The Enlightened, Post-Modern Object Model

Maxim Noah Khailo

May 31, 2016

1 Purpose

There is both a universal objective reality and our subjective views of it. The post-modern world is one of flux and uncertainty which can only collapse into the shadow of objectivity when illuminated.

Below I will describe an object model. However, we don't sell object models, we sell use cases. Most importantly, I will also describe the use cases the object enables at the end.

This document is an evolution of thought as described in the "The Enlightened, Post-Modern Product Model" and the "Product Model Versioning". If you want to understand the evolution of though, you can read those first.

2 Data Model

2.1 Multiple Tops

When you view any software system from a top-down perspective, you will quickly realize there is no ONE top-down perspective. You can view the system from a maintenance perspective or a security perspective. You can view the system from an accountant's perspective, or a merchandising perspective. Every piece of data and line of code can be viewed from multiple perspectives and is part of all these different "Tops". This is why you can't organize a system with just one hierarchy.

Every piece of PIM and Merchandising data in the FoxCommmerce system has different meaning and purpose depending on perspective you take and task you're doing. All object's in the system can exist in multiple "Views" and may look different given the view you are in.

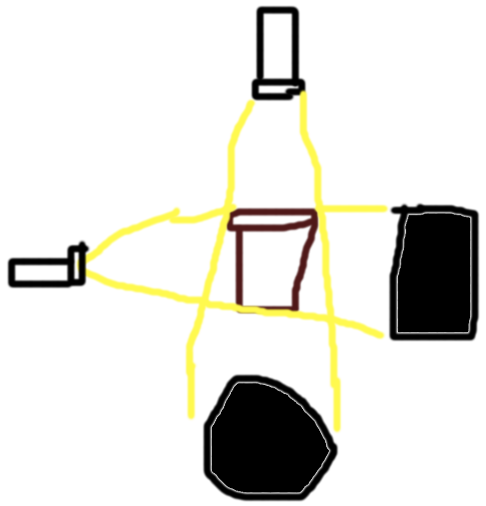
Views are stored as JSON and the system can start with a global view.

```
//A Possible View
{
    id: 3,
    name: 'global',
    language: 'en'
}
```

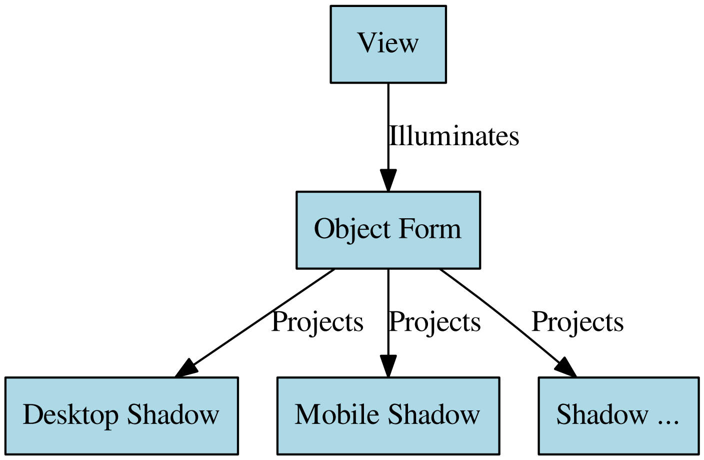
An object looks different depending on the View and we achieve this by splitting an object into two parts.

2.2 Form and Shadow

The object is split into two parts, the *Form* and the *Shadow*. An object has ONE form but can have MANY shadows. Shadows exist given a View. To understand this idea, imagine you are holding a cup and pointing a flashlight at it. When you point the light at the cup, it will cast a shadow of a rectangle on the wall. However when you take a different *view* from the top of the cup and point the light, you will cast a circle on the floor.



Another way you can view the relationship is.



So depending on the View, you will see a different Shadow.

2.3 Representing the Form and Shadow

We will store the Form as a JSON object with a flat collection of attributes The key of the attribute is a hash of the value.

```
//Object Form
{
    id: 3,
    kind: 'product',
    d5ae001: 'Just Red Shoe',
    c981c9f: 'Big Red Shoe'
}
```

To understand the shape of the Form requires a Shadow. The Shadow will be represented as a JSON object where each attribute has a type and a reference to the key hash in the Form.

```
//Object Shadow 1
{
    form: 3,
    name : { t: 'string', r: 'c981c9f'}
}
```

And of course the object may have another Shadow referencing a different hash.

```
//Object Shadow 2
{
    form: 3,
    name : { t: 'string', r: 'd5ae001'}
}
```

We can now “dereference” the object given a View by applying the View’s shadow to the Form. If we apply the first Shadow we will get...

```
//Applying Shadow 1 to the Form
{
    form: 3,
    name : { t: 'string', v: 'Big␣Red␣Shoe'}
}
```

If we apply the second Shadow we get...

```
//Applying Shadow 2 to the Form
{
    form: 3,
    name : { t: 'string', v: 'Just␣Red␣Shoe'}
}
```

It is called a “Shadow” because it only defines the shape of the object given a view, NOT the values. It references the values from the Form.

You can also organize the attributes in a Shadow under sections, for example.

```
//Attributes can be in multiple sections.
{
    form: 3,
    public: {
        general : {
            name : { t: 'string', r: 'c2134d34'}
            description : {t: 'string', r: 'dc3f74f'}
        },
        taxons : {
            category : 'cff545f',
            tags: 'a45d34b'
        }
    }
    private: {
        hiddenthing : { t: 'string', v: 'not␣indexed'}
    }
}
```

Notice that the “category” and “tags” attributes under the taxons section are just strings and don’t have a ‘t’ and ‘r’ key. If we omit the type and ref, the system will assume the key of the attribute is also the type.

```
// Can omit type if the type and key are the same.
{
    category : { t: 'category', r: 'cff545f'},
    category : 'cff545f',
    name: { t: 'string', r: 'c2134d34' },
    name: 'c2134d34'
}
```

Also basic types such as “string”, “number”, “array” can be derived from the type of JSON type stored in the Form. Defined types always override basic types, so if there is a type defined “name” as shown in the example above, then it will be the inferred type.

3 Versioning

In addition to representing changes of an object when a View changes, we want to remember changes over time. To handle versioning over time we will introduce two additional data structures.

1. Commit
2. Head

3.1 The Object Commit

The reason the keys in the Form are hashes of the values is to support versioning. You can think of a Shadow as a snapshot of the object Form. When a value in the object changes, the Form will get a new hash that the Shadow attribute will point to.

Since each Shadow is a "snapshot" of a Form, an ordered list of Shadows that point to the same Form would represent a history of changes of that object.

The Object Commit structure points to a Shadow, the Form, and a previous Commit, creating a linked list of Changes over time.

```
case class ObjectCommit(id: Int,
                        formId: Int,
                        shadowId: Int,
                        reasonId: Option[Int],
                        previousId: Option[Int],
                        createdAt: Instant)
```

3.2 The Object Head

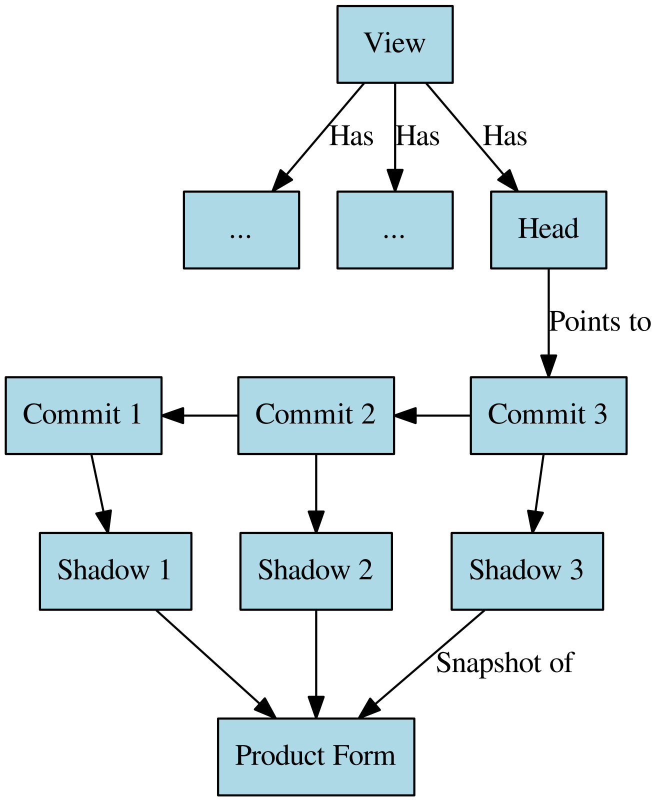
We now need to figure out which version an object is in a given View. We will create a pointer to a Commit called the Head.

The Head is what ties an object to a View.

```
calse class ObjectHead(id: Int,
                       viewId: Int,
                       shadowId: Int,
                       formId: Int,
                       commitId: Int,
                       kind: String,
                       updatedAt: Instant,
                       createdAt: Instant)
```

It also stores the *shadowId*, *formId* and *kind* here for convenience.

3.3 Versioning Structure



4 Higher Order Objects

There are many kinds of objects like products, SKUs, category pages, promotions. However, we can also represent higher order objects in our system using the same object model. This model is recursive and any features supported by this model also apply to the higher order objects.

4.1 Types

If you look at a Shadow attribute, it has a type field. We can use the same object system to define these types. For example, we can have a type representing review state of the object.

```
// Possible representation of Review State
{
    kind: 'type',
    name: 'reviewState',
    states : {
        t: 'option',
        v: ['Image', 'Description', 'Done']
    }
}
```

And then we can use this type inside another object

```
// Used by Product
{
    kind: 'product',
    name: { t: 'string', v: 'Awesome Product'},
    description: {t: 'string', v: 'Awesome Product'},
    review : { t: 'reviewState', v: 'Description'}
}
```

The front end UI can query the type “reviewState” from the object system to understand how to display the type and allow the user to select a value.

4.2 Taxonomies

It would be useful to organize objects under various taxonomies. There could be various kinds of taxonomies like categories and tags.

For example, we may want to represent a "sex" category which organizes products by male and female.

```
// Possible representation of a sex category taxon
{
    kind: 'taxon',
    name: 'sex',
    states : {
        t: 'option',
        v: ['Male', 'Female']
    }
}
```

And using it in a product can look like this...

```
// Possible representation of a sex category
{
    kind: 'product',
    general : {
        name: 'generic␣shirt',
    },
    taxons : {
        sex : 'female',
        tags : ['shirt', 'apparel']
    }
}
```

4.3 Collections

Since FoxCommerce indexes all data, we can index all objects in ElasticSearch. We can then have higher order objects called Collections which store a search query or explicit list of object ids and any other metadata

```
// Possible representation of a product collection.
{
    kind: 'collection',
    for: 'product',
    name: 'Collection of great products',
    description: '',
    query: 'magical ES query'
}
```

This approach has also the property that collections can be collections of other collections. Since collections themselves can be categorized with taxons and can have custom attributes.

4.4 Prototypes

We may want to define prototypes for certain kinds of objects. Prototypes are objects that have default attributes when you construct an object of the kind specified.

```
// Possible representation of a product prototype
{
    kind: 'prototype',
    for: 'product',
    general: {
        name: 'Generic Product Name',
        description: 'Fill Description Here',
    },
    media: {
        images: []
    }
    taxons : {
        tags: []
    }
}
```

It also makes sense to have many prototypes for the same kind of object.

5 Use Cases

Below are several use cases that are enabled with the model described.

5.1 Internationalization

You can have different language views of your objects providing support for internationalization for the storefront. The view changes given the user's language.

5.2 Staging

If every store Admin had their own View, they can modify products and other merchandising objects in their view before making changes live. Or there can be a general stage view where changes are reviewed before they go live.

5.3 Edge Commerce

You can use views to drive data displayed in secondary markets such as Amazon, Ebay, Twitter, and others. Imagine you have a Amazon view of a product, where attributes appropriate for the Amazon Marketplace exist.

5.4 Campaigns

Views can be used to create holiday based versions of data. For example, imagine a valentine view where category pages are modified with valentine specific imagery and special valentine promotions.

5.5 Business Specific Organization

Views allow different departments of a business to organize and manipulate the data based on their needs. Maybe the accounting department would want to organize and classify products based on different criteria than the marketing department.

5.6 Displaying Differences Between Versions

Versioning gives users the power to see all changes made to the system and the differences between two moments in time. They would also be able to see changes between objects in different Views.

5.7 Reverting Versions

Versioning allows users to revert changes that they didn't intend to make. There would be no limit in how far back they can go. This provides a sense of safety and security to users of the system.

5.8 Specifying Business Data

Each object is extensible with custom attributes. There should be maximum expressive power for businesses to define data on terms that makes sense to the customer. Domain experts in their business would be able to save all appropriate data using custom attributes.

5.9 Streamlining Creation of Merchandising Data

Custom types allow businesses to streamline the creation of merchandising data. Custom types allow a flexible UI for defining attributes. Along with custom types, prototypes further stream line the creation of data by specifying a default set of attributes and their values when a new object is created. Since each of these also can change depending on view, prototypes can adjust appropriately to business needs.

5.10 A/B Testing

Views can be used to create A/B testing. Views would allow an easy way to show different versions of merchandising data and associating it with collected stats.

5.11 Faceted Search

Taxons would provide a way to associate objects to defined classes. These taxons can be used directly in a faceted search system to filter category pages.