load** and **bringing** business value
- Very little disruption, **if at all**, expected to running business
- It would be difficult if not impossible to make changes to **data model** underneath
- Approach should **reduce the risk** of evolution
Some books recommend

- Identify the nouns in the system
- Do one thing and one thing well
- Single responsibility principle

- Good Advice but does not quite work for Legacy transformation
Approach

01 Identify Modules
Existing and New

02 Wrap with Service
After breaking out tables

03 Update Code for other services
To now use the API instead of tables

04 Rinse and Repeat
Modules

Huge number of interconnected modules

- Marketing
- Product
- Guest
- Vacation
- Ship board
- Emergency
- Dining
- Transport
- Security
- Crew
- Navigation
- Payment
- Content
- Luggage
Modules Drill Down

Guest Management

- Loyalty Management
- Guest Experience Management
- Digital Folio Management
- Identity Management
- Self-service Provisioning
- Guest Interaction Management
- Personalization Management
- Guest Service Management
- Guest Memories Management
- My Calendar Events Management
- Transportation Management
- Profile Management
Guest Account

Identify critical sub modules of one module

Loyalty
Identity
Personalization
Profile
Promotions
Self Service
Memories & Calendar
Great, let's decompose
Step 1
Identify modules and corresponding tables
Step 2

Breakout tables, wrap with service, update dependencies
Step 3
Ideal world, Repeat and Rinse
This looks great!
And we failed :(

Rarely, if at all, do monoliths lend themselves to **nice and neat modularization**

- Relationships **between tables can be highly normalized** and **exhibit tight coupling**
- **Understanding** monolithic code and connecting wires is a challenge
- We cannot migrate everyone to use the service on **click of a button**
- Ugly migration steps that cannot just be wished away
- Point of diminishing returns where it doesn’t make sense to break things out of the monolith
Let’s re-attempt
The monolith is tightly coupled
High coordination needed between different teams
We need to go through the CI pipeline
We cannot do a Big Bang, that is just a theory ;) … Sheldon!
Current State

Guest Account

DB
Step 1
Decouple the UI, created FE and BE team
Step 2

Let us try to introduce a new service

- **Understand the bounded context** within a monolithic module
  - Identify the context which *changes more rapidly* than the rest
  - Is *relatively disjoint* from rest of the monolith, can have its own db
  - *Focus on Events it emits or consumes*, Event storming, think about events which would be published as we take transactional load
  - *Seed the database* with data from the monolith as a start
  - There *should not be dependency* from microservice to monolith
Step 2
Introduce a new service
Let’s send some shadow traffic
Step 3
Send shadow traffic in production
Step 3.1
Compare the new service data with old monolith data
Step 4

Start the traffic flow to the new service - Gateway
Considerations

- Putting **new service into traffic** has inherent risks
- **Need to control traffic** to the service through a gateway which can
  - Route to new service and ignore old
  - Route to old monolith and ignore new
  - Route traffic to both
- We need to deploy this functionality to **some cohorts** and not the complete system
Cut the cord!
Step 5.1

New service just accessible through API now
Tools and techniques
Test Harness

Is paramount

- We took a lot of advice from the book
- The **test harness** of the areas that we were going to separate out were first effectively covered with tests, if the tests did not exist, *like in most cases :)*
- Junit and Arquillian
Subdomains

Identification and separation of bounded contexts

- We took a lot of advice from the book
- **DDD is very compatible with Reactive Architectures**
- Large domain needs to be broken down into smaller bounded contexts and fits well with the concept of microservices which need clear boundaries
- For example Persistent Entity in **Lagom** can be tied back to the entity from DDD
Art of Scaling
Understanding the Scale Cube

- We took a lot of advice from the book
- How to scale systems and organizations
Art of Scaling
Understanding the Scale Cube

1. Functional Decomposition
   Y-Axis Scaling

2. Data Partitioning
   Z-Axis Scaling

3. Horizontal Scaling of Individual Services
   X-Axis Scaling

Monolith  →  Microservices
We took a lot of advice from the manifesto
- Reactive microservices are **fault tolerant and self healing**
- They are **responsive to varying loads**
- They need to support **distributed computing**, this circles back to how we design the domain model
Tech

Technologies in use
Tech
Technologies in use
Technologies in use

<table>
<thead>
<tr>
<th>Technology</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Runtime</td>
<td>Lightbend</td>
</tr>
<tr>
<td>Container OS</td>
<td>Red Hat, Docker</td>
</tr>
<tr>
<td>Container Engine (Runtime)</td>
<td>Mesosphere</td>
</tr>
<tr>
<td>Container Orchestration</td>
<td>Mesos</td>
</tr>
<tr>
<td>Container Image Registry</td>
<td>Built-In</td>
</tr>
<tr>
<td>Service Discovery</td>
<td>Consul</td>
</tr>
<tr>
<td>Dynamic Property Management</td>
<td>DC/OS</td>
</tr>
<tr>
<td>Health/Circuit Breaker</td>
<td>Netflix, OSS</td>
</tr>
<tr>
<td>Load Balancer</td>
<td>Marathon-Ib</td>
</tr>
<tr>
<td>Host OS</td>
<td>VMware, Red Hat</td>
</tr>
</tbody>
</table>

API Gateway: Apigee
CI: Jenkins
Configuration management: Puppet Labs
Distributed Data store: Cassandra
Data ingest: Kafka, Zookeeper
Stream Processing: Kafka, Spark, Akka
Distributed Tracing: OpenZipkin
Event store Pub/Sub-messaging: Kafka
Logging: Splunk
Monitoring: Dynatrace
Key components

Technologies in use

- **Opinionated** microservices framework for Java and Scala
- Built on top of **Akka and Play**
- Helps build systems with reactive qualities: responsiveness, resilience, scalability, and elasticity
- Out of the box support for ES, CQRS, Circuit breakers etc
Key components

Technologies in use

- Service discovery and registry
- Service segmentation based on rules and authorization
- Service configuration with K/V store
Key components
Technologies in use

- **Large scale data processing** and computation
- **Batch processing** for promotions and predictions
- **Building models** with fast execution on batch and streaming data
Key components

Technologies in use

- **Distributed operating system** based on Mesos
- **Manages multiple machines** by providing deployment, scale and monitoring
- DC/OS is itself a distributed system, a cluster manager, a container platform, and an operating system.
Key components
Technologies in use

- Real time data pipelines
- Pub/Sub, for async communication between microservices
- Kafka connect for Ship Shore integration and data transfer
LaunchDarkly

- Decouples the development team from the product team
- Allows you to do blue-green, A/B testing and Canary releases
- Product teams can control the feature release to the market
- Development teams can learn the nerdy details about the feature
Key components
Technologies in use
Out of the dark

Mostly :)
Journey ahead
Conversion of all modules towards digital transformation
Current state
Where are we right now in the journey

- Guest, Payments and Personalisation microservices rolled out to 60% of the ships
- Data Centers at ship and shore kept in sync
In business terms
Numbers which matter the most

Customer satisfaction
89% Of the customers feel that the company engages much better than the earlier times

Business satisfaction
95% Of the business units feel that they have more flexibility to launch and enhance new products
Key learnings
Learnings
You are just less wrong next time

- Most successful systems start with a **Monolithic first approach** and break down into microservices eventually.
- Most large scale web sites including Netflix, Amazon and eBay have evolved from a monolithic architecture to a microservice architecture.
- Team working on microservices should be conversant with **Domain Driven Design**
- Team should understand the **Reactive Architecture principles**
- Focus on **BASE instead of ACID**
- **Compensating patterns** (Saga) would allow you to go eventually consistent
Learnings
You are just less wrong next time

- **Stop Digging** - Do not add new functionality to the monolith
- An easy (low risk) part to start is to split **FE and BE**. Rest can follow
- Keep the IPC **asynchronous and non-blocking**
- Follow the **scale cube**
- **Testing** microservices is much harder, tests should be first class citizens from day one
- Deploy with **containers**
- **Feature flags, dark launch** is your friend with monoliths in production
- **Monitoring and Operations** need setup early on just like CI/CD
Organizational considerations

- The most challenging aspects of moving from monolithic systems to microservices are the organizational changes required, such as building services teams that own all aspects of their microservices.
- This requires creating multidisciplinary units which include developers, testers and operations staff, among others.
- The idea is to embrace more collective code ownership and care for software craftsmanship.

Learnings
You are just less wrong next time
Interesting links

- Knoldus Blogs
- Knoldus Case Study
- Knoldus Digital Platform
Thank You!

Stay in Touch

vikas@knoldus.com
@knolspeak
www.knoldus.com
blog.knoldus.com