Article
Evgeniy Potapov · Oct 3, 2022 8m read

Automating the deployment of Adaptive Analytics

When we work with IRIS, we usually have the ability to quickly deploy a ready-to-use infrastructure for BI (data, analytical cubes, and IRIS BI dashboards) using modules. When we start using Adaptive Analytics, we typically want to have the same functionality. Adaptive Analytics has all the tools that we need. The documentation contains a description of how to work with an open web API. All interactions between the user interface and the engine also occur via the internal web API and can be emitted.

It is necessary to automate both processes: deploying Adaptive Analytics in a container and directly to the server system. For this purpose, the easiest way is to use bash scripts to work with API. The only third-party application we will need is a JSON Parser for JSON files named jq. You can install it using the following commands:

```
apt update
apt install -y jq
```

First of all, we need to log in to get an API access token. This token also applies to the methods of the engine itself. We must save the access token in a variable since now we will need it in almost every request. For a standard login and password admin/admin, the command will look like this:

```
TOKEN=$(curl -u admin:admin --lo
cation --request GET 'http ://localhost:10500/default/auth')
```

Next, we need an active engine to interact with the API. Without a license check, the engine is not available, so we provide it with a license. The web API does not have this option, so we will have to use the engine command:

```
curl --location --request PUT
'http://127.0.0.1:10502/license' --header 'Content-
Type: application/json' --data-binary "@$license"
```

The license variable contains the path to the license file. When we did some testing on different platforms and PCs of different capacities, we noticed one peculiarity. After the completion of the system boot script, Adaptive Analytics may have such a situation when the services have not yet started, but the initialization script has already given control to our script. In order to make sure that the engine is up and running, we organize the sending of the license in a loop until we receive a message that the license has been accepted. The code to do that looks like this:

```
RESPONSE="1"
TIME=1
echo "Waiting for engine ......."
while [ "$RESPONSE" != "200 OK" ]

do

  for license in /root/license/*

  do
```

```
    RESPONSE =$(curl --location --reque
st PUT 'http://127.0.0.1:10502/license' --header
'Content-Type: application/json' --data-binary "@$license" | jq -r '.status.message')

    sleep 1s

  done

  echo "$TIME seconds waiting"
  TIME=$(($TIME + 1))

done

echo "Engine started"
```

Printing the time variable is useful for debugging startup problems. As you can see, we are looping through the files in a specific folder so as not to be tied to the file name. We will use this approach again in the future.

Now we can interact with the web API, and we can upload our projects to Adaptive Analytics. We send all projects that are placed in the designated folder to the engine using the code below:

```
for cube in /root/cubes/*

do

  sleep 1s

  curl --location --request POST
"http://localhost:10500/api/1.0/org/defa
ult/"Authorization: Bearer $TOKEN" --header "Content-
Type: application/xml" --data-binary "@$cube"

  sleep 5s

done
```

If we wish our projects to be available for BI systems, we must publish them. Fortunately, there is also a method in API for this. Since projects get a random unique ID during import, first, we should parse these IDs into a variable:

```
PROJECTS_ID=$(curl --location --request G
ET
"http://localhost:10500/api/1.0/
org/default/projects" -- header
"Authorization: Bearer $TOKEN" | jq -r '.response[].id')
```

Then we need to iterate through all the projects in the variable and publish them.

```
for project in $PROJECTS_ID

do

  curl --location --request POST
"http://localhos
t:10500/api/1.0/org/default/proj
ect/$project/publish" --header "Authorization: Bearer $TOKEN"
  sleep 1s
```

```
done
```

Now we need to tell Adaptive Analytics how to connect to IRIS, and how the connections should be named so that our projects could pick them up. There is a method for this in the API, but it has a different URL address. As it turned out, this is a documented method of the engine. So if we want to use it, we need to access a different port. The method accepts connection information in the form of a JSON file, but it is not possible to export it in the same format from Adaptive Analytics. The JSON file that API returns to us upon request has some extra/missing fields. Here is the template for proper JSON file for IRIS connection:

```
{

    "platformType": "iris",

    "name": "name_you_want",

    "connectionId": "name_you_want",

    "overrideConnectionId":true,

    "extraProperties":

      {

        "namespace": "Your_namespase ",

        "udafMode": "customer_managed"

      },

    "aggregateSchema": "ATSCALE",

    "readOnly":false,

    "isImpersonationEnabled": false,

    "isCanaryAlwaysEnabled": false,

    "isPartialAggHitEnabled": false,

    "subgroups":

      [

        {

          " name": "iris",

          "hosts": "iris",

          "port": 1972,

          "connectorType": "iris",

          "username": "Your_username",

          "password":"Your_password",
          "isKerberosClientEnabled": false ,

          "queryRoles":

            [

              "large_user_query_role",

              "small_user_query_role",

              "system_query_role"

            ],

           "extraProperties": {},

          "connectionGroupsExtraProps": {}

        }

      ]

  }
```

It is worth mentioning about the “ udafMode ” parameter: we can set it to “ customer_managed ” if we have UDAF installed in IRIS, and “ none ” if UDAF is not installed. To learn more about what the UDAF is, why it is needed, and what pitfalls you may encounter when installing it, check this () article.

When we have prepared such files for all connections, we are supposed to load them from the folder:

```
for connection in /root/connections/*

do

  sleep 1s

  curl --location --request POST "http://localhost:10502/connection-
groups/orgId/default" - -header
"Authorization: Bearer $TOKEN" --header "Content-
Type: application/json" --data-binary "@$connection"
  sleep 5s

done
```

Next, we want to have the opportunity to connect Logi Report Designer to our Adaptive Analytics instance. That is why we need to simulate one of the steps of the first initialization of Adaptive Analytics. So, we set the port for hive connections:

```
sleep 1s

curl --location --request POST
"http://localhost:10502/organizat
ions/orgId/default" --header
"Authorization : Bearer $TOKEN" --header "Content-
Type: application/json" --data-raw '{"hiveServer2Port": 11111}'
```

In order not to exceed the connection limit in the community version of IRIS, we must limit the number of simultaneous connections for Adaptive Analytics. Based on our experience, to prevent exceeding the limit in the tested scenarios, out of 5 connections it is worth leaving 3. We simulate changing the settings using the code below:

```
for settings_list in /root/settings/*

do

  curl --location --request PATCH
'http://localhost:10502/settings' --header
"Authorization: Bearer $TOKEN" --header "Content-
Type: application/json" --data-binary "@$settings_list"
done
```

In the JSON file in the folder, write the following:

```
{

    "name": "bulkUpdate",

    "elements": [

        {

            "name": "connection.pool.group.maxConnections",

            "value": "3"

        },

        {

            "name": "connection.pool.user.maxConnections",

            "value": "3 "

        }

    ]

}
```

According to my experience with Adaptive Analytics, this limitation does not affect performance much.

After changing the settings, you need to restart the engine:

```
curl --location --request POST
'http://localhost:10500/api/1.0/referrerOrg/default/support/service-
control/engine/restart'
```

Now we have everything we need to automatically prepare Adaptive Analytics for work. For your convenience, we have published the above-mentioned scripts in the https://github.com/intersystems-community/dc-analytics/blob/master/atscale-server/entrypoint.sh

In this repository you will also find scripts for working with the aggregate update schedule. Aggregates are special tables generated by Adaptive Analytics when UDAF is configured in IRIS. These tables store the cached results of aggregation queries, which subsequently speeds up the work of BI systems. Adaptive Analytics has an internal logic for updating aggregates, but it is much more convenient to control this process yourself.

We have had some situations when these tables were storing data that was obsolete for several days. Accordingly, Adaptive Analytics returned values that were noticeably different from the real ones. To avoid such a case, we set up a daily update of the aggregates before the time when we update the data in the BI systems.

You can configure updates on a per-cube basis in the web interface of Adaptive Analytics. Then you will be able to use scripts to export and import schedules to another instance or use the exported schedule file as a backup. You can also find a script in the application to set all cubes to the same update schedule if you do not want to configure each one individually.

The last thing that is convenient to automate is the backup of the projects themselves from Adaptive Analytics at the development stage. To set up this automation, we wrote another OEX application that saves your selected projects to the Git repository once a day.

The beginner's guide describes the process of creating a widget in a new report file. You can insert a chart or table into an existing report too if you use the Insert tab or Component window.

#Adaptive Analytics #InterSystems IRIS BI (DeepSee)

Source URL:https://community.intersystems.com/post/automating-deployment-adaptive-analytics