Using WebSphere Cast Iron with OAuth to access Google APIs and Amazon Redshift, Part 2: Creating a Cast Iron orchestration

Giovanni Vigorelli (giovanni@nz.ibm.com)
IT Specialist
IBM

Jervis Lee (jervisl@tw.ibm.com)
IT Specialist
IBM

12 March 2015

This two-part tutorial series demonstrates the flexibility of WebSphere® Cast Iron by integrating with popular cloud services such as Google® and Amazon® Web Services. The tutorial series shows how to authenticate to Google Analytics via Google APIs and OAuth V2, and how to use a JDBC connection to copy data to the Amazon Redshift database. Part 2 describes how to compose the Cast Iron orchestration that copies data collected by Google Analytics into the Amazon Redshift database.

Introduction

In Part 1 of this tutorial series described how you can configure Google Analytics and OAuth in Google API and Amazon Redshift. You then created Cast Iron Integration endpoints to leverage all the components that were configured. In Part 2 of this series, you will use the previous endpoints to copy data collected by Google Analytics into the Amazon Redshift database.

As mentioned in Part 1, this tutorial does not describe Cast Iron development best practices, nor a sophisticated integration pattern. Instead, a simple orchestration will be implemented. The focus is more on connectivity and security against the target systems (see Figure 1).
Building a simple orchestration

In the same project where the HTTP and database endpoints have been created, let's create a simple integration flow (orchestration).

1. Creating a project in Cast Iron Studio, a default orchestration is created. Rename it with a meaningful name, such as GoogleAnalyticsToRedshift and open in the editor.
2. Once the project has been created, drag the following activities into the orchestration:
   - HTTP - Receive Request: Expose the orchestration via REST service.
   - HTTP - Get Request: Make a REST call to Google Analytics.
   - Transform - Read JSON: Transform the Google response from JSON to a variable.
   - Transform - Map Variable: Map just the interesting part of the Google response to a useful variable.
   - Transform - Apply XSLT: Transform the type, format, of the previously used variable into another one that suits the next activity.
   - Logic - For Each: Iterate actions to the elements of the previous variable array.
   - Database - Insert Rows: Eventually write the data gathered from Google into the Redshift database. The orchestration should look like Figure 2.

Note: This flow is not optimized and it does not claim to follow any best practice. The only purpose of the flow is to move data from one site to another. Nevertheless, there are a few interesting situations, like mapping the JSON data into a Cast Iron variable and transforming an unstructured XML data into a structured array.

Let's configure each activity.
Configuring HTTP - Receive Request

Using a "Receive Request" is the simplest way to test an orchestration. You can use any HTTP client to trigger the integration flow at any time. Cast Iron Studio offers a flexible way to add, remove, or change activities.

There are other scenarios, probably closer to real world situations, in which "Receive Request" is replaced with "Schedule Job" and synchronizes the data once a day or when the business organization requires it.

Clicking on the Receive Request activity, the bottom section of Studio is populated with the configuration checklist of the object itself. Each item of the checklist is explained (see Figure 3):

1. **Summary**: Provide a meaningful name to the activity, such as HTTP – Receive Request.
2. **Pick an endpoint**: In Part 1: Defining a Cast Iron – Google Analytics endpoint and Part 1: Defining a Cast Iron – AWS Redshift endpoint, two endpoints have been created. Those are access points to an external entity to the flow. A third one is necessary to expose the orchestration via the HTTP protocol to expose the logic of the flow as a REST API. Click on New and define the TCP IP port from which the orchestration will be exposed (Figure 3). If 80 is adequate, leave all the defaults and confirm. It is a good practice to give meaningful names to the endpoints, such as HTTPListener.

![Figure 3. HTTP endpoint configuration](image)

3. **Configure**: This is an interesting step. This option is mainly used to define the URL used by the HTTP client to invoke the orchestration. The URL can contain parameters and query strings. Studio is quite sophisticated in that you can add a real query string (from an actual call), and then Cast Iron Studio transforms the names of the parameters into a variable. This way, you do not need to code anything.

Fill the **URL** with the string used to test the HTTP-GoogleAPI endpoint as shown in Figure 4:

```
/analytics/v3/data.ga?ids=ga:91647549&dimensions=ga:browser,ga:city&metrics=ga:sessions,
ga:bounces&start-date=2014-09-24&end-date=2014-10-08
```
Figure 4. HTTP endpoint configuration

![HTTP endpoint configuration diagram]

**Note:** Do not check "Requires a Reply" (Figure 4) as this is an asynchronous call. The orchestration will reply as soon as the HTTP call is done and then the flow will be executed. If the box is checked, the integrating flow requires a Send Response activity.

4. **Map outputs:** In this step, the parameters and query strings passed by the HTTP client are stored to a local variable for a later reuse to call the GoogleAnalytics API. As mentioned earlier, this is the simplest way to test the project. One scenario is that the dates are not passed as parameters, but they are calculated based on the current date (scheduling scenario).

Notice in Figure 5 that the "queryParameters" are the same as the parameters defined in the URL of the previous step. Click on **Copy** and select **httpheaders** from the list. This way, a new variable or httpheaders type is created and available in the orchestration for further use. Because you are exposing an HTTP GET call (even if you are accepting any method), you can ignore the body of the call.
Automatically, Studio matches the source and destination fields of the copied variable. Rename the variable with a meaningful name, such as `GoogleAnalytics_queryParam` as shown in Figure 6.

**Figure 5. Map parameters**

```plaintext
Automatically, Studio matches the source and destination fields of the copied variable. Rename the variable with a meaningful name, such as `GoogleAnalytics_queryParam` as shown in Figure 6.
```

**Figure 6. Variable mapping**

**Configuring HTTP – Get Request**

It is now time to make use of all the effort spent in the first section of the tutorial. You can use Get Request to make REST calls. Cast Iron Integration provides connectors that make the integration to cloud services (for example, SalesForce®.com, Oracle® CRM OnDemand) and package applications (for example, SAP®, JDE) extremely easy. Services, for which there is not
a specialized connector, can be integrated by developing a custom connector or using generic activities such as HTTT and Web Services. To consume Google Analytics API, use an HTTP GET.

Clicking on the Get Request activity, the bottom section of Studio is populated with the configuration checklist of the object itself. Each item of the checklist is explained below:

1. **Summary**: Give a meaningful name to the activity such as **Google Analytics API**.
2. **Pick Endpoint**: Browse and select the endpoint created in Part 1: Defining a Cast Iron – Google Analytics endpoint, such as **HTTP-GoogleAPI** as shown in Figure 7.

   **Figure 7. Endpoint selection**

3. **Configure**: Leverage the variable by adding the query parameters defined in the first activity. In the "Request URL", add following string as shown in Figure 8:

   
   ```
   /analytics/v3/data/ga/?ids={ids}&start-date={start-date}&end-date={end-date}&dimensions={dimensions}&metrics={metrics}
   ```

   **Figure 8. URL configuration**

   The names in the curly brackets are the attributes name of the query parameters defined in the HTTP Receive Request (Figure 9). This way, you do not have to hardcoded anything.
Figure 9. Query parameters

4. **Map Inputs:** In this configuration step, the information needed to execute the activity is provided in the form of a variable. Click on **Select Inputs** and from the list, choose **GoogleAnalytics_queryParam**, which is the variable created in the previous activity (see Figure 10). Once done, the “From Orchestration” section is populated with the variable.

Figure 10. Map input parameters

At this stage, drag the **httpheaders** root element of the GoogleAnalytics_queryParam variable to **httpheaders** in the right panel of the Map Inputs page as shown in Figure 11.

Figure 11. Mapping variables

Because the parameters names and types match, all the parameters are automatically mapped (see the green lines in Figure 11).

5. **Map Outputs:** As mentioned already, this tutorial does not cover anything more than proof of work. Therefore, basic activities, such as error handling and exception path, are not covered. For this reason, all the essential information returned by the HTTP call will not be described,
such as the response code, the response error, and so on. Just the GoogleAnalytics data, body, will be available for the orchestration.
Click on Copy and select body from the list as shown in Figure 12. This way, a new variable or type string is created and available in the orchestration for further use. Automatically, Studio matches the source and destination fields of the copied variable. Rename the variable with a meaningful name such as GoogleAnalytics_res_body.

Figure 12. Mapping parameters

Configuring Transform – Read JSON

If the HTTP call to Google API is successful, the data from Google Analytics is available to the orchestration. However, there is a problem to address. While the response from Google is a JSON object, this is represented by a string: body. This is not easy to handle if the granular activities, like field mapping or transformation, need to be done. Cast Iron can help with this by providing an activity to map a "string-ified" JSON to a proper structure.

When you click on Read JSON, the bottom section of Studio is populated with the configuration checklist of the object itself. Each item of the checklist is explained below. Give a meaningful name to the activity such as Read Google Api body.

1. Configure: In order to configure this activity, you need a sample JSON response. With a sample JSON response, Studio can create a JSON schema, and then any runtime call is mapped to a structure using this schema. To get a sample of the Google Analytics response, make a call to Google API using the same query parameters, which are used by the runtime.

Listing 1 shows a sample response.

Listing 1. Sample response

```json
{
    "kind": "analytics#gaData",
    "rows": [
        ["Chrome", "Auckland", "1", "0"
```
Using WebSphere Cast Iron with OAuth to access Google APIs and Amazon Redshift, Part 2: Creating a Cast Iron orchestration

```json
{
    "containsSampledData": false,
    "profileInfo": {
        "webPropertyId": "UA-54932165-2",
        "internalWebPropertyId": "88253130",
        "tableId": "ga:91647549",
        "profileId": "91647549",
        "profileName": "All Web Site Data",
        "accountId": "54932165"
    },
    "itemsPerPage": 1000,
    "totalsForAllResults": {
        "ga:bounces": "0",
        "ga:sessions": "5"
    },
    "columnHeaders": [
        {
            "dataType": "STRING",
            "columnType": "DIMENSION",
            "name": "ga:browser"
        },
        {
            "dataType": "STRING",
            "columnType": "DIMENSION",
            "name": "ga:city"
        },
        {
            "dataType": "INTEGER",
            "columnType": "METRIC",
            "name": "ga:sessions"
        },
        {
            "dataType": "INTEGER",
            "columnType": "METRIC",
            "name": "ga:bounces"
        }
    ],
    "query": {
        "max-results": 1000,
        "dimensions": "ga:browser, ga:city",
        "start-date": "2014-09-21",
        "start-index": 1,
        "ids": "ga:91647549",
        "metrics": [
            "ga:sessions",
            "ga:bounces"
        ],
        "end-date": "2014-09-25"
    }
}
```
Because Google API requires OAuth for authentication, an OAuth call needs to be done. There are few ways to do that. The two easiest are:

- Testing the orchestration and getting the JSON response from the Verify tab.
- Using the Google OAuth v2 playground.

Either way, you need to get a response like JSON above. Once JSON is available, copy that into the **Use JSON or XML Message to generate output schema** input field, and then click on the **Generate Output Map** button as shown in Figure 13.

**Figure 13. Generate map**

2. **Map Inputs**: This is the same concept described in the previous sections. Click on **Select Inputs** and from the list, choose **GoogleAnalytics_res-body**, which was the variable created in the previous section. Once done, the “From Orchestration” section is populated with the variable.

Drag **GoogleAnalytics_res-body** to **JSONText** in the right panel of the Map Inputs as shown in Figure 14.
3. **Map Outputs**: This is the same concept described in the previous sections. Click on **Copy** and select **body** from the list (Figure 15). This way, a new variable is created and available in the orchestration for further use. This variable represents the JSON response from Google Analytics, but in a format (XML) that is easier to use in the orchestrations. Automatically, Studio matches the source and destination fields of the copied variable. Rename the variable with a meaningful name such as `GoogleAnalytics_outputXML`.

**Figure 15. Map parameters**

---

**Configuring the Transform – Map variable**

In this scenario, a subset of the information returned by Google API needs to be stored in the target database. It is convenient to work with a simpler data structure than the entire Google API response.

To accomplish that, you need to import a data structure (XML Schema) with the target data model (Figure 16). Then you can map a subset of a variable by using drag and drop.
Figure 16. Relevant response portion

Based on the above structure (kind, rows) shown in Figure 16, the following XML Schema has been created (Listing 2). Save the XML Schema as GoogleAnalytics_data_rows.xsd. In the right panel of Studio, click the Project tab and click the Add Document button. Select the schema you want to import.

Listing 2. XML schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Document">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="kind" type="xs:string"/>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="rows" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Note: To continue to follow the instructions below, save the XML Schema as GoogleAnalytics_data_rows.xsd.

1. Click on Map Variables and give a meaningful name to the activity such as Map GoogleAnalytics Rows. The bottom section of Studio is populated with the configuration.
2. Select GoogleAnalytics_outputXML as the input variable as shown in Figure 17.
3. As the output variable, create a new variable based on the schema just imported. Name the variable `GoogleAnalytics_data_rows` as shown in Figure 18.

**Figure 18. Select variable type**

![Select variable type](image1.png)

4. Map **JSON** to **Document** as shown in Figure 19.

**Figure 19. Variable mapping**

![Variable mapping](image2.png)

**Configuring Transform – Apply XSLT**

Before continuing, let's review our goal. The objective of this integration is to get city, browsers, sessions, and bounces from Google Analytics and to store them into a database table. That information comes from a Google response in the "rows" elements of the JSON object. Unfortunately, "rows" is not structured as shown in Listing 3.
Listing 3. Google response

```
"rows": [
  ["Chrome", "Auckland", "1", "0"],
  ["Firefox", "Milan", "1", "0"],
],
```

1. In order to work at the field level (city, browsers, sessions, and bounces), you need to transform and map "rows" into an XML structure. To accomplish this, an XSLT transformation is used. Create a file named `GoogleAnalytics_data_rows2table.xslt` and copy the code shown in Listing 4.

**Listing 4. GoogleAnalytics_data_rows2table.xslt**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes" />
  <xsl:strip-space elements="*" />
  <xsl:template match="/Document">
    <Document>
      <xsl:for-each select="rows">
        <xsl:if test="position() mod 4 = 1">
          <xsl:variable name="pos" select="position()" />
          <xsl:variable name="relativePos" select="($pos mod 4) - 1" />
          <Row>
            <xsl:element name="browser">
              <xsl:value-of select="." />
            </xsl:element>
            <xsl:element name="city">
              <xsl:value-of select="../rows[$pos + 1]" />
            </xsl:element>
            <xsl:element name="sessions">
              <xsl:value-of select="../rows[$pos + 2]" />
            </xsl:element>
            <xsl:element name="bounces">
              <xsl:value-of select="../rows[$pos + 3]" />
            </xsl:element>
          </Row>
        </xsl:if>
      </xsl:for-each>
    </Document>
  </xsl:template>
</xsl:stylesheet>
```

2. Import the XSLT into the Cast Iron project by selecting **Add Document** in the **Project** tab. The result of the XSLT above is an XML structure. This structure needs to be mapped to a variable in Studio. The variable type is a complex one, thus you need to import an XML Schema (XSD) that represents the type.

3. Create a file named `GoogleAnalytics_data.xsd` and copy the code shown in Listing 5.
Listing 5. GoogleAnalytics_data.xsd

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Document">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Row">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="browser" type="xs:string" />
              <xs:element name="city" type="xs:string" />
              <xs:element name="sessions" type="xs:int" />
              <xs:element name="bounces" type="xs:int" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

4. Import the XSD into the Cast Iron project by selecting Add Document in the Project tab. Create a new variable based on this schema (Figure 20).

**Figure 20. Create variable**

5. By clicking on the Apply XSLT activity, the bottom section of Studio is populated with the configuration checklist of the object itself. Each important item of the checklist will be explained. Give a meaningful name to the activity, such as Cast Rows to data structure.

6. **Pick Stylesheet:** Click Browse and pick the XSLT that was imported previously, **GoogleAnalytics_data_rows2table**, as shown in Figure 21.
Figure 21. Select stylesheet

7. **Set input and output**: Select `GoogleAnalytics_data_rows` as the Input variable as shown in Figure 22.

Figure 22. Select input

8. Select `GoogleAnalytics_data` as the output variable as shown in Figure 23.

Figure 23. Select output

**Configuring Logic – For Each**

The data is now available in the expected format and you can now insert records in the database. There are several ways to accomplish this. The easiest way is to iterate the row array from the
"GoogleAnalytics_data" data and for each item to insert a row into Redshift. Cast Iron offers a set of pre-built activities to add such logic to the integration flow. "For Each" makes the iteration of an array very simple.

By clicking on the **For Each** activity, the bottom section of Studio is populated with the configuration parameters of the object itself (Figure 24).

- **Activity Name:** Give a meaningful name to the activity such as `row`.
- **Variable name:** Browse and select the Output variable of the previous activity, `GoogleAnalytics_data`.
- **Element Name:** Click on the ellipses and click on `Row`.
- **Variable Name:** This variable contains the item value of each iteration.

**Figure 24. For Each configuration**

![For Each activity configuration](image)

**Configuring Database – Insert Rows**

This is the last activity of the orchestration. The data has been extracted from Google Analytics and manipulated to easily write records to a database. This orchestration can be quickly adopted for other scenarios, such as writing to a file, putting a message in a queue, replying to the initial HTTP Request, and many more.

In the context of this tutorial, Redshift is used as a master repository, so data from Google Analytics and possibly other sources are stored in the database for later analysis.

By clicking on the **Insert Rows activity**, the bottom section of Studio is populated with the configuration checklist of the object itself. Each item of the checklist is explained below:

1. **Summary:** Give a meaningful name to the activity such as `Insert Rows`.
2. **Pick Endpoint:** Browse and pick the endpoint created in **Part 1: Defining the Cast Iron – AWS Redshift endpoint, DB-Redshift**.
3. **Configure – pick table:** Here, the database activity allows you to select the necessary objects to easily insert the data into the table (Figure 25). Select the database schema (such as `public`), browse to find the target table (such as `googleanalyticsdata`), and select all the columns that will be affected by the database action.
4. **Map Inputs**: Select row as the input variable and map the fields to the target columns as shown in Figure 26.

**Figure 26. Variable mapping**

---

**Testing and running the orchestration**

At this stage, your work is done. If everything has been properly configured, run the orchestration. In Studio, play the orchestration in the **Verify** tab.

Once the orchestration is running, make a REST call to the HTTP service exposed by your Cast Iron application. As soon as it is completed, check in the Redshift instance that the records extracted from Google Analytics have been added. The test is shown in Figures 27 to 29.
Figure 27. HTTP REST client

Figure 28. Orchestration test in Studio
Troubleshooting tips

With Cast Iron Studio, you can check error messages and variable values of each activity of the orchestration in the "Variable/Parameter Message Data" section after you verify the orchestration.

Another source of information is the "error.log" file in the Studio installation directory, such as C:\Program Files\IBM\WebSphere Cast Iron Studio 7.0.0.1.

If Designer WebStart (Studio from Cast Iron Live) is used instead, the "error().log" file is located in the .castiron directory in the user's home directory such as C:\Users\Administrator\.castiron.

Conclusion

In the last part of this tutorial series, you configured an orchestration that retrieved data from Google Analytics by using OAuth as an authentication mechanism. You then copied that into a relational database. Even if the flow is straightforward, you learned about a few interesting product features, such as an HTTP Query String, XSL, and graphical variable mapping.

The implementation of the scenario is not a "production ready" prototype, but we believe that several aspects of this tutorial series can help you better understand how to configure and implement a Cast Iron orchestration in the context of this fictional scenario.
Resources

- IBM WebSphere Cast Iron V7 documentation
- Google Analytics
- Google API
- Amazon Redshift
- SQL Client
- URL Decoder/Encoder
- developerWorks WebSphere resource page
About the authors

Giovanni Vigorelli

Giovanni Vigorelli is currently a Pre-sales Specialist in the middleware brand. He works with clients as a technical specialist. Prior to that, he worked as a Subject Matter Expect in WebSphere by providing support to clients.

Jervis Lee

Jervis Lee is a WebSphere Cast Iron L2 Support Engineer at the IBM Taiwan Software Development Lab. He has worked as a technical leader of the WebSphere Transformation Extender Performance Testing team. His expertise covers performance testing, web application, development, and information security.

© Copyright IBM Corporation 2015
Trademarks
(www.ibm.com/developerworks.ibm/trademarks/)