SRCNet ExecutionBroker prototype

Dave Morris, Bob Watson

SRCNet team Coral

SRCNet demo
July 2024

Dave Morris
dave.morris@manchester.ac.uk

Bob Watson
bob.watson@manchester.ac.uk

Where we started :

- 61 page document
- lots of theory
- no code

IVOA meeting in May :

** name change **

~~IVOA ExecutionPlanner~~

IVOA ExecutionBroker

- slightly slimmer document
- still theoretical
- no code

IVOA Execution Broker

Version 1.0

IVOA Working Draft 2024-04-25

Working Group
GWS
This version
https://www.ivoa.net/documents/ExecutionBroker/20240425
Latest version
https://www.ivoa.net/documents/ExecutionBroker

https://github.com/ivoa-std/ExecutionBroker

Primary deliverable is a prototype that demonstrates how an ExecutionBroker service would function in SRCNet.

Secondary deliverable is an updated IVOA standard document with a functional OpenAPI service specification.

We started with 2 services defined in the standard ExecutionBroker and Executionworker.

First question was why do we need the second service ?

Turns out we don't.

The only function of the worker service was to monitor the status of the job as it was executed.

Everything the worker did could be handled by a second method within the main ExecutionBroker service.

ExecutionWorker - gone

We started with two main objects in the data model.

ExecutionBroker generates offers, representing how a job could be performed.

ExecutionWorker contained jobs, representing the state of the jobs as they were executed.

Turns out the only difference were the status fields.

A broker offer goes through the following steps :

- OFFERED
- ACCEPTED
- REJECTED
- EXPIRED

A worker job goes through the following steps :

- WAITING
- STANDUP
- READY
- RUNNING
- TEARDOWN
- COMPLETED
- FAILED
- CANCELLED

Turns out we can combine them into a single 'execution' object that goes through the combined set of steps :

- OFFERED
- ACCEPTED
- REJECTED
- EXPIRED
- WAITING
- STANDUP
- READY
- RUNNING
- TEARDOWN
- COMPLETED
- FAILED
- CANCELLED

We end up with the same data model representing the state of an offer before it is executed, an execution while it is running, and the historical record of the execution after it has completed.

Secondary deliverable is an updated IVOA standard document with a functional OpenAPI service specification.

This is part of a new direction for IVOA services.

To implement as much as possible of the service specification in a machine readable form.

Reducing the size and complexity of the human readable text description needed in the document.

ExecutionBroker is one of the specifications that have been selected to act as a path finder for this work.

To explore this way of developing our services we started by with an initial machine readable definition of the service API and data model

and then used this to generate the classes for the service interface and data objects

Our experience of this has been patchy, with different results for different languages.

There are two aspects of the ExecutionBroker design that push the boundaries of what is possible with auto generated code from an OpenAPI service specification.

The first complexity is the polymorhpic message structure.

The ExecutionBroker API is designed to handle different message structure depending on what type of thing it is trying to describe.

For example, we need to use different the fields to describe a Docker container and a Binder notebook.

The ExecutionBroker API handles this by using a type identifier to indicate the different message structures.

```
executable:
  type: urn:docker-container-0.1
  image: the image name
  namespace: the namespace in the repository
  repository: the docker repository
  platform: the target platform e.g AMD64 or ARM


executable:
  type: urn:binder-notebook-0.1
  repository: the source code repository
  notebook: the name of the notebook file
  requirements: the name of the requirements
file
```

This type of polymorphism is a normal part of object orientated programming languages, and is a well documented part of the OpenAPI specification.

https://swagger.io/docs/specification/data-models/inheritance-and-polymorphism/

However some of the OpenAPI code generators have problems interpreting this.

Particularly the Python generators.

The second aspect of the ExecutionBroker design that causes problems is the wire format negotiation.

The ExecutionBroker API should be capable of handling requests and reponses in JSON, YAML and XML.

Using the content-type and accepts HTTP headers to indicate which format to use.

Again, this type of content negotiation is a well established part of the HTTP protocol, and and is a well documented part of the OpenAPI specification.

https://swagger.io/docs/specification/media-types/

Again, we have found issues with some of the OpenAPI code generators.

The documentation for both the FastAPI and Connexion frameworks emphasis compatibility with spec-first schema driven code development.

“Connexion is a modern Python web framework that makes spec-first and api-first development easy. You describe your API in an OpenAPI (or swagger) specification with as much detail as you want and Connexion will guarantee that it works as you specified.”

https://connexion.readthedocs.io/en/latest/

fastAPi [is] based on open standards

- *OpenAPI for API creation, including declarations of path operations, parameters, body requests, security, etc.*
- *….*

https://fastapi.tiangolo.com/features/

Our experience did not live up to expectations

So far we have not been able to generate a fully functional webservice, with both polymorphism and content negotiation using either the Connexion or FastAPI code generators.

To explore this further we tried two alternatives.

First we tried the OpenAPI code generator for the Java Spring framework.

Which produced a fully functional implementation first time we ran it.

With full support for both polymorphic messages and the content negotiation for JSON, YML and XML.

No manual edits required.

In order to meet the time deadlines for this work we are using the generated Java Spring code as the platform for developing the rest of the business logic.

Once we find a way of reliably generating the FastAPI Python code we intend to go back and port the business logic code from the Java implementation into the Python FastAPI framework.

The second alternative we explored was to use ChatGPT.

We asked ChatGPT to give us some examples of how to represent things in OpenAPI and how to generate the corresponding code blocks.

The code it generated was so good, we ended up giving it the whole OpenAPI service specification and asking it to generate the implementation from it.

ChatGPT generates good well documented code.

However it tends to take shortcuts.

- It will generate code data objects for half of the schema objects, and skip the rest.
- It will generate code for some of the methods, but leave others undefined.
- It will generate handlers for two of the three formats, and forget to add the third one.

The difference between using ChatGPT and one of the conventional code generators is that you can have a conversation with it.

When it misses out some of the fields, you can give it an additional prompt asking it to fill in the missing parts and it will generate a new set of code with the missing parts included.

Gradually over a series of prompts you can identify the missing or incorrect parts and get ChatGPT to correct them.

Ending up at the end of an afternoon with a very nice implementation of the full service in FastAPI Python classes.

The problem with ChatGPT is reproducibility.

ChatGPT is just folding and editing blocks of text, that happen to be in the Python langauge.

ChatGPT doesn't understand what the text actually means.

It is not reasoning about what the code will do when it is executed.

If you make anything more than a minor change to the OpenAPI specification, ChatGPT will revert to missing out some things.

If you start a new ChatGPT session, and try to use a complex prompt to get it to generate the whole service code in one go, it will find new ways to forget things that will need new prompts to correct the errors.

ChatGPT is OK for a one-off code generation, with human supervision to catch and correct the errors and omissions.

Summary of where we are:

- We have the majority of the OpenAPI specification done.
- We have verified that OpenAPI is capable of representing polymorphic messages and content format negotiation.
- We have at least one service implementation generated directly from the OpenAPI specification.
- While Dave has been working on the OpenAPI service interface, Bob has been exploring how to configure a CANFAR service to use tokens for authentication and how to use the REST API to launch sessions in the CANFAR system.
- We plan to bring these two parts together over the comming weeks building up the ExecutionBroker business logic to the point where it can provide the information needed to launch a job in the CANFAR system.