Service Discovery and Binding

This is a proof of concept description of only one portion of the API Extension (API-X) Framework: the service discovery and binding component (SD&B). This document makes reference to other other components in API-X, not in order to define how those other components work, but simply as possible ways in which the SD&B component might interact with the larger API-X framework.

Authorization/Authentication of API-X and any registered service is beyond the scope of this document, though API-X should support these in some manner.

Background

Clients interacting with the API Extension framework will need a mechanism to discover the services that apply to a given repository resource. Likewise, services themselves will need a mechanism by which they can register or bind themselves to the API-X framework.

A principal role of the SD&B component is to support an architecture that recognizes that services come and go, sometimes unexpectedly. To that end, it should be possible to decouple the lifecycle of a particular service instance from the lifecycle (i.e. deployment) of the API-X framework, including the SD&B component. Furthermore, it should be possible to deploy this component in a distributed fashion across multiple machines, both to support high availability and high levels of concurrency. It should also be possible for services to be deployed on an arbitrary number of external hosts using any language or framework. With that structure, network partitions and service failure should not affect the overall operation of this component nor the overall operation of API-X.

In many ways, the SD&B component can be thought of as a management interface, distinct from individual service endpoints. While its role is not that of operating on specific repository resources, it can be viewed as a broker between clients, repository resources and external services.

The high level objectives of such a management interface are to support the following:

- Service Discovery (i.e. client interaction):
    - list all available services
    - list all services that apply to a given fedora object
    - list all services that apply to a given rdf:type of fedora object
    - list service status (availability/non-availability)
    - provide some level of description of services (e.g. as RDF)
    - use REST semantics
- Service Binding (i.e. service interaction)
    - Services should be able to register and deregister themselves from API-X
    - It should be possible for individual services to be available at N hosts (e.g. for high availability)
    - If a particular service instance fails or is removed, API-X should know about that (optional)
- Deployment
    - the SB&D component should be capable of being deployed in a fully distributed environment, across multiple hosts, and such deployment should be entirely transparent to clients.
    - it should be possible for the SD&B interface to be deployed on separate hosts from the services themselves.

Reverse Proxy

A related concept to SD&B is that of a reverse proxy. The design details of that are out of scope for this document, but a possible outline is described in order to provide more context to the SD&B component. At a high level, a client using the API-X proxy could interact with a Fedora repository as if there were no proxy at all. The proxy may choose to add headers such as (e.g. for the resource `/rest/resource`):

```
Link: <http://localhost:8080/rest/resource/svc:list>; rel="service"
Link: <http://localhost:8080/rest/resource/svc:validate>; rel="service"
Link: <http://localhost:8080/rest/resource/svc:ldcompact>; rel="service"
Link: <http://localhost:8080/rest/resource/svc:ldpath>; rel="service"
```

These headers would be generated using the SD&B interface. Then, when a client interacts with a service, e.g. at `/rest/resource/svc:validate` the proxy mechanism will pass the request directly to an instance of that service, using the context of `/rest/resource`. In this way, clients should have no need to interact directly with the SD&B component.

Still, there are cases where this higher level interface is not sufficient. For example, some services may require a raw TCP socket or some other non-HTTP-based interaction with a service. In order to also support those cases, it is recommended that the service discovery interface (or some portion thereof) be exposed directly to clients so that clients can connect and interact directly with external services (unmediated by the API-X proxy mechanism). These are described in the section on "Client Endpoints".

Endpoints

There are two categories of endpoints: those used by clients and those used by services. In these examples, all data exchange uses JSON-LD. These examples refer to a JSON-LD context such as the following:

apix.jsonld

```
{
    "@context": {
    "id" : "@id",
    "type" : "@type",

    "apix" : "http://fedora.info/definitions/v4/apix/",
    "rdfs" : "http://www.w3.org/2000/01/rdf-schema#",
    "dcterms" : "http://purl.org/dc/terms/",
    "fedora" : "http://fedora.info/definitions/v4/repository#",

    "Binding" : {"@id" : "apix:Binding", "@type" : "@id"},
    "Registry" : {"@id" : "apix:Registry", "@type" : "@id"},
    "Service" : {"@id" : "apix:Service", "@type" : "@id"},
    "ZooKeeperBinding" : {"@id" : "apix:ZooKeeperBinding", "@type" : "@id"},

    "hasEndpoint" : {"@id" : "apix:hasEndpoint", "@type" : "@id"},
    "hasParentZnode" : {"@id" : "apix:hasParentZnode"},
    "hasService" : {"@id" : "apix:hasService"},
    "hasZooKeeperEnsemble" : {"@id" : "apix:hasZooKeeperEnsemble", "@type": "@id"},
    "supportsType" : {"@id" : "apix:supportsType", "@type": "@id"},
    "seeAlso" : {"@id" : "rdfs:seeAlso", "@type" : "@id"},
    "label" : {"@id" : "rdfs:label"},
    "comment" : {"@id" : "rdfs:comment"},
    "identifier" : {"@id" : "dcterms:identifier"}
  }
}
```

This context file implies the existence of a defined API-X ontology, which is not defined here.

Client Endpoints

All client endpoints use HTTP REST semantics.

Discovering Services

Request:

```
GET /apix/registry
```

Request Parameters:

These optional parameters will filter the service list to include only those services that (a) can be applied to the provided Fedora resource, if defined and (b) can be applied to the provided `rdf:type` URIs. In the case of multiple `rdf:type` URIs, a boolean `AND` operator is assumed.

`id` - a particular Fedora resource
`type` - a comma-delimited list of `rdf:type` URIs

Response:

```
Content-Type: application/json
Link: <http://fedora.info/definitions/v4/apix.jsonld>; rel="describedby"; type="application/ld+json"
{
  "id" : "http://apix-host/apix/registry",
  "type" : "Registry",
  "hasService" : [
    {
      "type" : "Service",
      "label" : "a foo webservice",
      "seeAlso" : "http://example.org/foo",
      "identifier" : "foo",
      "supportsType" : ["fedora:Resource"],
      "hasEndpoint" : ["http://host-1/foo/rest", "http://host-2/foo/rest"]
    },
    {
      "type" : "Service",
      "label" : "a bar webservice",
      "seeAlso" : "http://example.org/bar",
      "identifier" : "bar",
      "supportsType" : ["fedora:Binary"],
      "hasEndpoint" : ["http://host-3/bar/rest", "http://host-4/bar/rest"]
    }
  ]
}
```

In a similar way, information about a particular service can be retrieved:

```
GET /apix/registry/foo
```

Response:

```
Content-Type: application/json
Link: <http://fedora.info/definitions/v4/apix.jsonld>; rel="describedby"; type="application/ld+json"
{
  "id" : "http://apix-host/apix/registry/foo",
  "type" : "Service",
  "label" : "a foo webservice",
  "seeAlso" : "http://example.org/foo",
  "identifier" : "foo",
  "supportsType" : ["fedora:Resource"],
  "hasEndpoint" : ["http://host-1/foo/rest", "http://host-2/foo/rest"]
}
```

Service Endpoints

Registering Services

Services can be registered by interacting with the service registry. This endpoint only registers the existence of a service but does not make any guarantees about any running instances of that service. Such a service must also first be registered before any service instances can be bound to it.

```
PUT /apix/registry/foo

Content-Type: application/ld+json
{
  "@context" : "http://fedora.info/definitions/v4/apix.jsonld",
  "id" : "http://apix-host/apix/registry/foo",
  "type" : "Service",
  "label" : "a foo webservice",
  "seeAlso" : "http://example.org/foo",
  "identifier" : "foo",
  "supportsType" : ["fedora:Resource"]
}
```

Note: the `hasEndpoint` element is not included here, but is part of the `/bind` interface, described below.

In a similar way, services can be de-registered. Any service instances bound to that service will be unbound, but this operation does not make guarantees about shutting down any instances of the service (which may be running on separate machines).

```
DELETE /apix/registry/foo
```

Service Binding

Before a client can interact with a particular service, that service must first be registered. In addition, one or more *instances* of that service must be *bound* to the API-X registry. Service binding can happen over HTTP or over another protocol defined by the implementation. These examples will use the ZooKeeper protocol for dynamic service binding, but other implementations could use a different binding protocol.

Manual binding

Some services may not be able to use dynamic service binding, e.g. a PHP web-application. For these, a manual binding interface is available. This example binds a particular service instance to the already-registered `foo` service.

```
POST /apix/bind/foo

Content-Type: text/plain
http://host-1/foo/rest
```

The response will contain a unique id of this service binding. That URI can be used to unbind the service at a later point.

```
204 Created

http://apix-host/apix/bind/foo/some-id
```

Manually unbind a service instance:

```
DELETE /apix/bind/foo/some-id
```

Dynamic Binding

Depending on the implementation, it may be possible to dynamically bind/unbind services. For instance, with ZooKeeper, a service may communicate directly with a zookeeper ensemble. The dynamic binding protocol is described at this endpoint:

```
GET /apix/bind/foo
```

Response:

```
Content-Type: application/json
Link: <http://fedora.info/definitions/v4/apix.jsonld>; rel="describedby"; type="application/ld+json"

{
  "id" : "http://apix-host/apix/bind/foo",
  "type" : ["Binding", "ZooKeeperBinding"],
  "hasZooKeeperEnsemble" : ["host-1:2181", "host-2:2181", "host-3:2181"],
  "hasParentZnode" : "/service/foo"
}
```

At this point (interacting directly with zookeeper), it would be the responsibility of the client to create an ephemeral, sequential znode under `/service/foo`, storing the value of the service's endpoint. For example:

```
create("/service/foo/instance-", "http://host-1/foo/rest", null, EPHEMERAL_SEQUENTIAL)
```

Service Availability

In these examples, when clients request a list of available services, that list will contain `hasEndpoint` values corresponding to the service instances that have been bound to API-X. For manually-bound services, those endpoints will continue to be included until they are manually un-bound. For dynamically-bound services, any interruption in the availability of the service (restarts, network partitions, host failure, etc) will cause the `hasEndpoint` value to disappear.

Distributed Shared State

The API-X architecture should support a distributed deployment model. As such, in a distributed context, shared state of the service registry must be managed carefully. ZooKeeper is one obvious choice for this, as it avoids creating a single point of failure. If that is not a concern, a shared database would accomplish the same thing. There are two types of shared data that each node of the API-X discovery service will need to have access to:

- Basic configuration information about the cluster (list of nodes, etc)
- Descriptions of each service (see the `/apix/registry` endpoint above)

- For each registered service, a list of each active service instance and the corresponding HTTP endpoint

Otherwise, no additional shared state should be maintained by the API-X SD&B component.