Design notes for the Mercador API

High level:

There are three API interactions that we need to design:

1. CLI to Publisher service (used by the Publisher CMS admin to manage virtual regions and subscriptions)
2. CLI to Subscriber service (used by the Subscriber CMS admin to manage virtual region subscriptions)
3. Subscriber service to Publisher service

The following principles (or possibly prejudices) have been applied in the design.

- Some of the API interactions look very similar to each other, and it's tempting to collapse them into a single API. Please resist this temptation until we've finished. No premature optimization.
- All objects are encoded as JSON.
- The RESTful APIs follow the HATEOAS principle. All CREATEs take an object representation, and (if successful) return a **Location** header with the URL (not simply the ID) of the created resource.
- For the most part, resources are normalized. We're not afraid of joins.
- Resources are designed for maximum cacheability. Subscriptions are expected to be long-lived relationships.
- The cacheability principle favors the separation of immutable from volatile resources. (This teds to follow from normalization.)
- For simplicity, we do not want services to have to perform elaborate filtering of resources depending on the identity of the requester. RBAC should apply to a resource, rather than to individual resource elements (or to a materialized view). If some information is public, and other information is private to a particular requester, we prefer to use two different resources.
- These interactions are infrequent, administrative actions. Making two requests to retrieve the immutable and volatile data is not a big deal.
- Filters, selectors, pagination, etc. are all specified via query parameters, not by extended URLs.

1. CLI to Publisher

These APIs are used by the publisher admin to manage (CRUD) a virtual region, and to list (R, optionally filtered) a set of virtual regions. `{pep}` is the Publisher endpoint.

```
POST https://{pep}/v1/vregion/  <  {vregion JSON} > {Location:
https://{pep}/v1/vregion/{uuid}  }
UPDATE https://{pep}/v1/vregion/{uuid}  < {vregion JSON}
GET https://{pep}/v1/vregion/{uuid}  >  {vregion JSON}
DELETE https://{pep}/v1/vregion/{uuid}
GET https://{pep}/v1/vregion[?filter]  >  {List of {vregion
```

```
JSON}}
```

Open design decision:

We have two ways of implementing admission control for subscriptions. The simplest (and most OpenStackish way) is to use RBAC and Keystone, so that a subscriber has to be able to authenticate with the local Keystone. An alternative is to actually require the publisher admin to set up a subscriber object. There are good operational reasons for this: we may want to make sure that OSS, BSS, and other non-OpenStack mechanisms are set up before we actually start publishing and subscribing to Virtual Regions. Also, when a subscription is cancelled, we may need some local publisher-side state about the former subscriber.

Let's go with the second model. It will affect the APIs in various, mostly minor, ways. Note that all we're doing is manipulating local state: there is no communication between Subscriber and Publisher involved in any of these calls.

```
POST https://{pep}/v1/subscriber/  <  {subscriber JSON} >
{Location: https://{pep}/v1/subscriber/{uuid}  }
UPDATE https://{pep}/v1/subscriber/{uuid}  < {subscriber JSON}
GET https://{pep}/v1/subscriber/{uuid}  >  {subscriber JSON}
DELETE https://{pep}/v1/subscriber/{uuid}
GET https://{pep}/v1/subscriber[?filter]  >  {List of
{subscriber JSON}}
```

The publisher admin also needs to be able to view the state of subscriptions. There's no CREATE or DELETE, because those are handled by the Subscriber-to-Publisher interactions. (To provide some publisher control, we'll introduce a `subscriptionstate` and some lease renewal controls.)

```
UPDATE https://{pep}/v1/subscription/{uuid}  < {subscription
JSON}
GET https://{pep}/v1/subscription/{uuid}  >  {subscription JSON}
GET https://{pep}/v1/subscription[?filter]  >  {List of
{subscription JSON}}
```

2. CLI to Subscriber

These APIs are used by the subscriber admin to manage virtual region subscriptions. Here, `{sep}` is the Subscriber endpoint.

We're going to allow the subscriber admin to define a set of publishers. In part, this is because it's likely that a subscriber may access multiple virtual regions (simultaneously or serially) from a given publisher. In addition, we may need to establish external (non-OpenStack) service couplings before we can initiate virtual region subscriptions. Part of the publisher object is an endpoint, which the Subscriber service will use to construct

URLs for Subscriber-to-Publisher interactions. (Again, note that all we're doing is manipulating local state: there is no communication between Subscriber and Publisher involved in any of these calls.)

```
POST https://{sep}/v1/publisher/  <  {publisher JSON} >
{Location: https://{sep}/v1/publisher/{uuid}  }
UPDATE https://{sep}/v1/publisher/{uuid}  < {publisher JSON}
GET https://{sep}/v1/publisher/{uuid}  >  {publisher JSON}
DELETE https://{sep}/v1/publisher/{uuid}
GET https://{sep}/v1/publisher[?filter]  >  {List of {publisher
JSON}}
```

There is one method provided to allow a Subscriber to "ping" a Publisher. It instructs the Subscriber service to invoke the corresponding operation from Subscriber to Publisher (described below).

```
GET https://{sep}/v1/publisher/{uuid}/publisherstatus  >
{publisherstatus JSON}
```

The Subscriber admin needs to be able to see what virtual regions have been defined by the Publisher:

```
GET https://{sep}/v1/publisher/{uuid}/vregion/{uuid} > {vregion
JSON}
GET https://{sep}/v1/publisher/{uuid}/vregion[?filter] > {list
of {vregion JSON} }
```

Each of these calls is implemented using the corresponding Subscriber-Publisher API call.

The key operation for a Subscriber admin is to create a subscription to a Provider's virtual region. Once created, it's not immediately available for use: we're going to separate the two steps of subscribing and advertising (i.e. pushing the subscription info to the local Keystone, Horizon, etc.). For simplicity, the `subscription` object should be the same for both Publisher and Subscriber, but this requires the Subscriber admin to do a lot of work in constructing the `subscription` object. This is because if we're following HATEOAS, we need full URLs for all of the resource references. More anon. The subscription request includes a reference to the Subscriber's local Keystone, to allow the Publisher to set up Keystone-to-Keystone authentication.

```
POST https://{sep}/v1/subscription/  <  {subscription JSON} >
{Location: https://{sep}/v1/subscription/{uuid}  }
UPDATE https://{sep}/v1/subscription/{uuid}  < {subscription
JSON}
GET https://{sep}/v1/subscription/{uuid}  >  {subscription JSON}
DELETE https://{sep}/v1/subscription/{uuid}
GET https://{sep}/v1/subscription[?filter]  >  {List of
```

```
{subscription JSON}}
```

Once a subscription has been created, the subscriber admin can advertise it. The Subscriber service implements the advertise function by updating the local Keystone (and Horizon) configuration to reflect a new region that maps to the subscription, and which inherits project/domain and endpoint information. The advertisement object includes the subscription reference, and also the name by which the virtual region should be known to local users.

```
POST https://{sep}/v1/advertisement/  <  {advertisement JSON} >
{Location: https://{sep}/v1/advertisement/{uuid}  }
UPDATE https://{sep}/v1/advertisement/{uuid}  < {advertisement
JSON}
GET https://{sep}/v1/advertisement/{uuid}  >  {advertisement
JSON}
DELETE https://{sep}/v1/advertisement/{uuid}
GET https://{sep}/v1/advertisement[?filter]  >  {List of
{advertisement JSON}}
```

The final part of the CLI to Subscriber API is a convenience function (and we may choose not to implement it in the first version). Recall that when the Subscriber calls the Publisher to create a subscription, one of the elements that is returned is a user identity, scoped to the Publisher's Keystone, which has domain admin rights over the domain which corresponds to the virtual region. The subscriber will not provision resources directly into that domain; instead it will create a sub-project in the domain (virtual region) on behalf of a local user. This project creation operation needs to be invoked with the credentials of the domain admin user supplied by the Publisher. For convenience, let's define an API to implement the workflow of creating that initial project, using the correct credentials. The request takes as arguments the virtual region name, the local user who will own the project, and the project name.

[Details TBD]

3. Subscriber to Publisher

These APIs are used by the Subscriber service to interact with the Publisher service. Here, `{pep}` is the Publisher endpoint.

There is one method provided to allow a Subscriber to "ping" a Publisher.

```
GET https://{pep}/v1/publisherstatus  >  {publisherstatus JSON}
```

The Subscriber needs to be able to see what virtual regions have been defined by the Publisher:

```
GET https://{pep}/v1/vregion/{uuid} > {vregion JSON}
```

```
GET https://{pep}/v1/{uuid}/vregion[?filter] > {list of {vregion
JSON} }
```

Note that if we compare the CLI to Subscriber API calls with the Subscriber to Publisher methods, the URL structure is different. This is because in the first case we need to identify the Publisher; in the second, we can derive the Publisher's endpoint from the `publisher` object.

The key operation for a Subscriber s to create a subscription to a Provider's virtual region. Once created, it's not immediately available for use: the subscriber admin needs to advertise the subscription. The subscription request includes a reference to the Subscriber's local Keystone, to allow the Publisher to set up Keystone-to-Keystone authentication.

```
POST https://{pep}/v1/subscription/  <  {subscription JSON} >
{Location: https://{pep}/v1/subscription/{uuid}  }
UPDATE https://{pep}/v1/subscription/{uuid}  < {subscription
JSON}
GET https://{pep}/v1/subscription/{uuid}  >  {subscription JSON}
DELETE https://{pep}/v1/subscription/{uuid}
GET https://{pep}/v1/subscription[?filter]  >  {List of
{subscription JSON}}
```