**YOUR LOGO**

**Weather service REST adapter**

**Jul 6, 2019**

**Version 1**

| REVISION HISTORY | | | |
| --- | --- | --- | --- |
| DATE | VERSION | DESCRIPTION | AUTHOR |
| Jul 6, 2019 | 1 | Initial document version | Hernán Camilo Quevedo Barrero |
|  |  |  |  |

Table of Contents

[Weather service REST adapter 1](#_Toc13470161)

[Jul 6, 2019 1](#_Toc13470162)

[Version 1 1](#_Toc13470163)

[INTRODUCTION 3](#_Toc13470164)

[PURPOSE 3](#_Toc13470165)

[INTENDED AUDIENCE AND PERTINENT SECTIONS 3](#_Toc13470166)

[PROJECT SCOPE 3](#_Toc13470167)

[REFERENCES 3](#_Toc13470168)

[DESCRIPTION 4](#_Toc13470169)

[PRODUCT PERSPECTIVE 4](#_Toc13470170)

[FEATURES 4](#_Toc13470171)

[USER OVERVIEW 4](#_Toc13470172)

[OPERATING ENVIRONMENT 4](#_Toc13470173)

[ASSUMPTIONS / DEPENDENCIES 4](#_Toc13470174)

[ARCHITECTURE AND DEPLOYMENT 5](#_Toc13470175)

[RESTful CONSIDERATIONS 6](#_Toc13470176)

# INTRODUCTION

## PURPOSE

The purpose of this document is to explain the technicalities of the REST adapter written for a legacy web-service.

## INTENDED AUDIENCE AND PERTINENT SECTIONS

This document is intended for a technical audience that will use the web-service adapter as a means to wrap for modernization of their Weather SOAP web-service.

## PROJECT SCOPE

Inside what is most likely a transformation initiative inside the organization, the scope of this product (the web-service adapter) is to deliver a working example of an “*apification*”, a new term that represents the efforts to a migration journey towards a more maintainable, flexible and efficient way to re-write services that were originally written as SOAP-based services.

## REFERENCES

1. Richardson Maturity Model

<https://restfulapi.net/richardson-maturity-model/>

1. Words by Martin Fowler about the previous RMM

<https://martinfowler.com/articles/richardsonMaturityModel.html>

1. What is REST?

<https://restfulapi.net/>

# DESCRIPTION

## PRODUCT PERSPECTIVE

Originally as a proof of concept for clients, this work was born out of our knowledge in integration tools; integration is still as relevant as fifteen years ago, when the SOA initiatives were being seen as rather disappointing. Integration is wide and convoluted area of software intensive systems that needs a keen eye and good amount of experience and expertise. We proud ourselves in having both and it is out opinion that the approach to enterprise-wide integration from a cultural and process perspectives is more important than tools, but tools, ultimately are what end up materializing the view of business processes.

MuleSoft has achieved a good name and its ESB product is becoming a very important player in the market. Although proprietary, it offers concrete and effective ways to integrate in an agile way. This service adapter is proof of that.

## FEATURES

The client has a Weather SOAP-based web-service, and through Mule ESB and Anytime Studio IDE, a RESTful adapter was done in order to simplify its consumption, from the perspective of simpler interfaces, much less verbose output and input and maintainability as well as modifiability.

## USER OVERVIEW

The user is typically another actor system, a client, that consumes HTTP endpoints with URI parameters and using the GET HTTP method. This would be accomplished through POSTMAN or other REST client.

## OPERATING ENVIRONMENT

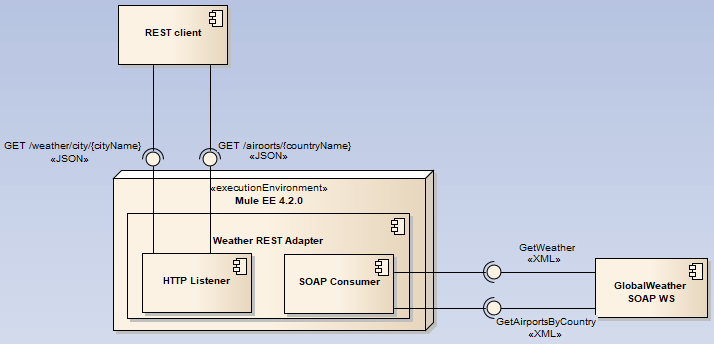
As long as the final user, which is a technical user, has Node and Mule runtime 4.2.0 installed, the modified SOAP code (more on this below), this service adapter will work normally. The original resource given by the client for their SOAP-based Weather web-service was offline almost a year ago, so I used the Node version to run it locally. Some modifications were in order to mock the responses of the original Weather service.

## ASSUMPTIONS / DEPENDENCIES

The original SOAP-based Weather service must be running locally on localhost and port 8080, as the source code, given by the client, indicates.

## ARCHITECTURE AND DEPLOYMENT

The design is fairly simple: a REST wrapper service that follows best practices for RESTful APIs and consumes the SOAP-based service transforming the inputs and outputs from verbose SOAP protocol inputs and outputs, to Hypertext transfer protocol and JavaScript Object Notation output.



The runtime where the REST adapter resides is Mule EE 4.2.0, and, through a series of components given by the compile and runtime libraries, there is component that listens to HTTP requests on a particular port in the local machine, capturing the input as URI parameters or path variables to set the input of another component: a SOAP consumer that invokes the SOAP-based service. A final component, not shown above, transforms the result to a JSON object that is returned by the HTTP listener. This component is using DataWeave expression language. This kind of wrapping around legacy services is what some in the industry call “*apify*”.

**NOTE\_1**: the client’s service has one operation that is actually a misnomer: the operation GetCitiesByCountry is not returning an array of cities but an array of airports, Australian airports, so the resources for the REST endpoint are different: one for weather of a city (just Melbourne), and another one for airports of a country.

**NOTE\_2**: the Node implementation of the WS returns the mock data for both operations as unescaped XML data that caused DataWeaver transformation errors in the form of unescaped XML CDATA being returned to the REST client.

This was resolved using NPM unescape module, adding it to package.json. Another way could have been to escape the characters but I decided to go with the node code because I like node. I found no way to adapt that character data within DW 2.

## RESTful CONSIDERATIONS

The Richardson Maturity Model is an excellent source for the start of a journey of the integration problems that enterprises face. “*The web is an existence proof of a massively scalable distributed system that works really well, and we can take ideas from that to build integrated systems more easily*”.

Level 1 of maturity for REST: *resources*

The RESTful approach to this humble solution comes from the consideration of resources in the problem domain rather than actions in the form of functions that translate directly into Remote Procedure Calls. The client’s SOAP-based service is just that: a way of calling logical functions on a remote interface that extracts its data from someplace else and aggregates it.

This REST adapter forms a very small API around these functions calls in which there are resources, inferred from the nouns in the problem domain. Generally speaking, nouns that form collections around the most general and convergent aspect or entity in the domain problem are the best candidates for resources in the API design.

For instance, weather instead of cities, and airports instead of countries, since most conversations about the domain would converge to /weather itself, not cities or countries; it is transversal, not to mention, the client’s original service talks about weather.

For the second resource, given the confusion of cities vs airports, the resource name is /airports; the collections of airports that can be queried by country.

Level 2 of maturity for REST: *HTTP verbs*

Both endpoints account for the GET HTTP verb, that does not make any change to the state of the resource and gives the benefit of allowing faster queries enabled by caching. No other verb is added since there was not in the scope of the proof of concept. Nevertheless, POST would be used to add airports in one case, and add a weather report in some sort of event sourcing fashion to a city’s weather collection, returning HTTP status 201. PUT would be done to update existent airports and weather reports of a city, in an idempotent fashion; PATCH to amend a data about a subordinate of a collection; DELETE to remove a resource, and finally GET to query. All of these according to RFC[[1]](#footnote-1).

Level 3 of maturity for REST: *HATEOAS*

Given that for this proof of concept there is no other possibilities for a resource after an HTTP call was executed on a resource, additional information about other HTTP calls applicable to that resource are not possible, but it is a common practice for RESTful APIs that are complete and self-documenting.

## REST API in action

On Github repository <https://github.com/hernihub/weather-adapter> you could also find the following instructions to run the project:

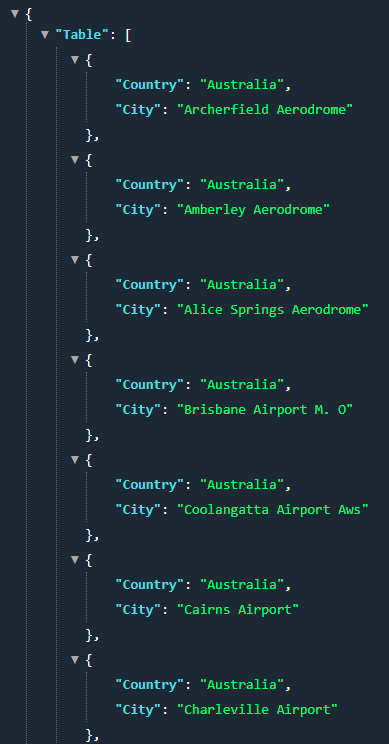
1. NODE
   1. Clone the aforementioned repository
   2. Go to weatherExcerciseDockerFile folder and run npm install
   3. Got to weatherExcerciseDockerFile folder and execute build the docker image and run it as a container or, simply run node server.js to start the Weather SOAP WS
2. MULE EE RUNTIME
   1. Go to <https://www.mulesoft.com/lp/dl/mule-esb-enterprise> and download Mule ESB enterprise 4.2.0
   2. Unzip the downloaded file to create the folder
      1. Mule EE folder
3. MULE APPLICATION
   1. Go to weather-rest-adapter/target folder and copy the JAR file weather-rest-adapter.jar into the app folder of the Mule Enterprise Standalone folder unzipped in the previous step
      1. 
   2. Be sure to have JRE or JDK 8 installed
   3. Go to /bin Mule standalone subfolder and run mule.sh in Linux or mule.bat in Windows
4. THE API
   1. Open postman or simply go to Firefox and navigate to these URLs
      1. <http://localhost:8081/weather-api/v1/weather/city/Melbourne>

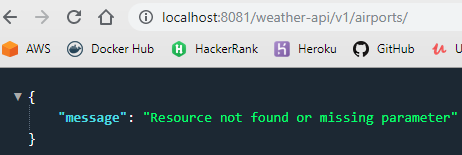
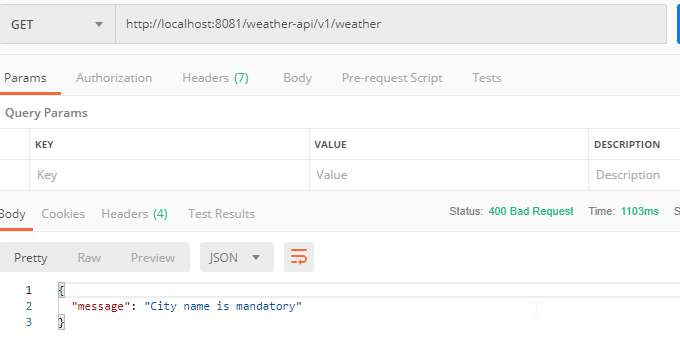
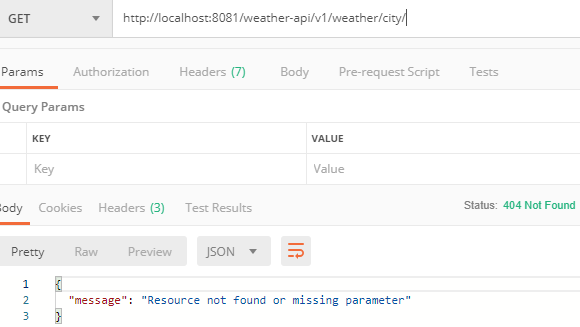
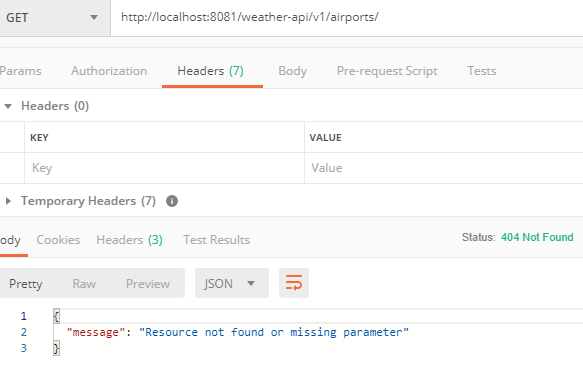
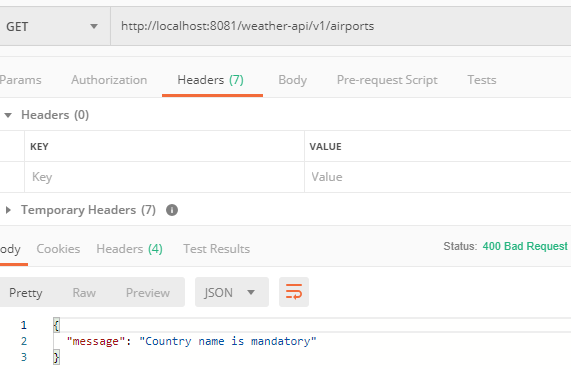
Get this response:



* + 1. <http://localhost:8081/weather-api/v1/airports/Australia>

Get this response:



1. THE EXCEPTIONS
   1. If the name of the city or of the country is missing, a HTTP 404 status code with the following message will print:
      1. 
   2. Any other call to an URL that is registered in the RAML definition and does not conform to the resource pattern, will end up in either bad request HTTP 400 or 404
      1. 
      2. 
      3. 
      4. 

## CHALLENGES

I’ve only heard of Mule, never worked with it before; DataWeave expression language was considerably more complex than connectors or flows since I’ve worked a long time ago with Java CAPS enterprise designer and it was just like this, except than it was BPEL. I found some features particularly difficult and candidates for refactoring in the future by the Mule team, like going into flow XML a lot of the time (when bad requests are returned, there is this <ee:variables> element that is not shown on the graphical flow). I faced this as usual when looking for new technologies’ viability: research and more research online and common sense, of course, formed through the years. Mule forum was particularly helpful for obvious reasons. I wrote questions and later answered them myself, learning more than expected in the process and putting my humble contribution out there for the community.

It is very powerful, this Mule ESB. Certainly, one can think of as a risky choice since the debacle of ESB back in the SOA era, but they are doing a great job.

I really hope this short exercise convince you guys that I can be a great member of your integration team; I’ve dedicated the majority of my professional career to integration and I really want to be part of such a prestigious employer as Deloitte is.

Thank you.

1. <https://www.ietf.org/rfc/rfc2616.txt> [↑](#footnote-ref-1)