The FoxCommerce System Architecture

May 25, 2016

1 Overview

This article describes various aspects of the FoxCommerce Architecture. We will go over three parts of the system.

1. Overall Design
2. Scalability
3. Data Model

2 Overall Design

2.1 Perceptive

The single best word to describe the FoxCommerce architecture is *Perceptive*. Using the Event Sourcing Pattern, the system responds immediately to changes. Tasks such as indexing, messaging, alerting, and integrating with 3rd party systems happen in real time.

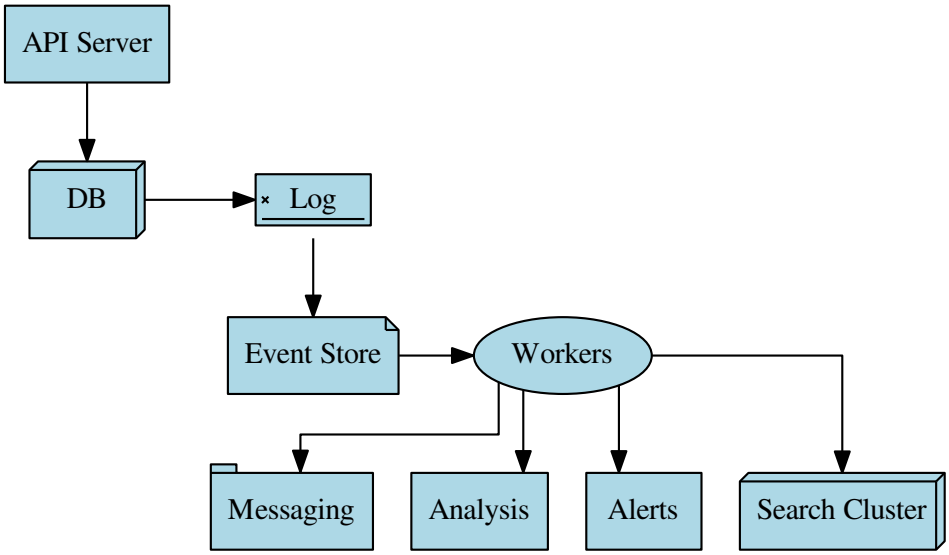
The Event Sourcing Pattern allows access to detailed information about the system enabling insights that would not have been possible otherwise.

2.2 Event Sourcing

Most Databases record every change event to a log. The database then applies the change events one after another to the data. You can think of the data in the database as a snapshot of the change events. Event sourcing is the idea of storing events in a append only data store.

The FoxCommerce system takes the database log and saves it in a distributed messaging system. Event Consumers process these change events

to do various tasks like indexing, messaging, analysis, and alerting, just-in-time as events are received. This allows us to avoid batch processing and enables a responsive system. We call this event sourcing architecture the "Green River"



2.3 Everything Searchable

Because of the event sourcing architecture, we index all data in a search cluster. This enables all data in the system to be searchable using a flexible query language. The search cluster can be used to get data instead of the DB which allows improved performance and functionality like full text search.

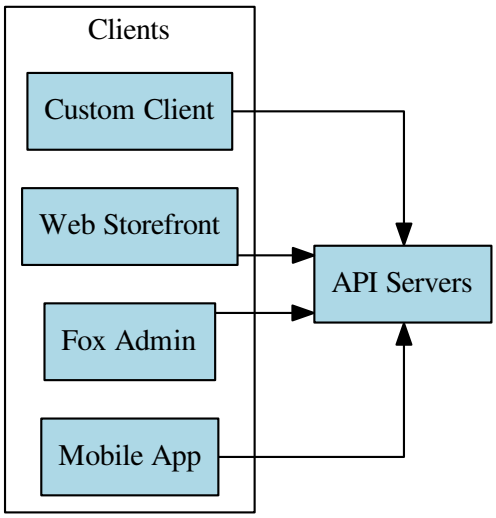
2.4 Extensible

There are three ways the system is extensible.

1. API First
2. API.js
3. Event Consumers

2.4.1 API First

The system is designed API first. This means that all our components are programmed against an HTTP API that is available to everyone. This allows customers to build custom applications and other components against the system. The API is divided into public and private versions which the customer has full access to.



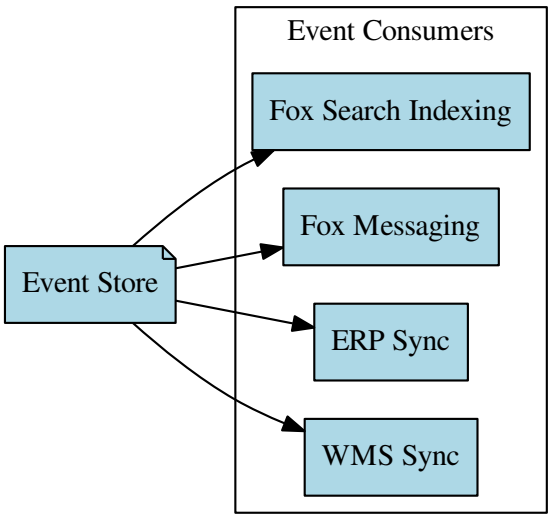
2.4.2 API.js

Ease of use is an incredibly important design criteria. That is why we built an easy to use JavaScript library to query the API. This library provides functions to get and put data into the system. The customer can then build their own components easily on top. The library is designed to use the simplest and most straightforward JavaScript.

2.4.3 Event Consumers

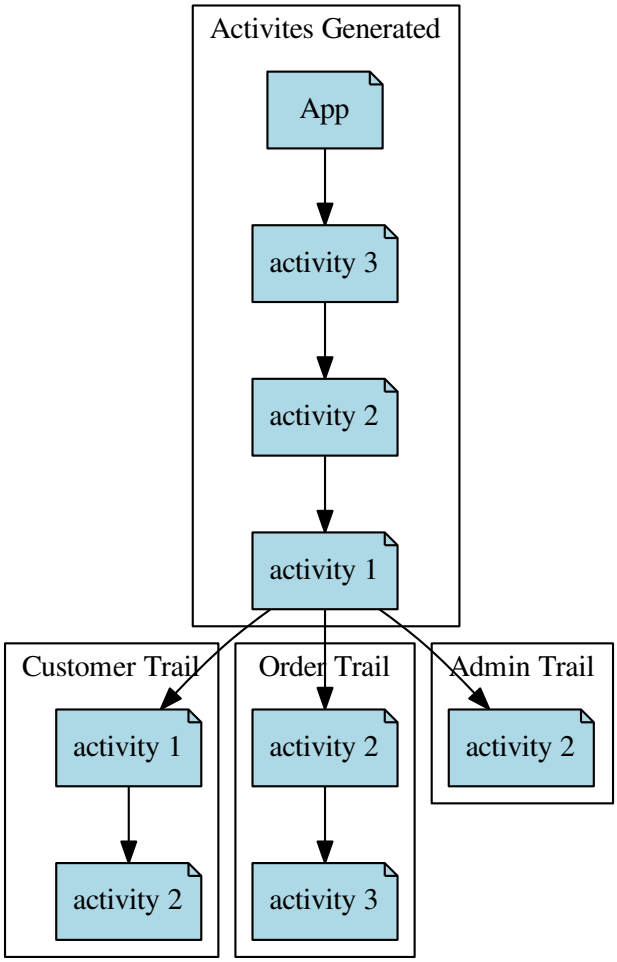
Because of the Event Sourcing architecture, customers can also build custom event consumers that respond immediately to changes in the database. This

can greatly simply integration and synchronization with 3rd party systems. For example synchronizing orders with an ERP and 3PL.



2.5 Activity Trail

In addition to monitoring changes in the database. The API Servers generate high level events called Activities. These Activities are linked to Activity Trails that belong to various objects in the system. You can monitor high level changes to all parts of the system including customers, orders, admins and products. You can see the same activity in different timelines. For example, if a customer changes the shipping address on their order, you will see the activity on the order's and customer's trails.



3 Scalability

Running a production system can be a significant cost. It is important that the system is designed to be the right size at the right time. The architecture needs to be designed so that the system can scale up and down without problems.

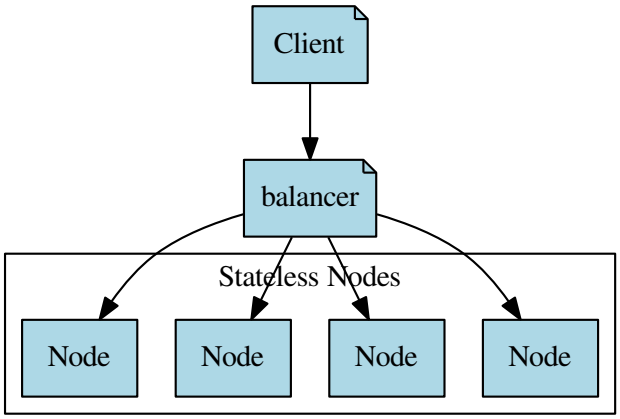
We designed the system to be scalable in several ways.

1. Stateless
2. Data Partitioning
3. Disjoint Parallelism

3.1 Stateless

A very important property of scalable systems is statelessness. The backend software is written in Scala and we don't allow mutable state to be stored within the program. All software serving requests to clients is stateless which allows nodes to be added or removed without affecting behaviour of the system.

This allows us to easily add a load balancer and then auto scale the machines based on utilization. We use few machines during slow times and scale up resources as demand increases. This means an overall cheaper system because there is less waste.



3.2 Data Partitioning

The data in the system is always growing and therefore it is important to partition the data. The FoxCommerce system has several stores of data and each of these is designed to be partitioned.

1. DB Sharding.
2. Event Store Partitioning.
3. Search Cluster Index Partitioning.

Each of these is partitioned so that as the data grows linearly, the query times stay consistent.

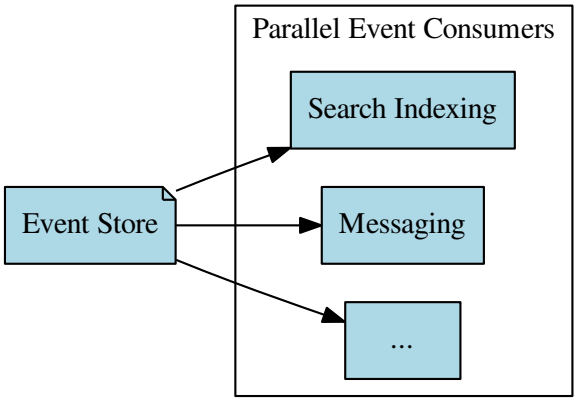
3.3 Read/Write Partitioning

Another important form of partitioning is separating where data is written to and where it is read from. Because data is indexed as soon as it changes, much of the frontend UI code reads from the search cluster instead of the DB. This means that the DB has less work to do which improves overall system performance.

3.4 Disjoint Parallelism

Just as the API servers nodes can be added or removed, the backend processing components need be added and removed without affecting system performance.

The Event Store is partitioned so that you can have many event consumers working in parallel without affecting each other. For example, the indexing and messaging can process the same event stream at the same time.



4 Data Model

An important technique in creating a good system architecture and data model is understanding the symmetry in the system and where the symmetry breaks. Where symmetry breaks is where there is something interesting happening. During the Big Bang, there was equal parts matter and dark matter. The symmetry of the universe somehow broke and all the dark matter went somewhere. Broken symmetry is why we are here today.

The FoxCommerce data model has a broken symmetry and is divided into two parts.

1. Transactions
2. PIM/Merchandising

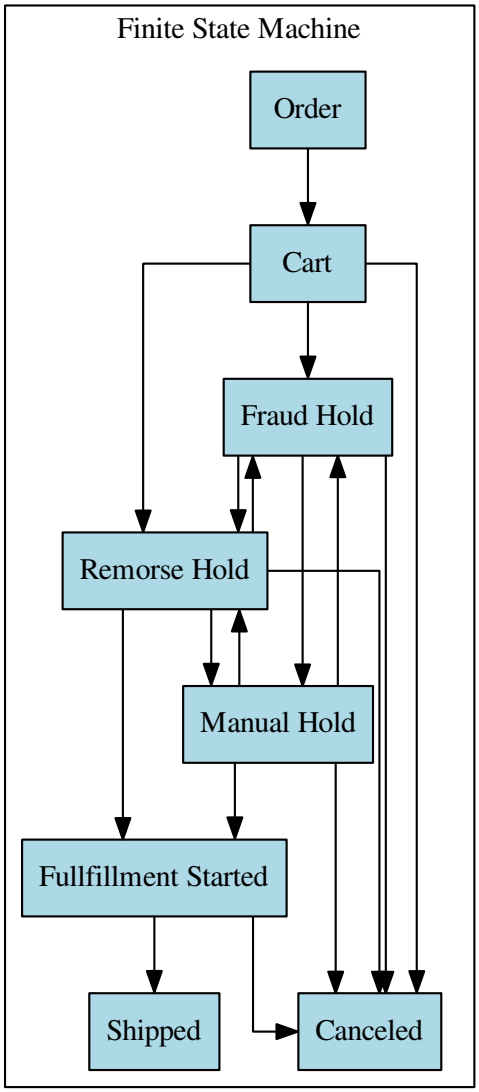
4.1 Transactions

Transactions are core to the business and it is important that the data is correct and the code is safe. Several techniques are used to maintain correctness and safety.

1. Strong Type System
2. Finite State Machines
3. Referential Integrity

The backend is written in Scala. For the transactional part of the system, the strength of Scala's type system is utilized to validate correctness. It is important that the transactional part of the system is consistent and that the data is always in a correct state. Strongly typed data structures are used to represent Orders and the possible states they can have.

Orders are modeled as a Finite State Machine where the valid state transitions are checked at compile time. We use referential integrity in our database model to make sure everything is structured correctly and that all relations are maintained. Pointing to non-existent data is not possible.


Finite State Machine
Order
Cart
Fraud Hold
Remorse Hold
Manual Hold
Fullfillment Started
Shipped
Canceled

4.2 PIM/Merchandising

While the transactions data model is built using principles of correctness and safety, the product and merchandising model is designed to be maximally flexible.

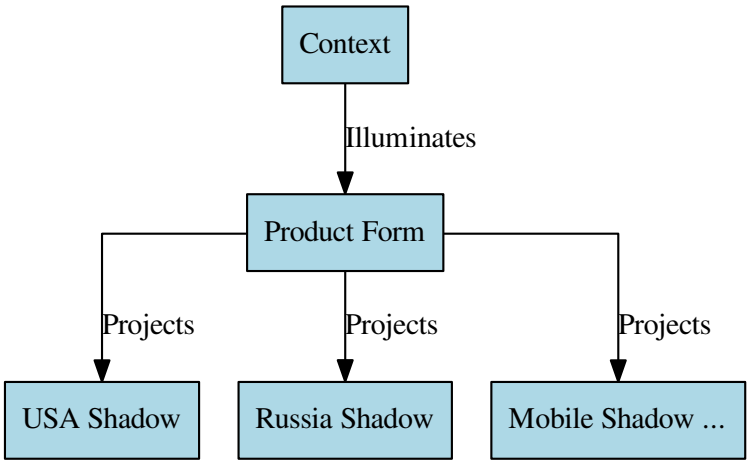
The PIM/Merchandising data model has three important components

1. Extensible Object Model
2. Versioning
3. Taxonomy

4.2.1 Extensible Object Model

The foundation of the product and merchandising model is a flexible object store where objects can have custom properties and dynamic associations. These objects are viewed from something we call a "Context". The Context is the lens you view the object with.

For example, a product might be viewable in a USA and Russia context. When you change the USA context, you see the product's USA specific properties, like an English description. When you switch to the Russian context, you would see the Russian description, and any other additional properties specific to Russia. The same object lives in both contexts. We call this model the Form/Shadow model.

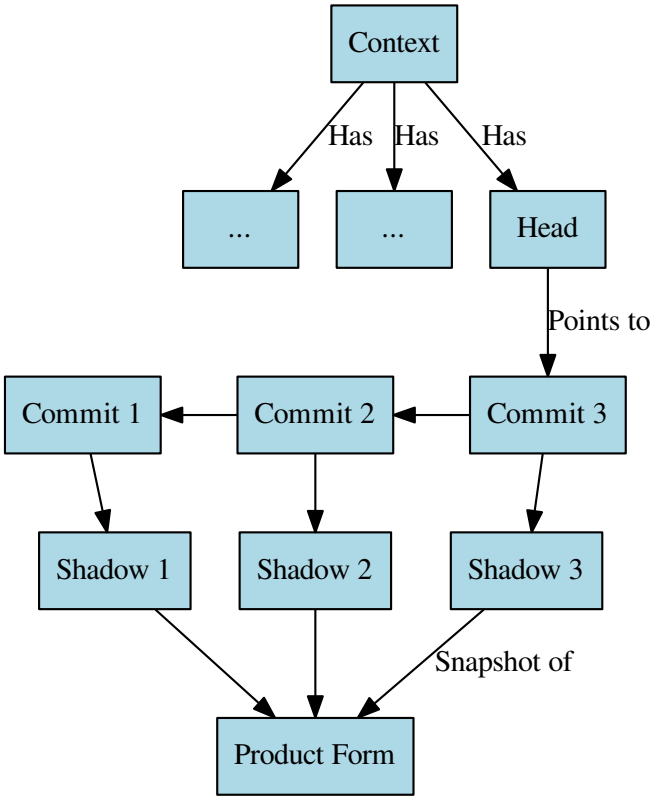


This model allows you to sanely manage merchandising various parts of the system to different groups of people.

4.2.2 Versioning

In addition, each change to any object in the merchandising system is versioned.

The versioning model is built on the Form/Shadow and Context model.



It allows the same object to have different versions in both time and context. This can enable very interesting workflows. For example, if every Admin had their own Context, they can make changes to a product and have the changes reviewed before they go live.

It can allow whole changes to a site while creating a holiday campaign. You can have the campaign go live and then switch back to the older version in the previous context.

4.2.3 Taxonomy

There is also a flexible model for organizing the objects. An object can be assigned various to taxonomies like categories and tags. These can be used to organize objects into groups and trees.

Since all data is indexed, groups can be defined as queries on the taxonomies and properties of objects. This allows you to have dynamic groups which change over time.

For example, you can have a group "top 10 popular sun glasses", and have it defined as a search query based on sales and taxonomy.