C4C-SBS-Extensions Refactoring POC functions Approach

Contents

[Abstract 1](#_Toc110622550)

[Flow Diagram 2](#_Toc110622551)

[Code configurability 2](#_Toc110622552)

[Function code refactor approach 3](#_Toc110622553)

[Function Pseudocode 4](#_Toc110622554)

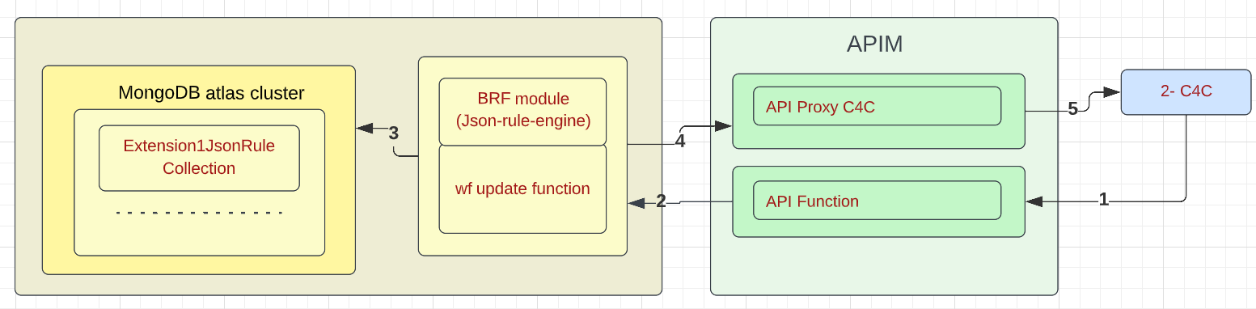
# Abstract

This document covers the key aspects to refactor the POC’s under C4C-SBS-Extensions Atlas MongoDB app services. Major issues with the current code are for all three different POC’s we have specific functions created for update, get, patch, and execute rule function. Whilst refactor the code the overwhelming number of functions should be able to reduce, and the function can make very generic by-passing right arguments and configurations. This document describes how the different functions can be standardized and make it generic. It also covers the configurability aspects to fetch values from a configuration value store available in MongoDB Atlas app services.

**Before we start:**

First let’s agree that our proposed solution design for the stateless service consists of only 1 function.

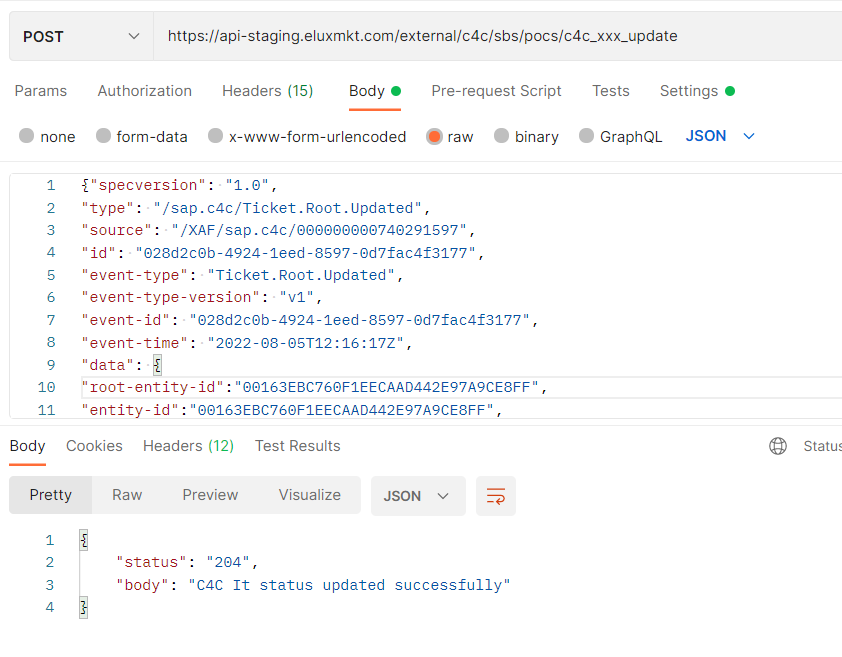
So following our first design, we can easily have a 1 function per usecase.



And here is an example of 1 single function “c4c\_xxx\_update” for poc3, tested and working.

[Functions - c4c\_xxx\_update | App Services (mongodb.com)](https://realm.mongodb.com/groups/6217649ee89133230f422f4d/apps/62a73145bf12d9cf6f4a841b/functions/62ecfc986f47d57c331a58da)

<https://api-staging.eluxmkt.com/external/c4c/sbs/pocs/c4c_xxx_update>

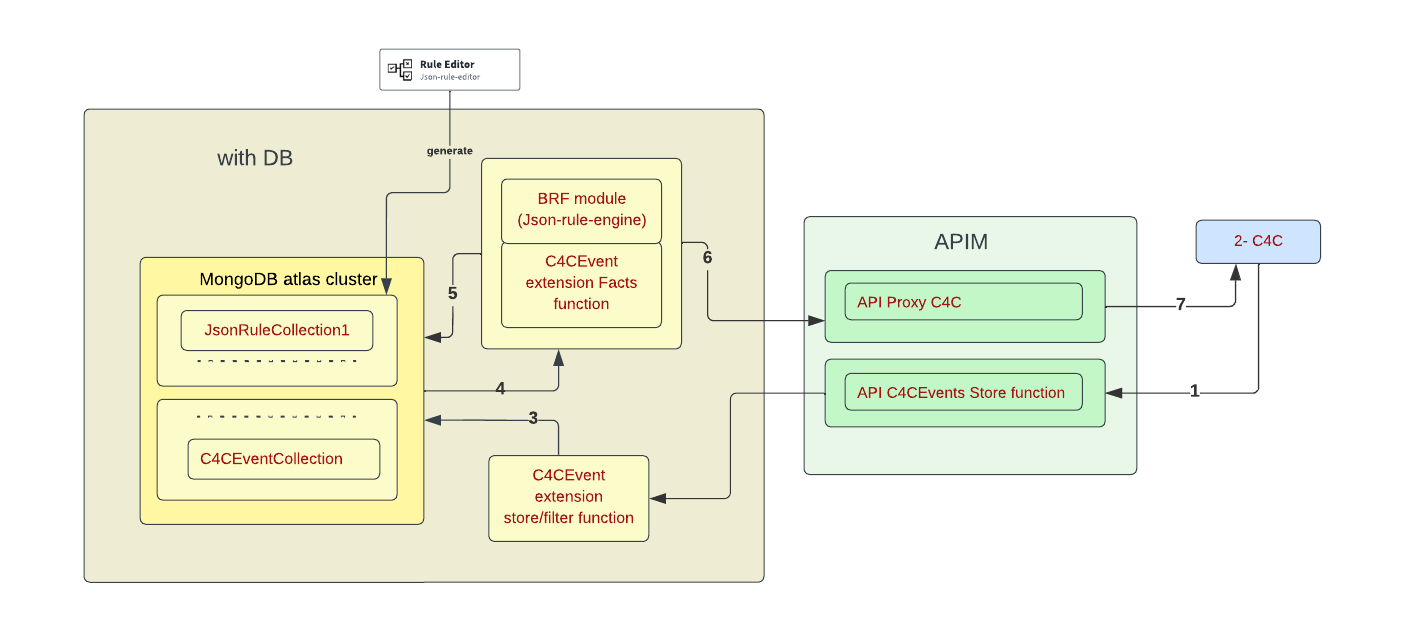


# Flow Diagram

the plan is to enable error handling, so to do that we can first start by configuring the “C4C-Event Extensions Facts function” or Basically the “UPDATE” function so that it can handle the multiple pocs we have.

We can as well introduce the “façade function” or “HTTP” function in the solution.

But it will mainly be used to consume the event, filter and store the payload in the database so that a DB trigger will execute the 2nd lambda, but for now we directly execute the update function.



Note: The Payload that we are going to store later in the DB for Error handling will now be passed between the functions.

First step:

**Diagram

Description automatically generated**

* We need to have one function to act as FACADE or Interface to "store/filter function” for all use cases. This will be triggered by an http trigger/ call from c4c, then executes a separate function for the update as a first step or stores the event payload in DB collection to be processed at a later stage.
* The "Facts/update function” to do all 4 steps: get c4c ticket data, get rule object and configuration from DB, validate ticket, patch c4c. This function should be generic so that only 1 lambda can execute multiple rules for the different use cases. If the use case is a complex and cannot be done using the common update function, only then we can create a dedicated service for the workflow.

# Code configurability

The code must be configurable. It should not contain any hardcoded values. All environment specific values must be stored in configuration value store in Atlas MongoDB app services. The secrets also must be stored and used inside the code, but it can be fetched from environmental specific secrets.

Few examples are shown below.

Graphical user interface, text, application, email

Description automatically generated

- values like : url, key, collection, query : select, filter, expand related to c4c calls, strings and values related the use cases etc.. should be stored in a values file.

A values file per use case.

- values related to the ruleset: should be stored in the same db object as the rule so that we don't have any hardcoded values to be able to use the rule to validate the events.

A db object for use case for the rule and other config values related to the rule.

- the execute function/method needs to accept the rule name as a parameter.

- we need to expose objects in the return of a function or operation , and not return specific fields related to different flows.

We can cleanse the c4c response objects and return them to the next step instead of the current way.

# Function code refactor approach

1. **The FAÇADE function.**

* The FAÇADE function will be named to **c4c\_customer\_serviceticket\_facade**.

Why: c4c ? All the functions are already starting with c4c, the ODL application also following c4c prefixes.

What does customer represent here ? customer\_serviceticket record is what we get and patch based on rule execution

Why façade ? why not just HTTP ? Façade is the naming standard we will use generally to identify we follow that design pattern.

* This function will have request and response parameters and will be implemented as a non-blocking asynchronous operation. This function will be accessed by c4c workflow logic via APIM layer which acts as the gateway and pass-through proxy.
* Do the condition check specific to the POC based on this check the field value, if its already updated to the required value log and skip the entire execution.
* The Skelton FAÇADE function will be as follows, which inspects the incoming rule and based on which the FACTS function will be invoked by passing the required parameters.

The decision of the rule should happen in the Facts function..In the context of error handling, why would we need to identify the rule before storing the event in the database ?

That is what we will do but the rule details are already known at FAÇADE so we can easily form projection also from there and pass it to FACTS

For that reason, we need to make sure both implementations of step 1 and 2 are aligned.

Note: The event Payload that we are going to store later in the DB for Error handling will be passed between the functions.

Text

Description automatically generated

* Based on the incoming rule name, decide the patch body, projection expression and rule name to pass to FACTS function.
* The FAÇADE function must log the entire FACTS response.

The façade should not be used to validate the event payload using the json rules.

FAÇADE only inspects the payload and gets the right rule nothing else it do. We only pass the required parameters to FACTS.

The façade should only filter and if needed process the data before passing it to the update/fact function or storing it in DB to first have a stateful service and the later error handling

1. **FACTS function.**

* The FACTS method will be named to **c4c\_customer\_serviceticket\_update**
* Why: c4c ? All the functions are already starting with c4c, the ODL application also following c4c prefixes.
* What does customer represent here ? customer\_serviceticket record is what we get and patch based on rule execution

Why facts execution ? why not just UPDATE ? Its mentioned like you needed an facts execution function so named that way. But it was UPDATE earlier but FACTS function naming came from your design so kept it. There is no issue in changing from facts\_execution to update.

* Need to include parameters ticket\_root\_entityIdIn, ruleName, patchResponse = {}
* Use configuration values and secrets to get values of subscription, baseUrl, authorizationKey, proxy\_host, proxy\_path, get\_fn\_guery\_param etc all are configurable.
* Include meaningful logging message and log it for the request data comes in.
* Fire the context.http.get call to APIM get endpoint using ticket\_root\_entityIdIn value and for further JSON iteration use the **c4cResponse.d.results[0].**
* Form the input data from the **c4cResponse.d.results[0]** object like below which can be dynamic as below.
* Form the input data from the get method response object like below which is dynamic.

let inputData = {};

if (data && data.attributes && data.attributes.length > 0) {

data.attributes.map((element) => {

if (element.name === 'CountryText') {

const ServiceRequestUsedAddress = request['ServiceRequestUsedAddress'];

console.log("ServiceRequestUsedAddress", JSON.stringify(ServiceRequestUsedAddress));

inputData[element.name] = ServiceRequestUsedAddress[element.name] ? ServiceRequestUsedAddress[element.name] : '';

} else {

inputData[element.name] = request[element.name] ? request[element.name] : '';

}

console.log("inputData", element.name, request[element.name]);

});

}

* Include necessary conditional logic to execute the rule engine with right set of parameters/input JSON. The parameters which we need to pass to rule engine can be stored in values for a specific rule name.
* Execute the rule engine against the respond with execution status.
* Execute the rule engine against the input and respond with rule match found status.
* Execute the patch method using context.http.patch depending on the data to be patched for a POC when the rule match found.
* Once the patch operation succeeded make sure to form the response object with right succeeded/failed status and return to the FAÇADE function.

The "Facts function"  to do all 4 steps:

* get c4c ticket data,
* Identify the rule name for the workflow and get the rule configuration from db,
* validate event payload using workflow json rule,
* patch c4c.

This function should be generic so that only 1 lambda can execute multiple rules for the different use cases.

If the use case is a complex and can't be done using the common facts function, only then we can create a dedicated service for the workflow.

# Function Pseudocode

|  |  |  |  |
| --- | --- | --- | --- |
| **Function Name** | **Parameters** | Route | **Psuedocode** |
| c4c\_customer\_serviceticket\_facade | request, response | /c4c\_customer\_serviceticket\_facade | 1.This function should only receive events and pass a payload to the second function that will do all 4 steps  2. Use configuration values as well as database configuration options to get environmental specific variables/constants to use inside code. The subscription, base URL, authorizationKey database cluster name, query etc configurable items.  3. Include meaningful logging message and log it for the request data comes in.  4. Once the FACTS function execution completed and the response availed, necessary logging has to be added.  5. The final response must be given back to c4c by returning back the patch response. |
| c4c\_customer\_serviceticket\_facts\_execution | ticket\_root\_entityIdIn, ruleName, patchResponse = {} | /c4c\_customer\_serviceticket\_facts\_execution | 1.The FACTS method will be named to c4c\_customer\_serviceticket\_facts\_execution  2. Need to include parameters ticket\_root\_entityIdIn, ruleName, patchResponse = {}  3. Use configuration values and secrets to get values of subscription, baseUrl, authorizationKey, proxy\_host, proxy\_path, get\_fn\_guery\_param etc all are configurable.  4. Include meaningful logging message and log it for the request data comes in.  5. Fire the context.http.get call to APIM get endpoint using ticket\_root\_entityIdIn value and for further JSON iteration use the c4cResponse.d.results[0].  6. Form the input data from the c4cResponse.d.results[0] object like below which can be dynamic as below.  7. Form the input data from the get method response object like below which is dynamic.  let inputData = {};  if (data && data.attributes && data.attributes.length > 0) {  data.attributes.map((element) => {  if (element.name === 'CountryText') {  const ServiceRequestUsedAddress = request['ServiceRequestUsedAddress'];  console.log("ServiceRequestUsedAddress", JSON.stringify(ServiceRequestUsedAddress));  inputData[element.name] = ServiceRequestUsedAddress[element.name] ? ServiceRequestUsedAddress[element.name] : '';  } else {  inputData[element.name] = request[element.name] ? request[element.name] : '';  }  console.log("inputData", element.name, request[element.name]);  });  }  8. Include necessary conditional logic to execute the rule engine with right set of parameters/input json. The parameters which we need to pass to rule engine can be stored in values for a specific rule name.  9. Execute the rule engine against the respond with execution status.  10. Execute the rule engine against the input and respond with rule match found status.  11. Execute the patch method using context.http.patch depending on the data to be patched for a POC when the rule match found.  12. Once the patch operation succeeded make sure form the response object with right succeeded/failed status and return to the FAÇADE function. |

Testing Effort

There is a significant testing effort involved as all the POC must be retested whenever a new POC is introduced into the generic workflow.

Question:

How are we getting the workflow identifer ? How do we identify the usecase ?

**From the request.params[“rule\_name”] we will be able to identify the workflow identifier/rule identifier.**