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)

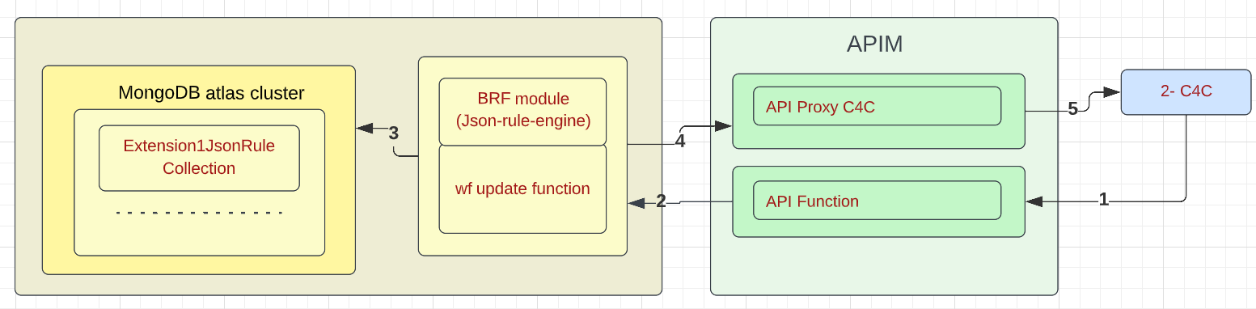
# Abstract

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.

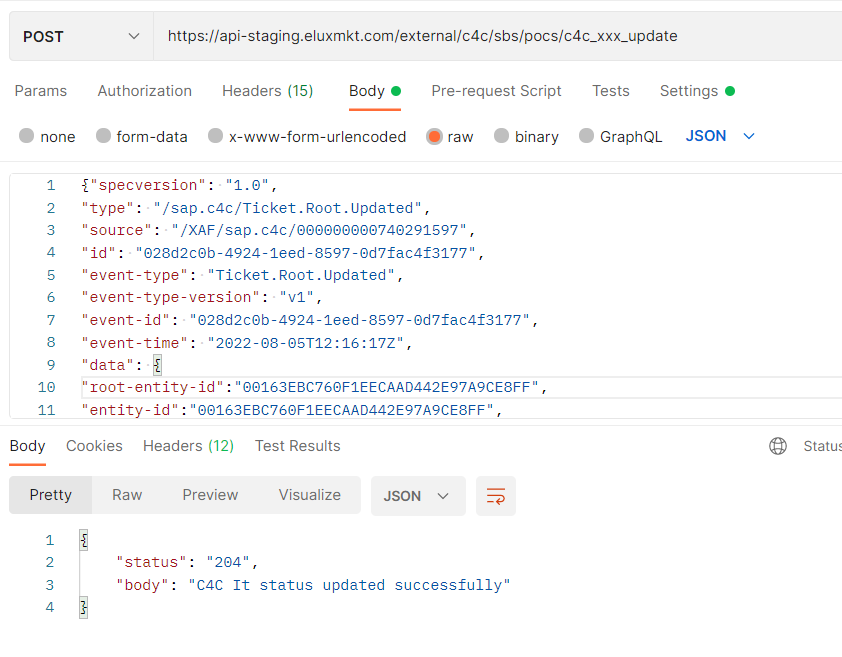
So following our first design, we can easily have a 1 function per usecase.



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>

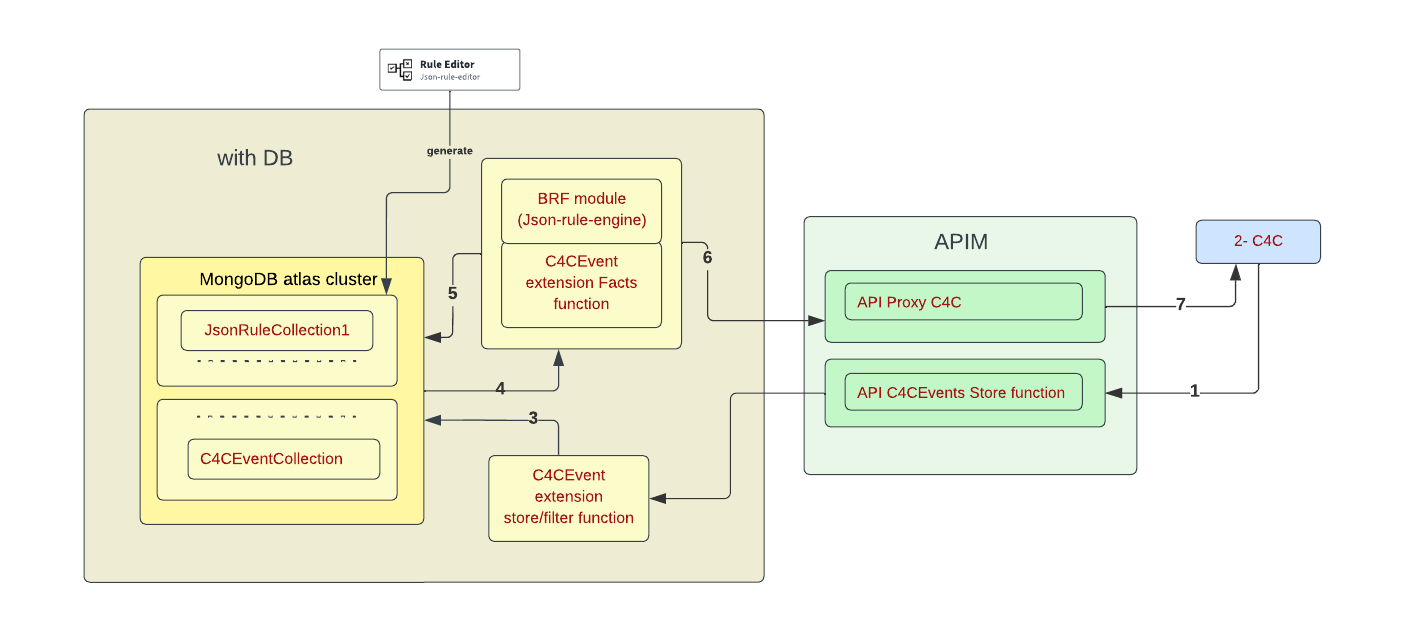


# Flow Diagram

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 functions.**

* The FAÇADE function will be named to **c4c\_customer\_serviceticket\_facade**.
* 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.
* This function will be storing the request payload along with ruleName as an event into C4CEventCollection.
* We also need to come up with a Database Trigger to invoke the real processing of this event which is described as a next step.

1. **Database Trigger to invoke FACTS function.**

* Create a database trigger which will fire the c4c\_customer\_serviceticket\_update generic function by passing ticket details from database event and workflow id from request parameter.

Graphical user interface, text, application, email

Description automatically generated

1. **FACTS function.**

* The FACTS method will be named to **c4c\_customer\_serviceticket\_update**
* Need to include parameters ticket\_root\_entityIdIn, ruleName
* 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 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.
* Log the details of patch operation at every get, execute rule & patch operations.
* Once the processing completed and successful patch operation is completed if there is rule condition met/not met remove the even from C4CEventcollection. In case of any http failures or any unprecedented errors keep the event intact in the collection for Error Handling/Reprocessing later.

Error handling/Reprocessing Logic.

* The Error handling process must identify the list of ticket\_id and ruleName to trigger and process a list of items. From the scheduled trigger call a function c4c\_customer\_ticket\_falures\_reprocess. This function will read a list of ticket\_ids which are failed to process along with the ruleName which will repeatedly call the facts function **c4c\_customer\_serviceticket\_update** to fix the all the issues. Make sure the events are purged after the reprocess completed.

Graphical user interface, text, application, email

Description automatically generated

# Function Pseudocode

|  |  |  |  |
| --- | --- | --- | --- |
| **Function Name** | **Parameters** | Route | **Psuedocode** |
| c4c\_customer\_serviceticket\_facade | request, response | /c4c\_customer\_serviceticket\_facade | • The FAÇADE function will be named to c4c\_customer\_serviceticket\_facade.  • 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.  • This function will be storing the request payload as an even into C4CEventCollection along with ruleName.  • We also need to come up with a Database Trigger to invoke the real processing of this event. |
| c4c\_customer\_serviceticket\_facts\_execution | ticket\_root\_entityIdIn | /c4c\_customer\_serviceticket\_update | • The FACTS method will be named to c4c\_customer\_serviceticket\_update  • Need to include parameters ticket\_root\_entityIdIn, ruleName  • 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 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.  • Log the details of patch operation at every get, execute rule & patch operations.  • Once the processing completed and successful patch operation is completed if there is rule condition met/not met remove the even from C4CEventcollection. In case of any http failures or any unprecedented errors keep the event intact in the collection for Error Handling/Reprocessing later. |

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.**