WebSphere Application Server

WebSphere Liberty Batch Exploring the REST API

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102632** under the category of "White Papers"

Version Date: March 21, 2016
See "Document change history" on page 22 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software

Written by:
David Follis
IBM Poughkeepsie
845-435-5462
follis@us.ibm.com

Don Bagwell
IBM Advanced Technical Sales
301-240-3016
dbagwell@us.ibm.com

Many thanks go to the WebSphere Batch team for getting all this work done.

Table of Contents

Introduction

Seems like lately if you want to be a player in the API Economy you need to have a REST API. When we developed Java Batch (JSR-352) support in WebSphere Liberty we didn't want to be left out, so we provided a REST interface to help manage the batch jobs.

There is certainly documentation provided in the Knowledge Center about the API and you can use the API Discovery feature of Liberty to explore the interface interactively (which is really pretty cool too). But it can be a lot to absorb and sometimes it is difficult to know where to start.

In this paper we'll try to find an easy way to approach the API and understand the basics.

Wow That's A Lot of Stuff

When you first take a look at the documentation for the API your reaction is likely to be similar to mine..”Wow...that seems like a lot of APIs to manage batch jobs.” The trick is to break it down into pieces. It isn't nearly as complicated as it seems when you first look at it.

Versions

One of the reasons there seem to be so many URIs is the versions. Simply put, we didn't plan ahead like we should have. When we started out we had some nice simple URIs. For example, you could drive a GET on `/ibm/api/batch/jobinstances` to get a list of job instances.

But a little while after we had shipped the batch support we wanted to make a change and add some new features to some of the services. Well, you can't just change the interface. That breaks people with applications using the old interface. So we decided to add a version into the URI. Well the new one couldn't very well be version 1, so it had to be version 2. Thus we added support for `/ibm/api/batch/v2/jobinstances`.

But what happened to version 1? Anybody looking at the API years from now would be confused as to why some URIs don't have versions and some do and there is no version one. Thus we decided to also support a 'v1' URI for any of the URIs we had supported originally. That means you can drive `/ibm/api/batch/v1/jobinstances` and get the exact same thing as the URI without the version at all.

It also means that at the moment we have almost twice as many URIs in the API as we really need since there are the base and 'v1' URIs that do exactly the same thing.

Probably the best thing to do if you are using the API is to ignore the original URIs and just use the 'v1' flavor. Then it will be clear going forward as we introduce other improvements just which version of which URI you are using.

GET and PUT and POST and DELETE

REST has a small set of main verbs. Looking at the list of URIs it is easy to lose track of the verbs and yet they are very important. In fact, if you ignore all the URIs and just look at the verbs you'll see that almost all of the interface is a GET. That's because there are only a few things you can do that actually cause something to happen. Almost the entire mass of the interface is just to allow you to get various bits of information back. When we look at the details of the API we'll start with the actual actions and then take a stroll through the various things you can GET.

Actually Using the API

Before we go crawling around in the API you might wonder how you'd actually drive it. There are quite a few ways to drive the REST API. Here are a few of them...there are surely more.

batchManager Command Line Interface

WebSphere Liberty's batchManagement feature includes a Java utility called batchManager that drives the REST interface under the covers. You don't need to know anything about the REST API to use it, which of course makes it the easiest way to use the interface!

Write Your Own Java Client

If you don't like the way batchManager uses the REST API you're free of course to write your own Java application, or perhaps to integrate use of the batch REST API into some pre-existing Java application.

cURL

This is a handy, free utility for sending commands using the URL syntax. It can be an easy way to include REST commands into a script or to just experiment with the API from the command line. There are easier ways though.

Web Browser Plugins

Some web browsers have available plugins that can act as a REST client. Capabilities and ease-of-use depend on the plugin and browser. Search the web for your browser and 'REST plugin' and see what you find.

API Discovery

There is a very cool feature in Liberty called apiDiscovery. If enabled it can help expose the Swagger 2.0 documentation for REST services supported by the server (such as the batch REST services). There is quite a lot of capability in this feature, but the one most useful to us here is the interactive discovery interface.

With the feature enabled you can use a browser to head to `https://host:https_port/ibm/api/explorer`. What you will see is a rendered HTML page much like the standard petstore.swagger/io sample. You can see all the verbs and URIs. Expanding any given URI you can also see a prototype for the required JSON payload (if any). If you like you can go farther and actually drive the interface, providing a payload or in some cases just filling in the prompts.

For me this was a great way to just go exploring the interface.

Others

There are certainly other ways to do this. You may write in other languages or, given the discovery capabilities, have other API management tools that let you view and work with the API.

Submitting a Job

We said earlier we'd start with the actions and move to the more passive 'GET' verbs later. The best thing to do first is to submit a job. All the other things you can do relate to jobs you are running or have already run, so it is best to get something out there first.

Submitting a job is the only POST action since it is the only one that creates something. The URI to use is `/ibm/api/batch/v1/jobinstances`. That's because what you're going to create is a job instance (and also an execution). Which means we should probably pause for a moment and talk about...

Job Instances and Job Executions

If you're familiar with the JSR-352 specification you already know this, but in case you don't the API will baffle you without this concept so it is worth spending a moment. Those of you who are comfy with it can go get some coffee.

Ok, so this is pretty easy. Every time you submit a job (any job) you create an *instance* of that job. If today at 2:00 PM I submit my purgeMyEmail job then that's the 2:00 today instance of that job. If I submit the job again tomorrow at 2:00 then that is a different instance of the job.

When the job actually runs you have an *execution* of that job. In the normal case where everything goes ok you'll just have one execution for each instance. But what if things go badly? If the job fails it may be possible to restart it. You're running the same *instance* of the job. This is my 2:00 today instance, but it is the second try to run it, restarting part way through. That second try gets a new *execution* for this instance.

You can have multiple instances of a job, meaning you've run that particular job more than once. And for any given instance of the job you probably only have one execution but restarts can lead to you having more than one execution for an instance.

Got it?

Parameters

There aren't a lot of required parameters to submit a job and you may not even need them all. You have to either specify applicationName or moduleName. This is basically the name of the .war file you used to package the batch application. If you specify moduleName we strip off the .war suffix and use that for the applicationName or if you specify applicationName and not moduleName we create the moduleName by appending a .war to the end of the applicationName. Take your pick.

If your module is an EJB then you also need to specify componentName. Basically you have to give us enough information to find the application and create its context so we can load bits of the application to run it.

You also have to point us to the JSL file (an XML file) that defines the job. Normally this is packaged with the application and you supply the XML file name (without the ".xml") as jobXMLName. Alternatively you can include the JSL with the REST request and there are two different ways to do that. See the documentation in the Knowledge Center for details. The Knowledge Center article is rwlp_batch_rest_api.

Finally, you can supply any job parameters that are required. These are name/value pairs that may be substituted into the JSL to provide appropriate execution-time values for job properties. Alternatively you may need them as message properties in a multi-server Liberty Batch configuration to make sure your job is routed to the correct Execution server.

As an example, I submitted a simple job in the BatchProjectWar.war file using the Job1.xml JSL file with just this set of parameters:

```
{
  "moduleName": "BatchProjectWAR.war",
  "jobXMLName": "Job1"
}
```

Results

A JSON string is returned as a result of submitting your job. The result I got back is pasted below. We'll go through this in detail as some of it is important to things we'll try later on.

```
{
  "jobName": "Job1",
  "instanceId": 0,
  "appName": "BatchProjectWAR#BatchProjectWAR.war",
  "submitter": "bob",
  "batchStatus": "STARTING",
  "jobXMLName": "Job1",
  "instanceState": "SUBMITTED",
  "_links": [
    {
      "rel": "self",
      "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobinstances/0"
    },
    {
      "rel": "job logs",
      "href":
"https://a.bb.cc.ddd:pppp/ibm/api/batch/jobinstances/0/joblogs"
    }
  ]
}
```

What does this tell us? Some of it we already know. The jobXMLName is the jobXMLName we supplied and the appName you can see is derived from the moduleName we supplied. The instanceId is zero. That's the job *instance* as we discussed earlier. This is the first job I've submitted to this environment so the instance ID counter starts at zero. Remember this value because we'll need it later on.

We also see the userid (bob) that I was using to connect to the server to submit the job. The jobName at the top comes from the JSL file (and happens to match the name of the XML file). We also have two status values: batchStatus and instanceState. The batchStatus is the status of the job at the time the REST request returned. At that moment the job was considered to be starting. Given the job does nothing at all, it is (I hope) done by now and so this state is quite old. Similarly the instanceState (the state of our job instance) tells us the job instance has been submitted.

Finally the API gives us two handy URLs we can use (yes, I blocked the ip address and port number of my server...leave me alone :-) These are actually URIs that are defined as part of the REST interface and you could build them yourself using the instanceID above. We'll get to them later, but having them nicely pre-built in the response can make it easier to drive the request.

Stopping a Job

By now my simple little job has finished so we have no chance to stop it. But suppose it was long running and we decided we needed to try to stop it. How would we do that? You use the PUT verb because that means to update or replace something.

The URI to use is: `/ibm/api/batch/v1/jobinstances/{jobinstanceid}`. Replacing the curly braces and jobinstanceid with your actual instance ID (a '0' in our case).

Now it turns out that the same URI is used to stop or restart a job. In both cases we're referring to a job instance and the right verb to use is PUT. Somehow we have to tell it what we want done, so the API requires a parameter (action) whose values can be stop or restart. That means to stop a job you'd do something like this: `/ibm/api/batch/v1/jobinstances/0?action=stop`

That all seems fine until you think about it a little bit. An instance isn't really running. So how can you stop it? Seems like you would want to stop an execution. Well actually you are. If you stop an instance the server will find the latest execution for that instance and stop it.

If you happen to know the execution ID (which we'll show you how to find later) then you can also direct a PUT against it with this URI: `/ibm/api/batch/v1/jobexecutions/{jobexecutionid}`. Again you'll need to specify an action=stop to tell the server what you want done with that job execution.

Restarting a Job

If we had managed to stop our job or if it had failed on its own we might want to restart it. This is done exactly the same way as we stopped the job just above. The only difference is that you specify an `action=restart` parameter instead of `action=stop`.

Here again if you think about it a bit you'll realize it makes no sense to restart an execution. That execution is over. What you really want to do is restart the job instance and create a new execution. And that's really what happens. If you use the execution URI and an execution ID the server just finds the related job instance and restarts it.

There's one more quirk to the job restart process and that has to do with job parameters. When you restart a job, do you want to use the same parameter values you supplied when you started it the first time or do you want to provide updated values? You can do either one or a combination of both.

Parameter reuse is controlled by another parameter you supply with the URI (along with the `action=restart`). This parameter is `reusePreviousParms` and it is a boolean. If you use the default value of false none of the previously supplied values are used and you can supply all new values as part of the request.

If you specify `reusePreviousParms=true` then the server will submit the job using the parameter values you supplied when you submitted it previously. But you can still provide parameter values on the restart to override the remembered parameters. The result is a merge of the two sets of parameters (remembered and supplied) with the parameters supplied on the restart overriding the remembered values.

Purging a Job

Our last action request is to purge a job. I'd suggest you don't play with this one until you've experimented with the various GET requests or else you'll need to submit another job to GET information about.

Purge is an interesting function because it is totally outside of the JSR-352 specification. The specification merely requires that there be a Job Repository to keep track of things. It doesn't specify how it is implemented and therefore provides no guidance on how you remove things from it. Thus there is no JobOperator method you can use to remove an old job from the repository. We often think of the REST interface as essentially 'wrapping' the JobOperator API, but in this case there is no equivalent and it is an entirely independent thing.

A job purge is done with the DELETE verb. That certainly make sense. The URI to use is just `/ibm/api/batch/v1/jobinstances/{jobinstanceid}`. You can't purge just an execution. You have to purge the whole instance. You just specify the job instance ID you want to purge.

There is one optional parameter you can supply. Purge normally removes all memory of the job from the Job Repository, and it also removes any job logs that go with all the executions of the job. It is possible you might want to purge the job from the repository but leave the job logs alone. Or perhaps the job logs are already gone because you have some other process that archives them or otherwise cleans them up. If removing the logs doesn't work, the purge will fail. So if the logs are gone already you need to tell the purge process not to try to delete them. That's done by specifying the parameter purgeJobStoreOnly=true.

But wait! There's more! The purge interface we've described here allows you to purge the artifacts for one job instance. What if you have hundreds (thousands!) of jobs to purge? Calling the REST API over and over again purging one job at a time could take quite a while. To support purging more than one job at once we introduced version two of this service.

You invoke it using the same DELETE verb with the v2 URI of: `/ibm/api/batch/v2/jobinstances`. Note that you don't specify a job instance ID. Instead you are going to give the service parameters that tell it about the jobs you want to purge.

There are a lot of different parameters you can specify to control which jobs are purged and again I'll refer you to the official Knowledge Center documentation for the whole list (mostly in case it changes as we add things in the future...don't want this paper to get out of date too fast). I'll just point out one easy thing you can do is specify a list of job instance IDs using the jobInstanceId parameter set to a comma separated list of IDs or a dash separated range of IDs.

A word of caution...before you use the very powerful V2 DELETE processing to purge a lot of jobs, you might use the equally powerful V2 GET job instances processing to get to a list of the jobs you're going to purge....just to be sure it is going to do what you think. The difference between purging job one **and** job one thousand and purging jobs one **through** one thousand is a single character (1,1000 vs. 1-1000). Type carefully. We'll go over the V2 GET job instances service later.

Tunneling Inward – Instances to Instance to Executions to Steps...

Job Instance List

In our evolving story we have submitted one job and we know what it's instance ID is, but suppose we just wondered what jobs had been run in our environment. Is there an API to get a list of all the job instances? Of course there is.

Every thing we do from here on will use a GET verb. The URI we need for a list-jobs request is: `/ibm/api/batch/v1/jobinstances`. That's going to return a JSON string something like this:

```
[
  {
    "jobName": "Job1",
    "instanceId": 0,
    "appName": "BatchProjectWAR#BatchProjectWAR.war",
    "submitter": "bob",
    "batchStatus": "COMPLETED",
    "jobXMLName": "Job1",
    "instanceState": "COMPLETED",
    "_links": [
      {
        "rel": "self",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobinstances/0"
      },
      {
        "rel": "job logs",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobinstances/0/joblogs"
      }
    ]
  }
]
```

The first thing to notice is those square brackets on the outside. That means this is an array so really we're looking at a list of jobs that happens to contain one element.

For each job we get back all the same information that we got as the response when we submitted the job originally...almost. You'll note that both the instance state and batch status values are now COMPLETED because the job finished (long ago).

Job Execution List

I've told you from the start that our job instance also had a job execution but we haven't seen anything about it yet. Next we'll look at the REST service that allows you to get a list of executions for a particular instance. The verb, of course, is GET. The URI is `/ibm/api/batch/v1/jobinstances/{jobinstanceid}/jobexecutions`. I filled in our job instance ID (0) and here is what I got back:


```
[
  {
    "jobName": "Job1",
    "executionId": 0,
    "instanceId": 0,
    "batchStatus": "COMPLETED",
    "exitStatus": "COMPLETED",
    "createTime": "2016/02/22 14:58:23.274 -0500",
    "endTime": "2016/02/22 14:58:23.370 -0500",
    "lastUpdatedTime": "2016/02/22 14:58:23.370 -0500",
    "startTime": "2016/02/22 14:58:23.298 -0500",
    "jobParameters": {},
    "restUrl": "https://a.bb.cc.ddd:pppp/ibm/api/batch",
    "serverId": "localhost/F:/Scratch/WLP-20160211/wlp/usr/server1",
    "logpath": "F:\\Scratch\\WLP-
20160211\\wlp\\usr\\servers\\server1\\logs\\joblogs\\Job1\\2016-02-
22\\instance.0\\execution.0\\",
    "stepExecutions": [
      {
        "stepExecutionId": 0,
        "stepName": "Step1",
        "batchStatus": "COMPLETED",
        "exitStatus": "COMPLETED",
        "stepExecution":
"https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0/stepexecutions/Step1"
      }
    ],
    "_links": [
      {
        "rel": "self",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0"
      },
      {
        "rel": "job instance",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobinstances/0"
      },
```

```
      {
        "rel": "step executions",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0/stepexecutions"
      },
      {
        "rel": "job logs",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0/joblogs"
      },
      {
        "rel": "stop url",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0?action=stop"
      }
    ]
  }
]
```


Wow..that's a lot. Notice first of all that this is also an array. We only have one execution so there is just one entry. If we had two or more this would be even longer. So what's in here? We'll take it in pieces and I'll re-paste in the sections as we talk about them.


```
    "jobName": "Job1",
    "executionId": 0,
    "instanceId": 0,
    "batchStatus": "COMPLETED",
    "exitStatus": "COMPLETED",
    "createTime": "2016/02/22 14:58:23.274 -0500",
    "endTime": "2016/02/22 14:58:23.370 -0500",
    "lastUpdatedTime": "2016/02/22 14:58:23.370 -0500",
    "startTime": "2016/02/22 14:58:23.298 -0500",
    "jobParameters": {},
    "restUrl": "https://a.bb.cc.ddd:pppp/ibm/api/batch",
    "serverId": "localhost/F:/Scratch/WLP-20160211/wlp/usr/server1",
    "logpath": "F:\\Scratch\\WLP-
20160211\\wlp\\usr\\servers\\server1\\logs\\joblogs\\Job1\\2016-02-
22\\instance.0\\execution.0\\",
```


Well there is our jobName as before. And now we know our execution ID. It is also zero. That should tell you that instance IDs and execution IDs are not mutually unique. If you have an ID you have to know what kind it is in order to use it.

We can see the batch and exit status are completed, but now we have some time stamps. We can see the time the job was created, when it ended, when the Job Repository entry was last updated, and when the job started. It is sort of an odd order, but there is good stuff in there. You can easily

see our tiny little job ran in just 0.072 seconds. That -0500 floating to the right is the GMT offset as these are local times.

We can see we submitted the job with no job parameters. If there were parameter values they would show up here. That can be useful if the job had a problem and you wonder how it was invoked.

The restUrl is the base URL to use to get information about this job. You had to already know this to get it back so honestly I'm not sure why it is in here. Perhaps if you put this into a file somewhere it might be nice to have later.

The serverId is path to where the server.xml file is for the server that ran the job. You can see it was on my F: drive.

And finally in this section we have the path to where the joblog files live. We'll see later that you can use the REST API to fetch the joblog, but it might be convenient to know where the actual file lives in case you need to get to it.

```
    "stepExecutions": [
      {
        "stepExecutionId": 0,
        "stepName": "Step1",
        "batchStatus": "COMPLETED",
        "exitStatus": "COMPLETED",
        "stepExecution":
"https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0/stepexecutions/Step1"
      }
    ],
```

Now we have some information about the steps in the job. Again this is an array because a job can have multiple steps. Our boring little job just had one. Steps have execution IDs also and our step's ID was zero. The stepName (Step1) comes from the JSL. The batch and exit status for the step are both COMPLETED. If our job had multiple steps and something interesting had happened you might be able to tell where things went wrong from looking at the statuses from the various steps.

Finally we have a URL we can use to get information about this specific step. There are actually a few ways to get information about a step depending on whether you know the step name or the step execution ID. We'll get there.

```
    "_links": [
      {
        "rel": "self",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0"
      },
      {
        "rel": "job instance",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobinstances/0"
      },
```

```
      {
        "rel": "step executions",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0/stepexecutions"
      },
      {
        "rel": "job logs",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0/joblogs"
      },
      {
        "rel": "stop url",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0?action=stop"
      }
    ]
  }
```


The final section in the response to our query about the execution is a set of useful links to get to other information. These are really just there to help you out. You can form the right URI from information you know (or can find out) but having them pre-formed in the response is easier and avoids a chance of making a mistake building the URI.

The Steps for a Job Execution

The list of executions included an array in each one that shows some basic information about each step in the execution. But we can get more information about the steps. We'll use yet another GET verb and a URI like this: `/ibm/api/batch/v1/jobexecutions/{jobexecutionid}/stepexecutions`. We'll fill in our job execution ID (zero) and I got back this:


```
[
  {
    "stepExecutionId": 0,
    "stepName": "Step1",
    "executionId": 0,
    "instanceId": 0,
    "batchStatus": "COMPLETED",
    "startTime": "2016/02/22 14:58:23.351 -0500",
    "endTime": "2016/02/22 14:58:23.368 -0500",
    "exitStatus": "COMPLETED",
    "metrics": {
      "READ_COUNT": "0",
      "WRITE_COUNT": "0",
      "COMMIT_COUNT": "0",
      "ROLLBACK_COUNT": "0",
```

```
      "READ_SKIP_COUNT": "0",
      "PROCESS_SKIP_COUNT": "0",
      "FILTER_COUNT": "0",
      "WRITE_SKIP_COUNT": "0"
    },
    "partitions": []
  },
  {
    "_links": [
      {
        "rel": "job execution",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0"
      },
      {
        "rel": "job instance",
        "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobinstances/0"
      }
    ]
  }
]
```


Yet again we have an array since this is information for every step in the execution. Our simple job just has one. But this time we have a lot more stuff than we got about the steps in the execution information. We still have our various IDs (instance, execution, step execution) and the step name and batch/exit status strings.

But now we also have the start and end timestamps for the step. And then we have a handy block of metrics about the step. Our job was a batchlet so the values are all zero, but if you have a chunk step these metrics will tell you a lot about what went on processing the step.

Next we have another array that contains information about step partitions. My job didn't have any, but if it did we'd get similar information for each partition that we have for the step as a whole: start/end timestamps, statistics, and batch/exit status strings.

And finally we have more links to help you find things. In this case to the instance and execution to which these steps belong.

Other Ways to Get the Same Information

With the exception of the joblog, we've already seen all the information you can get back about your jobs. What remains are a set of REST services you can use to get that information starting with different things. Let's review what we've done:

1. We got a list of all the job instances in the Job Repository.
2. For one of those job instances, we got a list of all its job executions.
3. For one job execution, we got detailed information about the steps that ran in that execution.

That's a great way to work from the outside in. We knew nothing at all and found our way to detailed information about a step in a specific execution of a specific job instance. But we started from a list of all the job instances the environment knows about. That could be a lot.

Remember back when we first started by submitting a job? The returned JSON string from the PUT request included the job instanced id. We've seen we could just skip item #1 and just get a list of its job executions. What else can we find out with just a job instance id?

Using the Job Instance ID

If you have a job instance ID you can drive a GET with the URI: `/ibm/api/batch/v1/jobinstances/{jobinstanceid}`. This gets you the same information that was returned when we submitted the job. Except the information is current. Remember the batch and exit status that came back from the submit showed the job was just getting started. A call to this service will tell you the current state of the job.

Finding an Execution by Count

Earlier we used the instance ID to find a list of executions and then used the execution ID to get information about that specific execution. Is there an easier way? Yes!

Suppose you submitted a job and you have the instance id. The job has problems and you've restarted it a time or two. You don't know the execution IDs that resulted from those restarts. But you want information about the second attempt to restart it. You can ask for it that way using a GET with URI: `/ibm/api/batch/v1/jobinstances/{jobinstanceid}/jobexecutions/{jobexecutionnumber}`.

An execution number is the 'try' that you are interested in. The number starts at zero for the first try and just increments for every restart. If we want the second attempt to restart, then the execution number we want is '2'. The information you get back is the same as you would get using the actual execution id.

Getting Information about a Step

We've gotten detailed information about all the steps in an execution, but to do it we needed the execution id. And what if I want information about just one step and not all the steps in an execution? We saw that each step has an execution id. Do we need to know that to reference the step? It is actually pretty easy.

If all you know is the job instance ID (returned by submit) and the execution number you care about and the name of the step (from the JSL) then you can get information about that step in that execution. Just drive a GET against URI: `/ibm/api/batch/v1/jobinstances/{jobinstanceid}/jobexecutions/{jobexecutionnumber}/stepexecutions/{stepname}`.

Fill in the instance id, the execution number, and the actual step name and you will get back the same detailed step information we saw from the list of steps for the whole execution.

If you do happen to know the step execution ID you can use it also to get information about that execution of that step. Drive a GET with this URI: `/ibm/api/batch/v1/stepexecutions/{stepexecutionid}`.

You don't need anything but that one ID because step execution IDs are unique across all the steps in all the jobs in the Job Repository.

Using the Job Execution ID

We've already seen how to get a list of executions for a particular job instance. But what if you have multiple executions and you are really just interested in one of them? If you have the execution ID you can use it in several ways.

First of all, you can get the same information we had earlier in the list of executions, but just about one by driving a GET with URI: `/ibm/api/batch/v1/jobexecutions/{jobexecutionid}`.

This is sort of a weird case but if you happen to have an execution ID and have somehow lost the job instance it goes with, you can use the execution ID to get information about the owning instance. Do that by driving a GET with URI: `/ibm/api/batch/v1/jobexecutions/{jobexecutionid}/jobinstance`. That yields the same basic job instance information as we've seen before.

Finally, if you have an execution ID and you are wondering about a particular step, you can use the step name to get it. This is similar to a service we saw just above using the job instance ID and the execution number. In this case though, we have the execution ID and not the number. It gets a bit confusing. Sometimes it is easier to intuitively know the execution number, but if you are driving the REST services from some graphical interface it may be simpler from the underlying code to know the execution ID itself. The services support both paths. Oh yes, so to get step information with an execution ID you drive a GET with URI: `/ibm/api/batch/v1/jobexecutions/{jobexecutionid}/stepexecutions/{stepname}`.

Searching for Jobs

Way back sort of near the start of this paper we talked about the v2 version of the purge service and suggested you might want to use the V2 GET job instances services to make sure you knew what you were deleting before you deleted it. I promised to talk about that service later. I'll bet you thought I forgot about it. Well, you're right...I did.

It is really pretty easy to use. You drive a GET with URI: `/ibm/api/batch/v2/jobinstances`. You specify as parameters the qualifiers to restrict which job instances are you interested in. The service will return an array of job instance information about all the job instances in the Job Repository that match the qualifiers you specified.

Earlier we talked about the ability to specify a list or a range of job instance ids. And we mentioned that there are other qualifiers that are documented in the Knowledge Center in the rwlp_batch_rest_api article.

We should probably talk about one other qualifier that some people have found to be confusing.

You can request that the service show you information about jobs relative to the creation time of the job. You do that by specifying the createTime parameter, an comparator (<, >, =), and a number of days. For example, createTime>3d.

Well what does that mean? You need to read it as, “I want to know about jobs whose creation timestamp value is greater than a timestamp of three days ago.” If you ran a job yesterday, then its creation time is more recent (the time value is greater than) a timestamp from three days ago so it would be included in the output. If you ran a job a week ago, its creation timestamp value is less than a timestamp from 3 days ago so it would not be included.

There is a tendency to sometimes read the createTime value as an 'age' as in “I want jobs older than 3 days,” but that's not how it works. You have to think in timestamps.

Remember to look at the Knowledge Center for some of the other cool ways you can filter what jobs get returned.

Limiting the results

This might be a good time to mention limiting the size of results. Some of the services we've described, and especially this one, can return rather extensive output. To avoid accidentally returning a massive response (tell me all about the four million jobs in the repository) there are some parameters you can specify to limit the response.

The first of these is `pageSize`. This parameter controls how many records (e.g. job instances) the interface will tell you about at one time. To protect you from accidentally getting a much larger result set than you were prepared for, the default value for `pageSize` is 50.

The second parameter that influences the response content is `page`. This tells the service which page to start returning results from based on the `pageSize`. So if the repository has information about 200 jobs to return and you take the defaults, you will get the first (really the zero) page back which is the first 50 jobs. If you call the service again, you should specify `page=1` to get information about the next 50 jobs, and so forth.

Of course the Job Repository isn't locked while you are doing this. We don't maintain a cursor for you in the repository tables.

Getting the Joblog

Every job that runs in Liberty Batch produces a job log. That log might just contain messages from the batch container about the job, but it might also contain messages issued from the batch application itself. If your job uses partitions, each partition produces its own separate log. It is also possible the log might end up split across several files if it gets to be too big.

We'll take a quick look at what actually ends up in the log (besides application messages) in a bit. First let's see how you get them.

Finding out how to find the joblogs

As with much of the Batch REST interface, it depends what information you have and what specifically you want to look at. There are three different URIs you use with a GET verb to find out about the logs.

1. `/ibm/api/batch/v1/jobinstances/{jobinstanceid}/joblogs`
2. `/ibm/api/batch/v1/jobexecutions/{jobexecutionid}/joblogs`
3. `/ibm/api/batch/v1/jobinstances/{jobinstanceid}/jobexecutions/{jobexecutionid}/joblogs`

The first one gives you information about the job logs for the job instance. There could be several depending on how many executions you have. The second one gives you information about the job logs for the execution you specify. And the last lets you specify both the instance and execution id, although this is really a bit redundant since the execution ID is really enough to uniquely identify the one you care about.

What do you get back? You get a JSON string (as before) that is an array of URLs you can use to get to the actual job logs themselves. It will start off with something like these two:

```
{
```

```
    "rel": "joblog text",
    "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0/joblogs?type=text"
  },
  {
    "rel": "joblog zip",
    "href": "https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0/joblogs?type=zip"
  },
```

These two are handy links to get back all the pieces of the job log for the execution. Both return you all the pieces of the log, whether from rollover because the log became too big or from separate logs written by the partitions of a partitioned step. If you specify type=text you get back all the logs merged together into one text document. If you specify type=zip then the various files are zipped together and you can save and un-zip the file to examine the various pieces separately.

You might notice that these URIs are really just the second of our three forms listed above with the addition of the type= parameter. So you don't really need to use the service to get these URLs if you want the whole log. You can easily just build them yourself.

The JSON response will also contain something like this:

```
    {
        "rel":"joblog part text",
        "href":"https://a.bb.cc.dd:pppp/ibm/api/batch/jobexecutions/0/joblogs?
part=part.1.log&type=text"
    },
    {
        "rel":"joblog part zip",
        "href":"https://a.bb.cc.ddd:pppp/ibm/api/batch/jobexecutions/0/joblogs?
part=part.1.log&type=zip"
    }
```

This part contains links to the actual pieces of the logs. In this case we can see that these links allow us to retrieve part.1.log (the first part) of the log for the main job. Other parts or partitions with logs will result in other links in this section. You can fetch these pieces either as straight text or the server can zip it up for you to minimize bandwidth used to transfer it.

What's in a Job Log?

This isn't strictly part of the REST interface, but we're on a roll. Here's a very simple job log from my very simple job.

```
[2/26/16 12:43:56:359 EST] com.ibm.ws.batch.JobLogger
============================================================
Started invoking execution for a job
 JobInstance id = 0
 JobExecution id = 0
 Job Name = Job1
 Job Parameters = {}
============================================================

[2/26/16 12:43:56:359 EST] com.ibm.ws.batch.JobLogger
CWWKY0009I: Job Job1 started for job instance 0 and job execution 0.
[2/26/16 12:43:56:372 EST] com.ibm.ws.batch.JobLogger
============================================================
For step name = Step1
 New top-level step execution ID = 0
============================================================

[2/26/16 12:43:56:410 EST] com.ibm.ws.batch.JobLogger
CWWKY0018I: Step Step1 started for job instance 0 and job execution 0.
[2/26/16 12:43:56:433 EST] com.ibm.batch.sample.Batchlet1
Hello World!
[2/26/16 12:43:56:433 EST] com.ibm.ws.batch.JobLogger
CWWKY0020I: Step Step1 ended with batch status COMPLETED and exit status COMPLETED for job
instance 0 and job execution 0.
[2/26/16 12:43:56:435 EST] com.ibm.ws.batch.JobLogger
CWWKY0010I: Job Job1 ended with batch status COMPLETED and exit status COMPLETED for job
instance 0 and job execution 0.
[2/26/16 12:43:56:435 EST] com.ibm.ws.batch.JobLogger
============================================================
Completed invoking execution for a job
 JobInstance ID = 0
 JobExecution ID = 0
 Job Name = Job1
 Job Parameters = {}
 Job Batch Status = COMPLETED, Job Exit Status = COMPLETED
============================================================
```

You can see at the top some of the information we already knew about our job. We have the instance and execution IDs, the job name, and our parameters (well, our lack of parameters).

As the job runs it issues messages to report on the job's progress. The timestamps from the messages can help you see where time is being spent in the job.

Right in the middle you'll see “Hello World!” which is a message written by my application.

At the end of the job you see some wrap-up information including the same stuff from the start of the job plus the batch and exit status values.

Conclusion

That wraps it up for the Liberty Batch REST services. Even if you never actually write a client that uses the services directly, hopefully this tour through them has helped you understand how other interfaces (like the batchManagement command line interface) are interacting with the server.

Thanks for reading!

Document change history

Check the date in the footer of the document for the version of the document.

March 21, 2016	Initial Version
March 21, 2016	Add document number

End of WP102632