Client Consumption of SHOP.COM REST Services

Background

I have a standing item on my todo list to develop a Java client API for FAMOS - I call it JavaSHOP. For any page on FAMOS, we follow the same general workflow in our controllers:

1. Make REST service calls (usually several)
2. Adapt the data received so it matches how it's rendered
3. Render the view with the adapted data

It seems easy enough, but it has turned out pretty messy. For one thing, developers started instrumenting REST calls directly in application code as needed. For example, the product controller started out like this:

```
String serviceUrl = "/Store/Product/PC/" +
     prodContainerID + "?permKey=" + permKey +
     "&siteId=" + siteId;
ObjectNode product = restTemplate.getForObject(serviceUrl,
     ObjectNode.class);
```

There are a number of problems with these two simple lines of code:

- If the same data is needed elsewhere these lines are repeated (boilerplate)
- Application code shouldn't have to know the details of making REST calls (too low-level)
- One-off generation of URLs using string manipulation
- Binding to ObjectNode, which forces the rest of the request processing to deal purely with strings (this is Java!)
- There is zero error checking/handling on the REST response

We pretty quickly abstracted the REST calls into a simple API that binds to real Java objects - the same ones used by the REST services themselves (amos_service_model.jar). Now to get product data, it's as simple as this:

```
Product product = shopAPI.getProduct(prodContainerID);
```

The controller and the velocity template can deal with strongly typed Java objects, which is appropriate because they are both Java technologies. We still don't have a very good error checking/handling strategy, but it will be added as time permits.

So far so good, but there is still a lot of spaghetti-ish code that prepares the data received from the REST services for display. When we first started working with the REST services, the responses were tailored for display on the SHOP.COM website, but even so the responses do not map 1:1 to elements on the UI, and as the UI gets redesigned (a lot) the REST data diverges even more with display requirements.

As a result, we end up writing a lot of code - sometimes many lines - either in the controller or in the velocity template to create new objects or otherwise process a REST response before rendering it. For example, look at this code in our product template that processes the image URLs before displaying the main product image:

```
<div class=”product-image”>
     #set($largeImageURL = $productContainer.imageURL)
     #set($largeImageHeight = $productContainer.imageHeight)
     #set($largeImageWidth = $productContainer.imageWidth)
     #foreach($alternateImage in
     $productContainer.alternateImages)
          #if ($alternateImage.isLargeView == true)
               #set($largeImageURL = $alternateImage.url)
               #set($largeImageHeight =
               $alternateImage.height)
               #set($largeImageWidth = $alternateImage.width)
               #break
          #end
     #end
     <img src=”$largeImageURL”... />
</div>
```

This is where we are today in FAMOS...

Other Clients + One Size Fits All

FAMOS was the first client for our REST services, but there are now other clients:

- Motives website
- Variety of mobile apps
- Surely more to come...

There is a lot of discussion in meetings about clients wanting data from the REST services in forms that are optimized for their specific display requirements. In particular, it is desired to minimize payload size in order to increase performance and throughput. A client doesn’t want to call a heavyweight service when all they need is a small piece of the response.

It soon becomes clear that a One Size Fits All REST API is difficult at best. The backend team would have to meet requirements for all clients with a single API. The clients will always have to do some amount of work to get the data into a form that fits their requirements. Depending on the situation, it could be a little work, or a lot of work, and clients would need to keep up with any changes in the REST API.

Another approach is to develop unique endpoints for each client that composes the data into a usable form just for them. We started going down this path with the page service. Its purpose is to compose REST data needed to display a page on the SHOP.COM website (home page, search page, product page, etc). If this is done so the data matches 1:1 with the UI, it GREATLY simplifies the application code. The product controller (or any other controller) is reduced to this:

```
@RequestMapping(value=”/{prodContainerID}-p.xhtml”,
      method=RequestMethod.GET
public String handleProductPage(ModelMap modelMap,
                              @PathVariable(“prodContainerID”) {
      ProductPage productPage =
            shopAPI.getProductPage(prodContainerID);
      modelMap.put(productPage);
      return “productView”;
}
```

Similarly there should be no logic in the velocity template (“productView.vm”) to massage the productPage object. Instead it is rendered directly like this:

```
<div class=”product”>
      <p class=”caption”>$productPage.caption</p>
      <img src=”$productPage.imageURL”/>
      …
</div>
```

Boom. Poetry in code. So what have we done? We provided a “super service” (client endpoint) that is composed of many “micro services” put together in a way to display a specific page on the SHOP.COM website without manipulating data. We took logic from the client (consider the controller and the velocity template as the FAMOS client) and moved it server-side.

There are still some issues though. This works great for SHOP.COM, but what about other clients? It’s probably not practical (or reasonable) for the backend team to understand display requirements for each client and develop a custom optimized endpoint. In addition, it places a dependency on the backend team for client-side development, and the client loses control over how the data is returned to them.

Another question is where the client endpoint lives (architecturally). It could be exposed as another REST service, it could be moved to an “API gateway”, or it could simply be a module inside a monolithic server program (which is how FAMOS is doing it now).

No matter what, it seems to make sense that client developers should be responsible for creating and maintaining their own endpoint. That way they control their own interface, can do any optimization for their specific display requirements, and control their own schedule and dependencies. An added benefit is the work is distributed more evenly all around. The backend team can worry about developing granular generic micro services and the clients can worry about how they're going to put them together in an optimal way for their display.

Architectural Approaches

Some brief research around the Internet shows the One Size Fits All REST API problem is very common. All of the "big guys" (e.g., Amazon, Netflix, Facebook) have dealt with it. This site has good descriptions of the architectural patterns used around REST APIs. We are proceeding down the same paths:

http://microservices.io/

Netflix are perhaps the most vocal about it. In the last year they re-vamped their microservice architecture into an API gateway model where the various UI teams develop and control their own endpoint for their display device. In fact, Netflix turned off their public REST API today (11/14/2014) and are limiting access to their backend services through the API gateway to internal UI teams and partners.

Here are some interesting articles from the Netflix Tech Blog:

http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html

http://techblog.netflix.com/2011/02/redesigning-netflix-api.html

http://techblog.netflix.com/2013/01/optimizing-netflix-api.html

The key points are:

- Data collection is executed and composed server-side inside the same network (not necessarily on one server)
- Each request from a client is turned into several lower level REST calls and composed in single response to the client
- Clients develop, control, optimize and maintain their own endpoints
- Parallelism is required to meet performance objectives

The concurrency aspect is especially interesting. Netflix (and now many others) are using a Reactive Programming approach:

http://en.wikipedia.org/wiki/Reactive_programming

Netflix is a Java house, so they did a Java port of Microsoft's open source Reactive Extensions:

https://rx.codeplex.com/

Netflix recently open sourced their version (RxJava):

https://github.com/ReactiveX/RxJava/wiki

Rx makes it super easy to put together Observable tasks in various ways. For example, suppose we make a REST request that returns HATEOAS links for related data. With Rx, you might wait for the main call to return then fork off asynchronous tasks to fetch the rest of the data. This paradigm has proven to be very effective.

All of the above assumes the client endpoints reside server-side. It is certainly one effective approach. In light of mobile initiatives, a mobile app would make a single request to its endpoint and receive back an optimized payload for rendering.

It doesn't have to be that way though. The reactive programming model is also being used directly on iOS and Android to solve these same problems. Here are a few articles:

http://joluet.github.io/blog/2014/07/07/rxjava-retrofit/

http://www.raywenderlich.com/62699/reactivecocoa-tutorial-pt1

http://www.raywenderlich.com/62796/reactivecocoa-tutorial-pt2