Managing Data at Scale: Microservices and Events

Randy Shoup
@randyshoup
linkedin.com/in/randyschoup
Background

- **VP Engineering at Stitch Fix**
  - Combining “Art and Science” to revolutionize apparel retail

- **Consulting “CTO as a service”**
  - Helping companies scale engineering organizations and technology

- **Director of Engineering for Google App Engine**
  - World’s largest Platform-as-a-Service

- **Chief Engineer / Distinguished Architect at eBay**
  - Multiple generations of eBay’s infrastructure
Stitch Fix

Create Your Style Profile.

Get Five Hand-picked Items.

Keep What You Like. Send Back the Rest.
Stitch Fix

Create Your Style Profile.

Get Five Hand-picked Items.

Keep What You Like. Send Back the Rest.

How do you prefer clothes to fit the top half of your body?

- Mostly Tight / Form Fitting
- Prefer Fitted / Showing my Figure
- **Straight**
- Mostly Loose
- Oversized

How do you prefer clothes to fit the bottom half of your body?

- Mostly Tight / Form Fitting
- Prefer Fitted / Showing my Figure
- Mostly Loose
- Oversized

Pants
- Waist: [ ] Regular [ ] Cuffed [ ] Loose [ ] Skinny
- Leg: [ ] Relaxed Fit [ ] Straight Leg [ ] Bootcut

Skirts
- [ ] L [ ] M [ ] S [ ] XS [ ] XXS [ ] XXXS [ ] Unspecified
Stitch Fix

Stitch Fix

Create Your Style Profile.

Get Five Hand-picked Items.

Keep What You Like. Send Back the Rest.
Combining Art and [Data] Science

- 1:1 Ratio of Data Science to Engineering
  - >75 software engineers
  - >75 data scientists and algorithm developers
  - Unique in our industry

- Apply intelligence to *every* part of the business
  - Buying
  - Inventory management
  - Logistics optimization
  - Styling recommendations
  - Demand prediction

- Humans and machines augmenting each other
Styling at Stitch Fix

Inventory

Personal styling
Personalized Recommendations

Inventory → Machine learning → Algorithmic recommendations

- Personalized recommendations
- Inventory
- Algorithmic recommendations
- Machine learning
Expert Human Curation

Algorithmic recommendations

Human curation

[Diagram showing human and computer interaction]
Evolution to Microservices

- eBay
  - 5th generation today
  - Monolithic Perl → Monolithic C++ → Java → microservices

- Twitter
  - 3rd generation today
  - Monolithic Rails → JS / Rails / Scala → microservices

- Amazon
  - Nth generation today
  - Monolithic Perl / C++ → Java / Scala → microservices
No one starts with microservices

... Past a certain scale, everyone ends up with microservices
First Law of Distributed Object Design:

Don’t distribute your objects!

-- Martin Fowler
If you don’t end up regretting your early technology decisions, you probably over-engineered.
Microservices

• Single-purpose
• Simple, well-defined interface
• Modular and independent
Microservices are nothing more than SOA done properly.

-- me
Microservices

- Single-purpose
- Simple, well-defined interface
- Modular and independent
- Isolated persistence (!)
Microservice Persistence

• Approach 1: Operate your own data store
  o Store to your own instance(s) of {Postgres, MySQL, etc.}, owned and operated by the service team

• Approach 2: Use a persistence service
  o Store to your own schema in {Dynamo, RDS, Spanner, etc.}, operated as a service by another team or by a third-party provider
  o Isolated from all other users of the service

• ➔ Only external access to data store is through published service interface
Microservices

**Pros**
- Each unit is simple
- Independent scaling and performance
- Independent testing and deployment
- “Optimal” technology stack
- Security boundary

**Cons**
- Multiple cooperating units
- Exchange in-process for network latencies
- More sophisticated deployment and monitoring tools
- Overall system complexity
Why Rearchitect?

• **Velocity**
  - Time to market is constrained by coupling and lack of isolation in the monolith
  - Teams step on each others' toes, and can no longer develop independently
  - Difficult for new engineers to be productive

• **Scaling**
  - Vertical scaling of the monolith no longer works
  - Parts of the system need to scale independently of others
Why Rearchitect?

- Deployment
  - Parts of the system need to deploy independently of others
  - Monolithic release is too slow, too complicated, too risky
“The only thing a Big Bang migration guarantees is a big *Bang*.”

-- Martin Fowler
Extracting Microservices

- Problem: Monolithic shared DB

- stitchfix.com
- Styling app
- Warehouse app
- Merch app
- CS app
- Logistics app
- Payments service
- Profile service

- Clients
- Shipments
- Items
- Styles, SKUs
- Warehouses
- etc.
Extracting Microservices

• Decouple applications / services from shared DB

- stitchfix.com
- Styling app
- Warehouse app
- Merch app
- CS app
- Logistics app
- Payments service
- Profile service

- Clients
- Shipments
- Items
- Styles, SKUs
- Warehouses
- etc.
Extracting Microservices

- Decouple applications / services from shared DB

**Diagram:**
- Styling app
- Warehouse app
- core_client
- core_sku
- core_item
Extracting Microservices

- Step 1: Create a service

  - Styling app
  - Warehouse app
  - client-service
  - core_client
  - core_sku
  - core_item
Extracting Microservices

- Step 2: Applications use the service
Extracting Microservices

- Step 3: Move data to private database
Extracting Microservices

- Step 4: Rinse and Repeat
Extracting Microservices

• Step 4: Rinse and Repeat
Extracting Microservices

- Step 4: Rinse and Repeat
With Microservices, how do we do

• Shared Data
• Joins
• Transactions
Events as First-Class Construct

• “A significant change in state”
  o Statement that some interesting thing occurred

• Traditional 3-tier system
  o Presentation ➔ interface / interaction
  o Application ➔ stateless business logic
  o Persistence ➔ database

• Fourth fundamental building block
  o State changes ➔ events
  o 0 | 1 | N consumers subscribe to the event, typically asynchronously
Microservices and Events

- Events are a **first-class part** of a service interface.

- A service interface includes:
  - Synchronous request-response (REST, gRPC, etc)
  - Events the service produces
  - Events the service consumes
  - Bulk reads and writes (ETL)

- The interface includes **any mechanism for getting data in or out of the service (!)**
Microservice Techniques: Shared Data

- Monolithic database makes it easy to leverage shared data

- Where does shared data go in a microservices world?
Microservice Techniques: Shared Data

• Principle: Single System of Record
  o Every piece of data is owned by a single service
  o That service is the **canonical system of record** for that data

• Every other copy is a **read-only, non-authoritative cache**
Microservice Techniques: Shared Data

• **Approach 1: Synchronous Lookup**
  - Customer service owns customer data
  - Fulfillment service calls customer service in real time
Microservice Techniques: Shared Data

- Approach 2: Async event + local cache
  - Customer service owns customer data
  - Customer service sends `address-updated` event when customer address changes
  - Fulfillment service caches current customer address
Microservice Techniques: Shared Data

- Approach 3: Shared metadata library
  - Read-only metadata, basically immutable
  - E.g., size schemas, colors, fabrics, US States, etc.
Microservice Techniques: Joins

- Monolithic database makes it easy to join tables

\[
\text{SELECT FROM A INNER JOIN B ON ...}
\]

- Splitting the data across microservices makes joins very hard
Microservice Techniques: Joins

- Approach 1: Join in Client Application
  - Get a single customer from `customer-service`
  - Query matching orders for that customer from `order-service`
Microservice Techniques: Joins

- Approach 2: Service that “Materializes the View”
  - Listen to events from item-service, events from order-service
  - Maintain denormalized join of items and orders together in local storage
Microservice Techniques: Joins

• Many common systems do this
  o “Materialized view” in database systems
  o Most NoSQL systems
  o Search engines
  o Analytic systems
Microservice Techniques: Workflows and Sagas

- Monolithic database makes transactions across multiple entities easy
  
  \[\text{BEGIN}; \text{INSERT INTO A} \ldots; \text{UPDATE B} \ldots; \text{COMMIT};\]

- Splitting data across services makes transactions very hard
Microservice Techniques: Workflows and Sagas

• Transaction ➔ Saga
  o Model the transaction as a state machine of atomic events

• Reimplement as a workflow

• Roll back by applying compensating operations in reverse
Microservice Techniques: Workflows and Sagas

• Many common systems do this
  o Payment processing
  o Expense approval
  o Travel
  o Any multi-step workflow
Microservice Techniques: Workflows and Sagas

- Simple event-driven processing
  - Very lightweight logic
  - Stateless
  - Triggered by an event

Consider Function-as-a-Service (“Serverless”)
With Microservices, how do we do

- Shared Data
- Joins
- Transactions
Events!
Thank You!

- @randyshoup
- linkedin.com/in/randyshoup