Extractor Info Fetcher Service

This page is under construction. This page will describe the features of Extractor Info Fetcher Service, the current choices of how it can be implemented and the implementation method that is chosen, elaborating the reasons behind it.

Introduction

Moving compute to data is one of the ways through which Brown Dog will address Big data. This means that rather than sending large files for processing to remote extraction services, BD clients will download the extraction services to local machine as docker containers and will be able process large files locally using those containers. Now, to accomplish this, BD clients will need to find out the list of extractors that can process a given file. The Extractor Info Fetcher service comes into picture here.

Feature Description

Extractors are currently linked to filetypes through its use in routing keys. For finding the extractors that can process a given file, the clients will first call a service (let's call it *"Extractor Info Fetcher Service"* for the time being) through the BD-API endpoint **/extractors**. This service will return a list of extractors (including details like docker image name, extractor name, etc.) that can process files belonging to a given file type. Depending on which instance of BD-API is used (Dev, Prod, etc.), the returned list of extractors should be associated with that instance. I.e. only those extractors that are bound to a particular Brown Dog instance should be returned based on a request. Ideally these extractors should be either currently running or available, i.e., they shouldn't be obsolete.

The service will need to know the file type of a file to find the extractors that can process it. There are two options here; either the client can find out the mime type of a file and send it to the service or the client can send the file extension and the service can find out the mime type based on the extension. Each result entry returned by the service should contain extractor name, extractor id, docker image name and git repository name.

Possible Sites for the Service

There are many places within the Brown Dog environment where this service can reside.

Independent Service

If implemented as an independent service, this service will run from a VM but will be accessible only through Fence (BD-API) since the service as such won't have any authentication or authorization in place. The advantage here is that it's implementation can be changed independent of other module like Clowder and Tools Catalog. Now, this also means adding one more component to Brown Dog which adds to the overhead of setting it up.

Inside Tools Catalog

Tools Catalog manages the tools (extractors and converters) that are part of Brown Dog. It's like an app store were all the tools are stored forever for achieving tool reuse. The service can also be part of Tools Catalog since it is closely associated with the tools that the Tools Catalog manage.

Inside BD-Clowder

BD-Clowder is the web application that does the Brown Dog content management. It stores the files that are submitted for processing by different BD clients, sends those files to be processed by remote extractor services, and stores the generated metadata and other auxiliary information for future use. Currently a Mongo database is being used as the data store. In this Mongo database, inside a collection titled *extractors.info*, the details about each extractor that gets registered with this Clowder instance is getting stored. Those details comes from the *extractor_info.json* file that is now part of every extractor. In future there is a plan to include filetypes on which an extractor will fire inside the *extractor_info.json*. This means that all information needed for finding the extractors that can process a file type will be available in the extractor itself. So, the extractor info fetcher service if it resides inside BD-Clowder can query the database and obtain the needed the list of extractors with its details.

Implementation Details

The prototype of this feature has been developed using Python Flask app. It is easy to develop and debug API end points using Flask.

Server side vs client side calculation of filetype

There are two methods to calculate the filetype. One method is that clients can find the filetype of a file and send them to the service. Though when the clients are written in languages like Python, this is easy, in some other clients, this may be difficult. Another option is that clients will send in the file extension and the service can figure out the filetype from the file extension. This gives much more control to the service. If file extensions cannot be uniquely mapped on to filetypes, this approach may not probably work, since in that case the client will be in a better situation to figure the filetype by scanning the input file header. This also means that the logic for calculating MIME type need not be implemented at multiple places (clients). In both these situations there are cases where the MIME type may not be standard and it may not be possible to find those from file extensions.

Finding extractors that are running (or active) at any time

It is very important that this service returns only those extractors that are running or are currently active (in the sense that the extractors should get fired if the file is submitted to a Brown Dog extraction service). Now, there is a tricky situation here. If there are consumers in a RabbitMQ extractor queue, then it means that it is running, but this is not always the case. Brown Dog uses its Elasticity module to automatically change the number of consumers based on request. It has a provision to set the number of consumers to be 0. This can be helpful to conserve resources for especially those extractors that use a lot of disk space / memory but are rarely used. Now, this provision means that even if the number of consumers is 0, it doesn't really mean that the extractor is not a current one. It can be because of its elasticity module. What seems practical is that if we maintain the Brown Dog RabbitMQ queues properly, i.e., by deleting unused or old extractor queues, RabbitMQ API can be used to find those extractors that are presently in active use.

Finding extractors that are bound to a specific Brown Dog instance

Currently, we have two instances of Brown Dog, namely the *development (dev)* and *production (prod)* instance. In future as Brown Dog gets adopted by other institutions, some of them may want to have their own internal instances of Brown Dog. This means that when a user submits a file for processing, it should be processed by a specific instance of Brown Dog on which it was intended to be processed. For the Extractor Info Fetcher service, this means that when returning a list of extractors that can a work on a given file type, it should also take into consideration the Brown Dog instance to which those extractors belong. In RabbitMQ for Brown Dog, we use specific virtual hosts for specific instances. Extractors can register themselves with multiple instances of Clowder (e.g. BD-Clowder-Dev, BD-Clowder, etc.) using the registration API end point. This information can also be used to find extractors that are bound to a Brown Dog instance. But, when a new BrownDog instance (synonymously a new Clowder instance behind the scenes for managing data) is created, extractor code (*extractor_info.json*) has to be modified to register it with this new instance. This can make things less scalable.