Discovering Interoperative Services for Continuous Operation (DISCO)

Mike Amundsen
<mca@mamund.com>

Table of Contents

Status

Status
: **Working Draft** — *Only experimental and 'proof-of-concept' apps should be built on this unstable draft.*

Repository
: https://github.com/open-disco/open-disco-docs

Email List
: https://groups.google.com/forum/#!forum/open-disco

Last Updated
: 2019-09-27

Summary

This document details an *adhoc specification* called **DISCO** (**D**iscovering **I**nteroperative **S**ervices for **C**ontinuous **O**peration). DISCO is a simple language for managing the adding/removing of services as well as the ability to search ("find") and make connections with ("bind") registered services.

DISCO was designed to be easy, open, lightweight, and extensible. For this reason, readers/implementers may find things "missing" or "underspecified." This is intentional. Getting started is meant to be easy. And local customization is supported as needed. This allows the DISCO spec to safely grow and improve over time without breaking exsting implementations.

The DISCO "language" supports the following features:

- `register` : add a service to the shared registry
- `find` : query the registry for services (dependents) to consume
- `bind` : notify the registry the intention to connect with and use another service
- `renew` : renew a service's registry *lease* to prove is is still up and running
- `unregister` : remove a service from the registry

Optionally, services can expose a `health-check` URL to allow registries to periodically check the status of a registered service.

Implementing a DISCO server is a simple as exposing URLs that support each of the above actions and supporting the defined input and output arguments in one or more identified media types.

Implementing support for DISCO in a service (or service proxy) can be accomplished by adding method to an individual service that handle the `register`, `unregister`, and `renew` or `health-check` actions. Optionally, services can add support for the `find` and `bind` actions if they wish to use the registry to locate other dependent services they consume.

DISCO Resources

The following resources are available for those interested using, and contributing to, the DISCO specification:

- DISCO Registry (NodeJS)
- DISCO Master Repo (NodeJS & HTML)
- DISCO Specification (ASCIIDoc)

> **Tip** If you are working on a DISCO-related project and would like to have it listed here, add an issue in the DISCO Specification repo and we can update the doc.

DISCO Design Goals

Here are some key design goals of the DISCO specification:

DISCO is Easy
: The goal of DISCO is to make it safe/cheap/easy to create connections between APIs *at runtime*. Essentially, automating the process normally handled by humans (discovery at design/implementation time) and supported by CI/CD tools (resolution at build/deploy time).

DISCO is Open
: DISCO was also created to foster an open approach to service discovery and management. For this reason, DISCO has been implemented to run over HTTP returning standardized message formats (HTML, JSON, etc.).

DISCO is Lightweight
: DISCO is not meant to create a close-loop, heavy-weight approach to managing services. It follows the same ethos as DNS, HTTP, etc. by doing only the minimum needed to accomplish a goal and to make no assumptions about the hardware, software, platform, or languages used to build and interact with DISCO servers.

DISCO is Extensible

DISCO was designed to be highly extensible. As long as implementers provide the required minimum service actions and support the defined semantic profile, they are free to add any number of optional features (as long as they don't break the defined DISCO profile).

DISCO is for the Web
: There are some very powerful targeted platforms for managing dynamic machine instances and clusters and they work especially well for 'closed communities' such as enterprises and large companies. DISCO was not created to compete with them. DISCO was designed to offer similar methods on the Web itself.

DISCO is for Services, Not Machines
: The target audience of the DISCO standard is the API/service developer community, not the DevOps VM/machine-instance community. DISCO allows you to register a running service, find other registered services you want to interact with, and keep track of the health of services on the Web.

Service-Side Implementation for NodeJS

As part of the DISCO release, a simple NodeJS module (`discovery.js`) is available. This module can be added to any existing NodeJS project and supports all the required and optional DISCO interactions.

> **Note** For more on the `discovery.js` module, see the DISCO Examples repo.

After including the module in your NodeJS project (and the accompanying `discovery-settings.js` configuration file), you can add default support for DISCO by wrapping your NodeJS `httpServer` like this:

Basic DISCO Support in NodeJS

```
// register this service w/ defaults
discovery.register(null, function(response) {

  // sample service discovery action
  discovery.find(null, function(data, response) {

    // select endpoints from query
    if(data.success===true) {
      // launch http server
      http.createServer(zipServer).listen(8080);
      console.info('zip-server running on port 8080.');
    }
    else {
      console.error('unable to bind to dependent services');
      process.kill(process.pid, "SIGTERM");
    }
  });
});
```

The DISCO Registry Specification

The DISCO Registry Specification describes the actions (required and optional) as well as all the input and output parameters for each action. The prototype DISCO Registry is implemented over HTTP and generates both HTML and JSON responses.

> **Tip** You can find the source code for the prototype DISCO Registry server in the DISCO Registry repo.

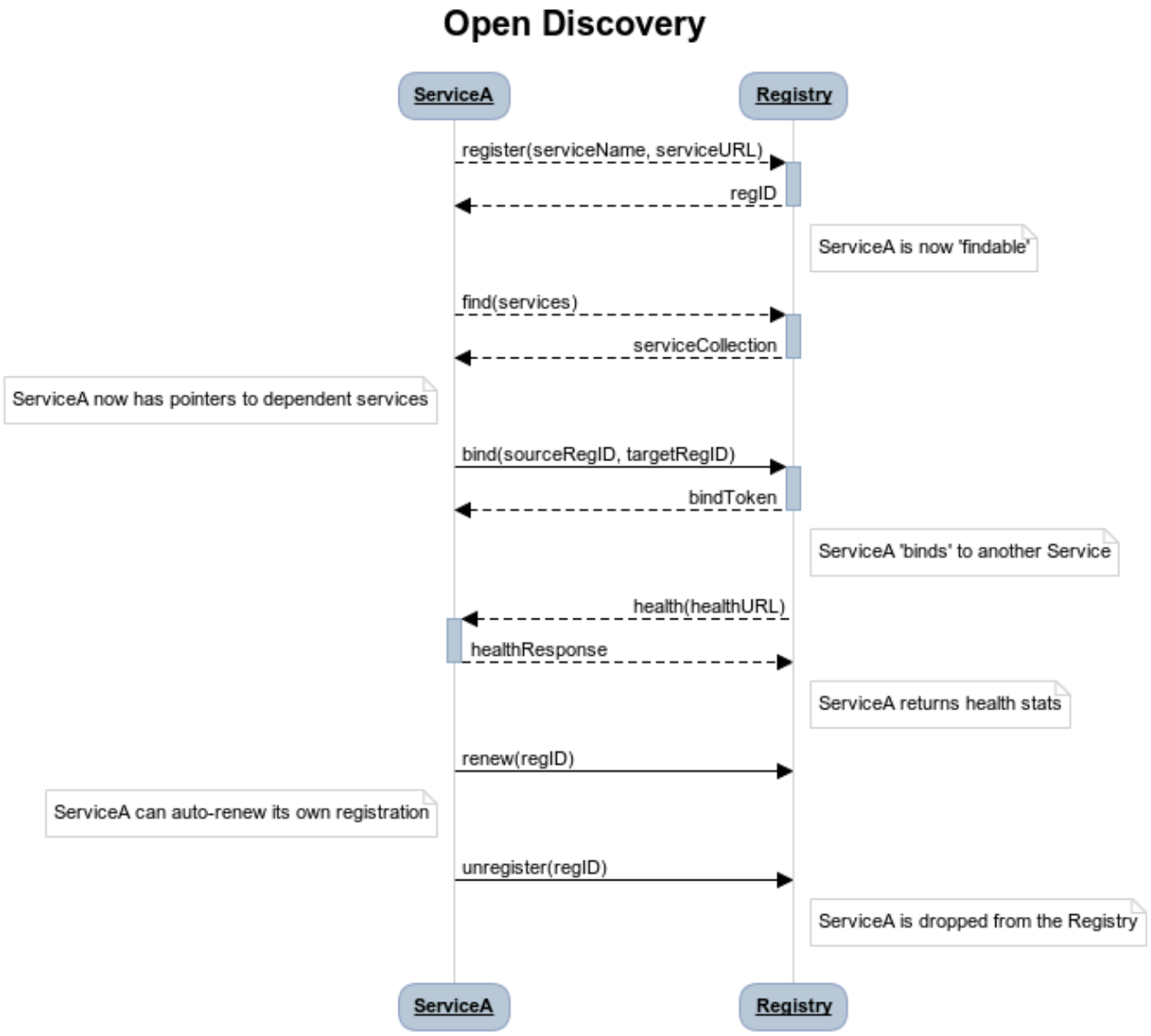
DISCO Registry Service Definitions

An up-to-date copy of the **DISCO Registry Service** can be found at http://rwmbook-registry.herokuapp.com/files/openapi.yaml. There is an HTML rendering of the OpenAPI document here: http://rwmbook-registry.herokuapp.com/files/openapi.html.

You can also find an ALPS-XML document for the **DISCO Registry Service** here: http://rwmbook-registry.herokuapp.com/files/disco-alps.xml.

DISCO Registry Service Sequence Diagram

Below is a simple sequence diagram that shows how the DISCO server interacts with external services.



DISCO Registry Service Basics

Below are a list of basic guidelines for implementing a DISCO Registry

Standard Protocols and Formats

The DISCO Registry Service is an open specification based on Web standards. It SHOULD support network interactions over the HTTP protocol and MAY support other protocols. It SHOULD support both HTML and JSON message formats and MAY support other formats which should be negotiable at runtime.

Security

DISCO Registries MAY require user-level security to be accessed (e.g. via the `WWW-Authentication` header over HTTP). DISCO Registries MAY implement authorization checks to restrict any user's ability to view data or execute actions. The authentication and authorization details are not part of this specification and SHOULD be implemented using existing open standards and well-documented for each implementation.

Extensions

DISCO Registries are free to implement additional extensions by adding new data fields and/or actions to the list of supported elements. However, these extensions MUST NOT remove or redfine any exsiting DISCO specifications. All extensions MUST be backward and forward compatible with the published DISCO specifications.

DISCO Registry Service Actions

The DISCO Registry Service includes the following Actions:

`register`

The `register` action is used by a *service* to register with a DISCO *registry*. This SHOULD be iniated by the service at start up (or when it is deployed into production). The `register` action has two required data elements `serviceName` (e.g. "UserMangement") and `serviceURL` (e.g. `http://example.com/services/user-mgmt/`). Upon successfully registering the service, the registry server SHOULD response with a `201 Created` and a `Location` header with a URL to the new service. The service MAY be able to pass additional parameters to the registry, but these MAY be ignored. The `register` action MUST return a `registryID` — a unqiue value — to the service for use in subsequent interactions.

`unregister`

The `unregister` action is used by a *service* to remove itself from the DISCO registry. This SHOULD be initiated whenever the service is stopped (either by controlled means, or a crash). The *service* MUST pass the `registryID` that is associated with the service to remove. The registry MAY require additional information in this request. Upon successfully 'unregistering' the service, the registry server SHOULD respond with `204 No Content`. The registry MAY return additional data but the service MAY ignore this information.

`renew`

The `renew` action is used by a *service* to 'ping' the registry — a means of proving the service is still up and running. The *service* MUST pass the `registryID` that is associated with the service entry to renew. The registry MAY require additional information with this request. Upon success, the registry SHOULD respond with `200 OK`. The registry MAY return additional data but the service MAY ignore it.

`find`

The `find` action is used by a *service* to query the registry for a list of service that match a search criteria. This is a way for services to "ask" a registry for a pointer to one or more services that can fulfill a need for that service (e.g. "Hey, registry, are there any services running that support credit card payments over HTTP using application/json?"). Services SHOULD send a list of search values for one or more fields (see the list below) and the registry SHOULD return zero or more service records that match the criteria. The returned list MUST include the `serviceURL` of services that match the search cirteria and MAY include additional fields. Registries are free to decide the manner in which the search is fulfilled, the order of the service, list, etc. Services can then use the returned `serviceURL` to initiate interactions with the target service directly.

`bind`

The `bind` action is used by a *service* to inform the registry that the service intends to "use" that service in subsequent interactions. The service MUST pass the `registryID` of the source service (the one "asking" for a connection, `sourceRegistryID`) and MUST pass the `registryID` of the target service to be "used" (`targetRegistryID`). This is an OPTIONAL action and MAY NOT be supported by the registry. If it is supported, the registry MUST return `201 Created` and a `Location` header with a URL to the new service binding, upon a successful completion of the request. The registry SHOULD also return a `bindToken` value. This value MAY be used in subsequent interactions between the source and target services. The registry MAY return more information but it MAY be ignored by the service.

`health`

The `health` action is used by the *registry* to check on the health of a registered service. The registry MUST use the `healthURL` provided by the *service* when that service completed the `register` action. The registry SHOULD honor the `healthTTL` value (in msec) provided by the _service at registration, too. Upon recieving a request from the registry, the service SHOULD return a list of status values and key information about the health

of the running service. This is an OPTIONAL element. Registries MAY NOT make health checks and services MAY NOT respond to health requests from the registry.

NOTE: Registries are responsible for determining when to 'evict' a service entry from their listings if/when the service is no longer sending `renew` requests or responding to registry `health` requests. To maintain their entry in the registry, services SHOULD support either `renew` or `health` or possibly both.

Tip

For a full description of the DISCO Registry service interface, see the ALPS (Application-Level Profile Semantics) document in the DISCO Registry repo.

DISCO Registry Vocabulary

The semantic vocabulary for the DISCO specification includes both property names and action names. The complete vocabulary can be found in the ALPS (Application-Level Profile Semantics) document found here: https://github.com/open-disco/registry/blob/master/disco-alps.xml.

DISCO Search

All DISCO registry servers SHOULD implement basic search support (for the `find` action). The internal records for DISCO servers currently have the following fields defined:

```
  <descriptor id="registryURL" type="semantic" text="URL of registry" />
  <descriptor id="registryID" type="semantic" text="unique registry id of the
service"/>
  <descriptor id="serviceURL" type="semantic" text="URL of service" />
  <descriptor id="serviceName" type="semantic" text="text name of service,
non-unique" />
  <descriptor id="tags" type="semantic" text="space separated lists of filter
tag words" />
  <descriptor id="status" type="semantic" text="current status of service [up,
down, unknown" />
  <descriptor id="semanticProfile" type="semantic" text="space separated list
of profile URIs" />
  <descriptor id="mediaType" type="semantic" text="space separated list of
mediaType identifiers" />
  <descriptor id="healthURL" type="semantic" text="URL to use when sending
health-check pings" />
  <descriptor id="healthTTL" type="semantic" text="time-to-live (in seconds)
for a valid health-check response" />
  <descriptor id="healthLastPing" type="semantic" text="last date/time
registry receigved a ping from the service" />
  <descriptor id="bindCount" type="semantic" text="count (estimate) of clients
using this service" />
  <descriptor id="renewURL" type="semantic" text="URL to use when renewing the
registry entry" />
  <descriptor id="renewTTL" type="semantic" text="time-to-live (in seconds)
for a valid renewal" />
  <descriptor id="renewLastPing" type="semantic" text="last date/time of
successful renewal" />
```

Other Search Considerations

Below are some other considerations when implementing search (`find`) support for DISCO servers.

Search Implementation supports *CONTAINS*

All of these fields SHOULD be exposed are URL query parameters. Searches SHOULD be executed as a "contains" query (e.g. `?tags=accounting` means "the tag fields contains the string *accounting*").

Some field values are transient

Note that some fields are local to the registry server (`registryURL`), only good for a particular running instance (`registryID`), or are temporary (`status`, `healthLastPing`, `bindCount`, `renewLastPing`). Others are static and SHOULD be the same across multiple instances of the same service (`serviceName`, `tags`, `semanticProfile`, `mediaType`, `healthURL`, `healthTTL`, `renewURL`, `renewTTL`).

All fields are valid

Registry servers SHOULD support searches on all these fields even if all these fields are not supported and/or contain null values in the registry's data storage. This means registry servers MUST NOT return error codes when requests pass fields not supported or fields w/ empty values.

Additional fields are possible

Reigstry servers MAY support additional fields for searching. If they do, these registries MUST NOT change the list of current fields (take them away, change their meaning) and all additional fields MUST be treated as OPTIONAL (e.g. you cannot create a new REQUIRED field for passing search queries to the registry). Registries that have additional search fields SHOULD signal their custom support using an additional profile record (ALPS is recommended) to signal the new fields that are supported. These registries MUST also emit the default profile record and MUST NOT reject queries based on the default profile.

Registry Bind Tokens

When a service uses the registry to locate ("find") other services to use, that registry MAY offer a "bind" action. This action is meant to represent the intent source service (requestor) to use the target service in some future interactions. The result of this bind action MAY be a `bindToken` — a value that contains metadata about both the services and the registry involved in making the connection between the two services. The target service MAY ask the requestor service supply the registry's `bindToken` as a way to validate or in some other way track service use and registry interactions.

> **Note**
>
> The format of the `bindToken` is designed to allow for customizable extensions. This specification only deines two elements of the `bindToken`: `type` and `token`. This specification only defines one token type: `simple`. Registries or other service groups MAY define their own token types in the future and they should document then sufficiently.

The Simple Bind Token

The `simple` bind token contains the following information:

```
bindToken = simple:<base64-body>
<base64-body> = registryKey:sourceRegistryID:targetRegistryID:utc-date-time
```

- `registryKey` is the unique ID of the registry that brokered the bind.
- `sourceRegistryID` is the unqiue ID of the source service (the 'requestor')
- `targetRegistryID` is the unique ID of the target service (the one to be 'used')
- `uts-date-time` is the UTC-formatted date-time the binding token was created.

Here's an example `bindToken`:

Simple Bind Token

```
Start with the following values:
 - "registryKey":"ecc01cda-689a-4237-9590-9be7d45bd5ad",
 - "sourceRegistryID":"22i52uadfbr",
```

```
 - "targetRegistryID":"2qfe1yzbnms",
 - "dateCreated":"Sun, 11 Mar 2018 02:08:20 GMT"

Create a token string:
  ecc01cda-689a-4237-9590-9be7d45bd5ad:22i52uadfbr:2qfe1yzbnms:Sun, 11 Mar
2018 02:08:20 GMT

Convert the token string to base64 and prefix with the bindToken type of
'simple'
  simple:ZWNjM...gR01U
```

Last updated 2019-09-27 23:07:18 EDT