International
Virtual
Observatory
Alliance

IVOA Execution Planner - design outline

Version

IVOA Note 20211014

Working Group
Grid and Web Services Working Group
This version
https://www.ivoa.net/documents/IVOA-EP-note/20211014
Latest version
https://www.ivoa.net/documents/IVOA-EP-note
Previous versions

Author(s)
Dave Morris, Sara Bertocco
Editor(s)
Dave Morris, Sara Bertocco

Abstract

The IVOA Execution Planner (IVOA-EP) interface is a HTTP webservice interface that provides a simple way to discover and access computing services. This document uses a series of use cases and example applications to illustrate the functionality provided by the IVOA-EP interface.

Status of this document

Contents

Acknowledgments

This document derives from discussions among the Grid and Web Services working group of the IVOA.

This document has been developed with support from the UK Science and Technology Facilities Council (STFC), from the Italian National Institute for Astrophysics (INAF) and from the European Commission's Research and Innovation Program Horizon 2020 under the project ESCAPE (Grant n.824064).

Conformance-related definitions

The words "MUST", "SHALL", "SHOULD", "MAY", "RECOMMENDED", and "OPTIONAL" (in upper or lower case) used in this document are to be interpreted as described in IETF standard, (Bradner, 1997).

The *Virtual Observatory (VO)* is general term for a collection of federated resources that can be used to conduct astronomical research, education, and outreach. The International Virtual Observatory Alliance (IVOA) is a global collaboration of separately funded projects to develop standards and infrastructure that enable VO applications.

1 Introduction

The Execution Planner (IVOA-EP) interface aims to provide a simple way to discover and access computing services.

The design of the IVOA-EP interface is based around two key classes of objects, computing Tasks and Services.

The primary question the IVOA-EP interface is designed to answer is *"where can I run this Task ?"*, or more specifically, *"which computing Services can I use to run this Task?"*

The simplest solution to this problem would be for a central agency to maintain enough metadata about all the available computing Services to be able to answer that question using a simple database query.

The simplest case is just to match the type of task with the type of Service, using a simple string match.

```
SELECT * FROM services WHERE servicetype = 'binder'
```

This works for a small fixed set of Task types with a simple set of acceptance criteria. However as the range and complexity of Task types begins to grow a centralised solution like this becomes harder to maintain.

Different types of Tasks will have different metadata to describe them and different Service instances will have different criteria for accepting or rejecting Tasks. Each time a new type of Task or Service becomes available, the software for evaluating execution requests will need to be updated.

As the system evolves, we can see the criteria or rules for accepting a Task growing in complexity over time. If we imagine a system capable of deploying and executing a complex chain of interconnected software components, the criteria for accepting or rejecting a complex Task like this will also grow in complexity.

The design of the IVOA-EP interface aims to address this complexity by using the Separation of Concerns[1] pattern to delegate as much as possible to the Service instances

The IVOA-EP interface defines a simple stateless HTTP interface that supports `GET` and `POST` requests. The following sections use a series of example Task types to illustrate how the IVOA-EP service interface works. In all of these cases, the IVOA-EP service interface provides a common method to query Services about their capabilities.

For simplicity, this document will represent IVOA-EP query request and response messages using a simplified same JSON notation. Hopefully this makes it easier for the reader to compare the information in the request and response messages. Section 9 describes more detail about how IVOA-EP requests and responses should be serialised in HTTP requests.

There is no absolute requirement for the URI identifiers to be resolvable into a resource. However providing a resolvable resource at the URI location is an ideal way to communicate information about the Task and Service types to others outside the local community.

2 Type identifiers

The IVOA-EP service specification does not define a fixed list of Task and Service types. The intention is that the Task and Service types will be developed organically by communities based on their science use cases.
In order to support this grass-root model, the IVOA-EP specification recomends using resolvable Uniform Resource Identifier (URI)[2] identifiers to refer to Task and Service types.
The choice of schema and format for the type URIs is left to the individual communities, the only requirement is that they be globally unique.

For readability this document will use a simplified URI scheme to represent the Task and Service type identifiers.

```
    tasktype = uri://jupyter-notebook
```

For a real-world deployment, the easiest way of ensuring global uniqueness is to use a HTTP URL that points to a location controlled by the community.

```
    tasktype = http://purl.escape.eu/types/jupyter-notebook
```

[1] https://en.wikipedia.org/wiki/Separation_of_concerns
[2] https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

Alternatively one of the established resolvable URI registration schemes may be used, such as the DOI scheme or IVOA registry.

```
    tasktype = doi://########/jupyter-notebook
```

```
    tasktype = ivo://########/jupyter-notebook
```

3 Notebook services

The following sections describe different types of notebook Task, including a generic JupyterHub Service, a BinderHub Service and an ESCAPE ESAP notebook Service, and finally a Zeppelin notebook Service that supports multiple languages including PySpark.

3.1 Jupyter notebook

Replicating the simplest use case outlined above, where the name of the Task type matches the type that the Service implements, the client can call the IVOA-EP interface with a single parameter, `tasktype`, representing the type of Task the client is asking about.

```
    HTTP GET /accepts?tasktype=uri://jupyter-notebook
```

If the service is not able to accept `uri://jupyter-notebook` Tasks, then it can simply reply with JSON response containing a `reponseword` of `NO`.

```
    {
    "reponseword": "NO"
    }
```

If the service can accept `uri://jupyter-notebook` Tasks then it should reply with a simple JSON response containing a `reponseword` of `YES` along with details of how to execute the Task.

```
    {
    "reponseword": "YES",
    "servicetype": "uri://jupyter-hub",
    "serviceinfo": {
        "endpoint": "http://jupyter.example.org/"
        }
    }
```

The `servicetype` value tells the client what kind of service is available, and the `serviceinfo` element provides details of how to connect to it.

In this example, `tasktype=uri://jupyter-notebook` in the request applies to a generic Jupyter notebook, as defined by the Jupyter[3]

[3] https://jupyter.org/

project. In the response, servicetype=uri://jupyter-hub refers to a JupyterHub service, as defined by the Jupyter project.

In order to run a notebook in a JupyterHub Service, a client would need to know the endpoint URL of the Service, which is provided in the serviceinfo.endpoint element of the response. The client can use this endpoint URL to pass the notebook to the JupyterHub service and launch the Task.

3.2 Binder notebook

It is also possible to run a generic Jupyter notebook in a BinderHub service provided by the Binder[4] project. In which case, given the same request to run a uri://jupyter-notebook Task.

```
HTTP GET /accepts?tasktype=uri://jupyter-notebook
```

A BinderHub service would also reply with a positive response.

```
{
"reponseword": "YES",
"servicetype": "uri://binder-hub",
"serviceinfo": {
    "endpoint": "http://binder.example.org/"
    }
}
```

The response from the BinderHub Service is similar to the response from the JupyterHub Service, but the meaning is slightly different. Setting the servicetype to uri://jupyter-hub or uri://binder-hub in the response, tells the client what kind of Service to expect at the endpoint URL. It is then up to the client to decide how to send the details of the notebook to the Service based on the service type.

A BinderHub service can also handle a more complex Task than just a generic Jupyter notebook. If the notebook comes as part of a git repository that contains additional information about the dependencies or environment the Task requires, such as requirements.txt or environment.yml, then a BinderHub service can use this information to build a new Docker container based on the requirements and deploy it in the BinderHub service.

In order to check if a Service accepts this more complex type of Task, the client would set the tasktype request parameter to uri://binder-notebook.

```
HTTP GET /accepts?tasktype=uri://binder-notebook
```

A generic JupyterHub service would not be able to accept a uri://binder-notebook Task, so it would reply with a JSON response containing a reponseword of NO.

[4]https://binderhub.readthedocs.io/

```
{
"reponseword": "NO"
}
```

A BinderHub service that can accept a `uri://binder-notebook` Task would reply with a positive response, with the `servicetype` set to `uri://binder-hub` and the `serviceinfo.endpoint` pointing to BinderHub service endpoint.

```
{
"reponseword": "YES",
"servicetype": "uri://binder-hub",
"serviceinfo": {
    "endpoint": "http://binder.example.org/"
    }
}
```

Given the `servicetype` and `serviceinfo.endpoint` elements in the response, the client now has enough information to pass launch the Task by passing a reference to git repository to the BinderHub service.

3.3 ESAP notebook

In terms of the ESCAPE project, there may be additional components beyond simply adding the required software dependencies. If a notebook requires access to data in the ESCAPE DataLake, then for the *"DataLake as a Service"* to work correctly, the notebook needs to be run on a compute platform that is co-located with a Rucio Storage Element (RSE) [5] that is part of the ESCAPE DataLake, and the mechanism used to launch the Task needs to pass the appropriate authentication tokens into the notebook environment to enable it to access the DataLake.

If we define a new Task type, `uri://esap-notebook`, which refers to a notebook Task defined by the ESCAPE ESAP project. Then an IVOA-EP service client can use this to check if a Service supports this environment.

```
HTTP GET /accepts?tasktype=uri://esap-notebook
```

In this case, the generic JupyterHub and BinderHub Services would not understand the new Task type, and so would reply with a negative response.

```
{
"reponseword": "NO"
}
```

A Service deployment that does understand this new Task type, and can provide access to data in the ESCAPE DataLake, would reply with a positive response.

[5] https://rucio.readthedocs.io/en/latest/overview_Rucio_Storage_Element.html

```
    {
    "reponseword": "YES",
    "servicetype": "uri://binder-hub",
    "serviceinfo": {
        "endpoint": "http://binder.example.eu/"
        }
    }
```

Note that the Service type in the response is still `uri://binder-hub`. This means that the webservice interface that the client interacts with is the same as the generic BinderHub. The difference with this Service instance is that it is deployed within the ESCAPE network and is capable of providing access to the ESCAPE DataLake. Which means that in addition to being able to run generic `uri://jupyter-notebook` and `uri://binder-notebook` Tasks, this Service is also capable of understanding and executing a `uri://esap-notebook` Tasks.

3.4 Zeppelin notebook

Apache Zeppelin[6] is a browser based notebook platform that provides a similar user experience and functionality to the Jupyter notebook platforms. However, the technical details of the notebook format and webservice API are different, which means that the notebook Tasks are not equivalent.

The IVOA-EP API provides a common interface to enable a client to ask questions about this different type of notebook Service by setting the `tasktype` parameter to `uri://zeppelin-notebook`.

```
    HTTP GET /accepts?tasktype=uri://zeppelin-notebook
```

IVOA-EP services that represent JupyterHub and BinderHub Services would not understand the `tasktype` and would simply reply with a negative response.

```
    {
    "reponseword": "NO"
    }
```

An IVOA-EP services that represented a Zeppelin Service would reply with a positive response, and include details of how to connect to the service.

```
    {
    "reponseword": "YES",
    "servicetype": "uri://zeppelin-service",
    "serviceinfo": {
        "endpoint": "http://zeppelin.aglais.uk/"
        }
    }
```

[6]https://zeppelin.apache.org/

Given the `servicetype` and `serviceinfo.endpoint` elements in the response, the client now has enough information to launch the Zeppelin notebook Task.

3.5 PySpark notebook

The Zeppelin platform includes interpreters for several different programming languages, and can support multiple languages within a single notebook.

An example of this is the PySpark[7] Python API that enables users to write Python code that performs data analysis using a Spark cluster. If a Zeppelin platform has access to a Spark cluster, then it will be able to handle notebooks that contain both standard Python and PySpark elements in the same notebook.

If we follow the same pattern as we did for the Jupyter notebooks, then we could define another Task type, `zeppelin-pyspark-notebook`, to describe Zeppelin notebooks that include PySpark code in them. However, as a Zeppelin notebook can include more than one language within a single notebook, the list of Task types would become overly complex if we tried to handle all the possible combinations.

A better solution would be to add a second parameter to the `accepts` request that contains a list of the required languages.

A JSON representation of an `accepts` query for a Zeppelin notebook Task that includes Markdown[8], Python and PySpark elements would be as follows:

```
    {
    "tasktype": "uri://zeppelin-notebook",
    "taskinfo": {
        "languages": [
            "md",
            "python",
            "pyspark"
            ]
        }
    }
```

The same query serialised as a HTTP GET request would be:

```
    HTTP GET /accepts?tasktype=uri://zeppelin-notebook
        &taskinfo.languages={md,python,pyspark}
```

In this example, we have a Zeppelin notebook task that includes a list of three languages that are used in the notebook.

[7] http://spark.apache.org/docs/latest/api/python/
[8] https://daringfireball.net/projects/markdown/

```
    {
    "tasktype": "uri://zeppelin-notebook",
    "taskinfo": {
        "languages": [
            "md",
            "python",
            "pyspark"
            ]
        }
    }
```

An IVOA-EP service that represents a Zeppelin Service that can support all three languages would reply with a positive response. In addition, the `servivceinfo` element for the `uri://zeppelin-service` could contain a list of the languages it supports.

```
    {
    "reponseword": "YES",
    "servicetype": "uri://zeppelin-service",
    "serviceinfo": {
        "endpoint": "http://zeppelin.aglais.uk/",
        "languages": [
            "md",
            "python",
            "pyspark"
            ]
        }
    }
```

The IVOA-EP service specification defines the order in which the request elements are processed. The first step is to check the `tasktype`. If the target Service does not understand the `tasktype`, then the IVOA-EP service should simply ignore the rest of the request and reply with a negative response.

An IVOA-EP service that represents a JupyterHub or BinderHub Service would not recognise `uri://zeppelin-service` as a valid `tasktype`, and so would skip the `taskinfo` block and simply reply with `NO`.

```
    {
    "reponseword": "NO"
    }
```

Defining the parsing sequence in this way means that the content of the `taskinfo` element can be specific to each Task type. In this example, if the IVOA-EP service recognises and understands the `uri://zeppelin-service` `tasktype`, then it will know to expect a list of `languages` in the `taskinfo` element.

An `accepts` query for a different `tasktype` would have different, type specific, content in the `taskinfo` element.

4 Container services

The following sections describe different types of container execution Services. Starting with the basic Docker container and Docker Compose Services described in the IVOA UWS-CE [cite] note, and an IVOA-EP service that represents a Portainer Service deployment.

4.1 Docker UWS

The UWS-CE [cite] note describes a basic Docker container execution Service that uses the IVOA UWS webservice interface to manage container execution Tasks.

The simplest IVOA-EP implementation for a basic UWS-CE service could just check the `tasktype` parameter to answer the question *"Can I run a Docker container here?"*.

```
    HTTP GET /accepts?tasktype=uri://docker-container
```

IVOA-EP services that represent Services that can execute Docker containers would reply with `YES`, and IVOA-EP services that do not support Docker containers would reply with `NO`.

However, a particular UWS-CE instance may want to apply some checks on the content of the Docker container before allowing it to be executed on their platform. For example a UWS-CE Service may only accept containers from a white list of allowed images, or it may want to examine the container image to check that it is derived from a specific base image.

In this situation, the question changes from *"Can I run **a** Docker container here?"* to *"Can I run **this** Docker container here?"*, and in order to answer this question the IVOA-EP service needs to see the container image to check that it meets the acceptance criteria.

Following the pattern outlined in the previous section, the IVOA-EP request to ask this question would start with the `tasktype` parameter set to `uri://docker-container`, followed by the Task specific content in the `taskinfo` element, containing the fully qualified name of the container image, for example `docker.io/example:1.0`.

```
    {
    "tasktype": "uri://docker-container",
    "taskinfo": {
        "image": "docker.io/example:1.0"
        }
    }
```

The IVOA-EP service can then compare the image name with an internal white-list of accepted images, or it could download and inspect the image to verify that it is derived from the correct base image.

Assuming the container image meets the acceptance criteria, the IVOA-EP service would reply with a positive response, with the `servicetype` set to `uri://docker-uws` to indicate the type of Service, and the `serviceinfo` would contain the endpoint URL to access the service.

```
    {
    "reponseword": "YES",
    "servicetype": "uri://docker-uws",
    "serviceinfo": {
        "endpoint": "http://example.org/dkuws",
        }
    }
```

4.2 Docker compose

In addition to the basic Docker container execution Service, the UWS-CE [cite] note describes a Docker Compose Service that handles Docker Compose Tasks as UWS jobs.

Again, the simplest IVOA-EP implementation could just check the `tasktype` parameter to answer the question *"Can I run a uri://docker-compose Task here?"*.

```
    HTTP GET /accepts?tasktype=uri://docker-compose
```

IVOA-EP services can execute Docker Compose Tasks would reply with `YES`, and IVOA-EP services that don't would reply with `NO`.

However, a more realistic scenario would be for the UWS-CE Service to apply some checks on the content of the Docker Compose Task before allowing it to be executed on the platform.

One way to enable this would be for the client to send a URL that points to the location of the Docker Compose file.

```
    {
    "tasktype": "uri://docker-compose",
    "taskinfo": {
        "composefile": "https://edin.ac/3jqocuV.yml"
        }
    }
```

However, this requires a copy of Docker Compose file to be publicly accessible on the internet, which may not always be possibly. A better alternative would be to use a HTTP `multipart/form-data POST` [9] request to include the content of the Docker Compose file in the request body.

TODO develop a better example using a containerized version of the Vollt TAP service[10].

[9] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST`
[10] `https://github.com/gmantele/vollt`

```
POST /accept HTTP/1.1
Host: foo.example
Content-Type: multipart/form-data;boundary="boundary"

--boundary
Content-Disposition: form-data; name="tasktype"

uri://docker-compose
--boundary
Content-Disposition: form-data; name="taskinfo.
composefile"; filename="compose.yml"
Content-Type: text/yaml

version: '3.9'
networks:
    external:
    internal:
services:
    database:
        image:
           "example/postgres:9.2"
        networks:
            - internal
        environment:
            POSTGRES_DB:       "${database}"
            POSTGRES_USER:     "${usernam}"
            POSTGRES_PASSWORD: "${password}"
    webapp:
        image:
           "example/webapp:1.1"
        networks:
            - internal
            - external
        ports:
            - "8080:8080"
--boundary--
```

This example sets the `tasktype` to `uri://docker-compose` in the first block of `form data`, and then includes the content of the Docker Compose file in the second block.

Note that the only part required by the IVOA-EP specification is the initial `tasktype` element set to `uri://docker-compose`. After that, the rest of the content of the message is defined by the specification for the UWS-CE application and the IVOA-EP extension for handling Docker Compose Tasks.

TODO "No, but maybe" response offering a template compose file ?

4.3 Portainer service

Portainer[11] is a commercial platform that *"enables centralized configuration, management and security of Kubernetes and Docker environments, allowing you to deliver 'Containers-as-a-Service' to your users quickly, easily and securely."*

The IVOA-EP design pattern supports a number different of ways to integrate a Portainer service into the IVOA ecosystem.

- A UWS-CE interface in front of a Portainer service accepting Docker container Tasks.
- A UWS-CE interface in front of a Portainer service accepting Docker Compose Tasks.
- Exposing the Portainer service directly via an IVOA-EP interface

In the first two cases, the UWS-CE interface acts as a proxy for the Portainer service. Initialising and running the Docker or Docker Compose Tasks on the Portainer service, and providing an IVOA compatible interface that represents the Portainer tasks as UWS jobs.

In the third case, the IVOA-EP service would provide a registered entry-point to represent the Portainer service in the IVOA ecosystem. The IVOA-EP service would accept Tasks with `tasktype` of `uri://docker-container`, `uri://docker-compose` and two new types of `uri://portainer-container` and `uri://portainer-compose`. The IVOA-EP service would simply match the `tasktype` names and return a positive response, with `servicetype` set to `uri://portainer-service`

```
{
"reponseword": "YES",
"servicetype": "uri://portainer-service",
"serviceinfo": {
    "endpoint": "http://portainer.example.org/",
    }
}
```

If a client does not understand how to drive a Portainer service, then it simply ignores this offer and moves on to try another service. If a client does understand how to drive a Portainer service, then the `serviceinfo.endpoint` element of the response enables the Portainer client to contact the Portainer service directly.

This represents a key design pattern of the IVOA-EP service. Implementing just enough to provide a standard IVOA interface that can be registered in the IVOA Registry to make a 3rd party service find-able and use-able as part of the IVOA ecosystem, without having to standardise the whole of the 3rd party interface.

[11] https://www.portainer.io/

4.4 Multiple interfaces

A complex service like a full Portainer deployment may be capable of providing multiple methods for running the same Task type.

Given a basic request to execute a Docker container Task.

```
HTTP GET /accepts?tasktype=uri://docker-container
```

An IVOA-EP service that represents the Portainer service may return an array of service interfaces in the response, allowing the client to choose the most appropriate method to use.

```
{
"reponseword": "YES",
"interfaces": [
        {
        "servicetype": "uri://portainer-service",
        "serviceinfo": {
            "endpoint": "http://portainer.example.org/"
        },
        {
        "servicetype": "uri://compose-uws",
        "serviceinfo": {
            "endpoint": "http://uws.example.org/compose"
        },
        {
        "servicetype": "uri://docker-uws",
        "serviceinfo": {
            "endpoint": "http://uws.example.org/docker"
        }
    ]
}
```

In this example, the IVOA-EP service is offering three methods for executing a `uri://docker-container` Task.

- The full Portainer service interface.
- A UWS-CE interface that accepts can accept the `uri://docker-container` Task if it is wrapped as a `uri://docker-compose` Task.
- A UWS-CE interface that accepts `uri://docker-container` Tasks directly.

The client is free to choose which method is most appropriate to use.

- If the client understands how to use a Portainer service, then it can use the `serviceinfo.endpoint` to access the Portainer Service directly.

- If the client understands how to wrap a `uri://docker-container` as a `uri://docker-compose` Task, then it may choose to use the `uri://compose-uws` Service.
- Or the client may choose to simply use the basic `uri://docker-uws` Service.

5 Computing resources

Request/response section describing the required and available resources.

Example resource request:

```
    resources:
        # Storage space (Gbytes)
        minstore:  8G
        maxstore: 10G
        # CPU cores
        mincores:  8
        maxcores: 10
        # Memory (Gbytes)
        minmemory:  8G
        maxmemory: 10G
        # Startup time (s)
        minstartup:  5s
        maxstartup: 60s
        # Execution time (duration)
        minduration:  5m
        maxduration: 10m
```

If the requested resources are available, the IVOA-EP service may reply with a simple YES response. It may also include a resources element with updated values for the available resources.

If the request asks for more than the available resources, the Task will be rejected. Simple NO. It may also include a resources element with available resources and optional warning messages.

ISO 8601 duration

6 Data access

Data access requirements ..

Data access vocabulary ..

6.1 VOSpace

Example reference to data in VOSpace (INAF archive?). Identity and auth ..

6.2 Rucio

Example reference to data in Rucio (ESCAPE DataLake?). Identity and auth ..

6.3 Amazon S3

Example reference to data in S3 (internal and external). Examples, Amazon, DigitalOcean, Openstack, STFC Echo. Identity and auth ..

7 Authentication

Different Service instances may have different criteria for who they will allow to execute Tasks on their Service.

Some services, such as the public Service provided by the BinderHub Federation may be free and open to the public to use [1].
An IVOA-EP service that represents a public access Service like this may accept any HTTP request, with or without authentication. However, most computing services will be funded to provide compute resources for specific communities, and will require some level of user authentication to control access to their resources.
If an authenticated identity is provided as part of the `/accepts`, then the EP service can use this identity as part of the evaluation criteria.

```
HTTP GET /accepts?tasktype=uri://docker-container
Authorization: Bearer S1AV32hkKG
```

If the authenticated identity is allowed to perform the task on the target platform, the EP service replies with a positive response as normal.

```
{
"reponseword": "YES",
"servicetype": "uri://portainer-service",
"serviceinfo": {
    "endpoint": "http://portainer.example.org/",
    }
}
```

If the authenticated identity is not allowed to perform the task on the target platform, the EP service may reply with a simple negative response.

```
    {
    "reponseword": "NO"
    }
```

Or it may provide additional information about the reason why.

```
    {
    "reponseword": "NO"
    "reponseinfo": {
        "reasons": [
                {
                "httpcode": 403,
                "text": "Not authorised"
                }
            ]
        }
    }
```

8 Deployment and discovery

8.1 Service deployment

The IVOA-EP interface is designed to work both within the context of the IVOA community, or as apart of a separate domain outside the IVOA, such as the ESCAPE ESAP community.
In both situations, there would typically be oneIVOA-EP service associated with each platform that provides computing resources to the community.
In most cases the IVOA-EP service would normally be deployed as part of the computing platform itself. The entity that provides the computing platform would also provide and maintain an IVOA-EP service that handles queries about running Tasks on that computing platform.
However that is not a necessarily required. For example, a project like ESCAPE may choose to deploy a stand-alone instance of an IVOA-EP service that handles queries about an external computing platform like the MyBinder service provided by the BinderHub Federation[12]. The IVOA-EP service itself would be hosted and maintained by a member of the ESCAPE community, but it can be configured to handle queries about running Tasks on the MyBinder service.
This distributed micro-service architecture enables communities to build up a network of IVOA-EP services that describe both internal and external computing resources in an interoperable way.

[12]https://mybinder.readthedocs.io/en/latest/about/federation.html

8.2 Service discovery

The IVOA-EP interface is designed to be compatible with using the IVOA registry for service discovery. The IVOA-EP specification defines a VOResource metadata extension for describing IVOA-EP services and their capabilities.
IVOA-EP services deployed by members of the IVOA community will be registered in the IVOA registry, enabling service discovery using the existing IVOA registry tools.
However, although registering services in the IVOA registry is encouraged, it is not required for the IVOA-EP services to function. For example, a project like ESCAPE may choose to deploy their own service discovery mechanism, separate from the IVOA registry. This can be as simple as a database table maintained inside the an ESAP portal that lists the IVOA-EP services associated with the computing platforms available to that community.

9 Data formats

Default data format for IVOA-EP services is to use YAML for data inputs sent via POST messages and the default response format is JSON for outputs. This pattern of using different formats for the request and response data is used by a number of webservice interfaces, for example the Kubernetes `kubectl` control application.

The reasoning behind this pattern is that YAML is the best format for storing human edited configurations, such as the resource requirements and metadata. Whereas JSON is best format for handling and parsing webservice responses.

9.1 Request formats

The IVOA-EP interface can accept both YAML and JSON documents in the elements of a HTTP multipart POST messages. The client should specify the format for each element in the HTTP POST message.

```
    POST /accept HTTP/1.1
    Host: foo.example
    Content-Type: multipart/form-data;boundary="boundary"

    --boundary
    Content-Disposition: form-data; name="tasktype"

    example-task
    --boundary
    Content-Disposition: form-data; name="taskinfo.example.
   one"; filename="example-1.yml"
    Content-Type: text/yaml
```

```
        ....
        ....
        ....
    --boundary
    Content-Disposition: form-data; name="taskinfo.example.
   two"; filename="example-2.json"
    Content-Type: text/json

        ....
        ....
        ....
    --boundary--
```

Where there is a preference for a particular format, e.g. to match the format used by a 3rd party application, it should be declared in documentation for that particular IVOA-EP application.

9.2 Response formats

The IVOA-EP interface can generate JSON, YAML or XML response formats. A client can specify the preferred format using the HTTP Accepts header.

References

Bradner, S. (1997), 'Key words for use in RFCs to indicate requirement levels', RFC 2119.
http://www.ietf.org/rfc/rfc2119.txt